

SDR - Systèmes Distribués et Repartis

Résumé TE1

02 novembre 2025

Table des matières

1 Introduction aux Systèmes Distribués	2
1.1 Définition d'un système distribué	3
1.1.1 Enjeux des systèmes distribués	3
1.2 Parallélisme vs Concurrence	4
1.2.1 Parallélisme	4
1.2.2 Concurrence	4
1.2.3 Unités de traitement	4
1.3 Classification de Flynn	5
1.3.1 SISD (Single Instruction, Single Data)	5
1.3.2 SIMD (Single Instruction, Multiple Data)	5
1.3.3 MISD (Multiple Instruction, Single Data)	6
1.3.4 MIMD (Multiple Instruction, Multiple Data)	6
1.3.4.1 MIMD - Shared Memory (Mémoire partagée)	7
1.3.4.2 MIMD - Distributed Memory (Mémoire distribuée)	8
1.4 Couplage : Matériel vs Logiciel	8
1.4.1 Couplage matériel	8
1.4.2 Couplage logiciel	8
1.4.3 Couplage logiciel et exécution réseau	9
1.4.4 Exécution réseau vs Programme réparti	10
1.4.5 Couche logicielle de répartition	10
1.4.6 Système réparti	10
1.5 Propriétés d'un bon système réparti	10
1.5.1 Abstraction (Transparence)	10
1.5.2 Fiabilité	11
1.5.3 Performance	11
1.5.4 Dimensionnement (Scalability)	11
1.6 Gestion des erreurs réseau	11
1.6.1 Garanties du réseau	11
1.6.2 Responsabilités du système réparti	11
1.7 Protocole Request-Reply-Acknowledge	12
2 Notion de pannes	12
2.1 Couplage	13
2.2 Types de pannes	13
2.2.1 Panne permanente	13
2.2.2 Panne récupérable	13
2.2.3 Panne arbitraire (Byzantine)	14
2.2.4 Autres types de pannes	14
2.3 Détecteur de pannes	14
2.3.1 Détecteur de pannes parfait	14
2.3.2 Heartbeat	14

2.3.3 Détecteur de pannes parfait <u>un jour</u>	16
3 Horloges logiques	17
3.1 Problématique	18
3.2 Protocole existant	18
3.3 Résolution	18
3.4 Propriétés des ordres	18
3.5 Horloge logique	19
3.6 Horloge de Lamport	19
4 Exclusion mutuelle	20
4.1 Problématique	21
4.2 Système centralisé	21
4.3 Solution répartie	22
4.3.1 Version priority queue	22
4.3.2 Version tableau	23
4.4 Propriétés de l'algorithme Lamport	23
5 Mutex par jetons (Ricart & Agrawala)	23
5.1 Amélioration	24
5.2 Pseudo-code	25
5.3 Optimisation d'accès	27
5.4 Propriétés	27
5.5 Cas particulier	28
6 Carvalho & Roucaïrol	28
6.1 Amélioration	29
6.2 Propriétés	30
6.3 Pseudo-code	30
7 Mutex par jeton unique	31
7.1 Introduction	32
7.2 Algorithme par jeton	32
7.3 Critères de réussite	32
7.4 Approche naïve (anneau)	32
7.5 Approche avec un arbre (Raymond)	33
7.5.1 Propriétés	33
7.5.2 Demande du jeton	34
7.5.3 Demande multiple	35
7.5.4 Résumé	36
7.5.5 Pseudo-code	37

1 Introduction aux Systèmes Distribués

1.1 Définition d'un système distribué

Definition

Un **système distribué** est un ensemble de machines autonomes et indépendantes qui apparaissent à l'utilisateur comme un système unique et cohérent.

Les systèmes distribués permettent de répartir le traitement et les données sur plusieurs machines interconnectées, offrant ainsi des avantages en termes de performance, de fiabilité et de scalabilité.

Système Réparti Distribué Décentralisé

Système s'exécutant sur

- un ensemble de processus (process, machine, etc),
- sans mémoire partagée,
- vu par l'utilisateur.rice comme une seule entité.

S'emploie plus quand on parle **des tâches et leur répartition**.

S'emploie plus quand on parle de **l'architecture du système**.

Système distribué dans lequel **il n'existe pas d'autorité centrale** responsable du contrôle du système.

Relativement interchangeables.

Fig. 1. – Capture des slides du cours – Architecture d'un système distribué

1.1.1 Enjeux des systèmes distribués

Logiciel

- Problèmes simples deviennent compliqués
- Tous langages ne sont pas adaptés
- Niveau de transparence sur l'aspect réparti

Fiabilité

- Résilience à
- Délai du réseau
 - Perte de messages
 - Crash de machines

Partage de données

- Synchronisation des données entre machines
- Protection et sécurité des données

Fig. 2. – Capture des slides du cours – Enjeux des systèmes distribués

Les principaux enjeux incluent :

- La coordination entre les différentes machines
- La gestion des pannes et de la tolérance aux fautes
- La cohérence des données réparties
- La performance et la scalabilité

1.2 Parallélisme vs Concurrence

1.2.1 Parallélisme

Definition

Le **parallélisme** se produit lorsque deux tâches sont en cours d'exécution **au même instant** sur différentes unités de traitement.

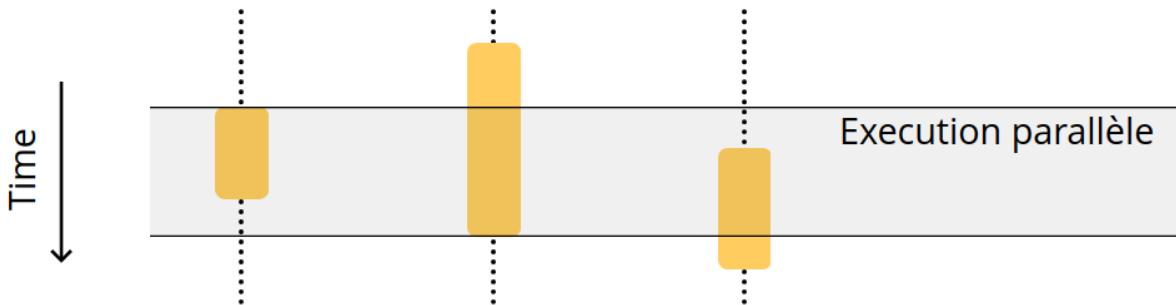


Fig. 3. – Capture des slides du cours – Parallelisme

Warning

Difficulté principale : Coordonner les unités de traitement pour garantir un résultat correct.

1.2.2 Concurrence

Definition

La **concurrence** se produit lorsque deux tâches ont progressé dans un **intervalle de temps commun**, sans nécessairement s'exécuter au même instant.

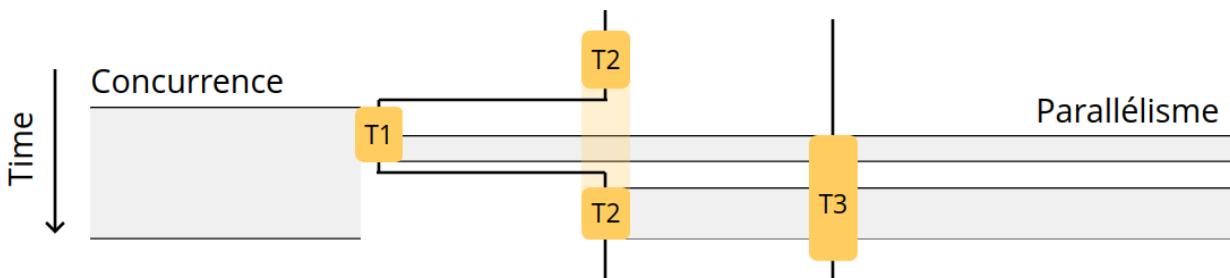


Fig. 4. – Capture des slides du cours – Concurrence

Note

T1, T2 et T3 s'exécutent de manière concurrente, mais pas toujours en parallèle.

1.2.3 Unités de traitement

Les unités de traitement exécutant ces tâches peuvent être :

- **Threads** : Système multi-threaded (ex: CPU multi cœur)
- **Machines** : Système distribué (ex: réseau de PC interconnectés)

1.3 Classification de Flynn

La classification de Flynn catégorise les architectures de machines selon deux axes :

- Flots de données
- Flots d'instructions

		Flot d'Instructions (Contrôle)	
		Single	Multiple
Flot de Données	Single	SISD	MISD
	Multiple	SIMD	Shared memory MIMD Distributed memory

Fig. 5. – Capture des slides du cours – Classification de Flynn

1.3.1 SISD (Single Instruction, Single Data)

Architecture Von Neumann classique avec un seul processeur exécutant une instruction à la fois.

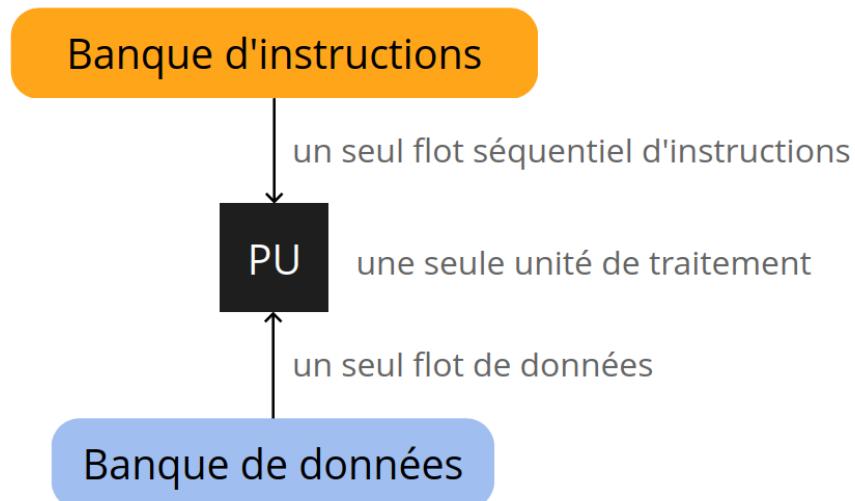
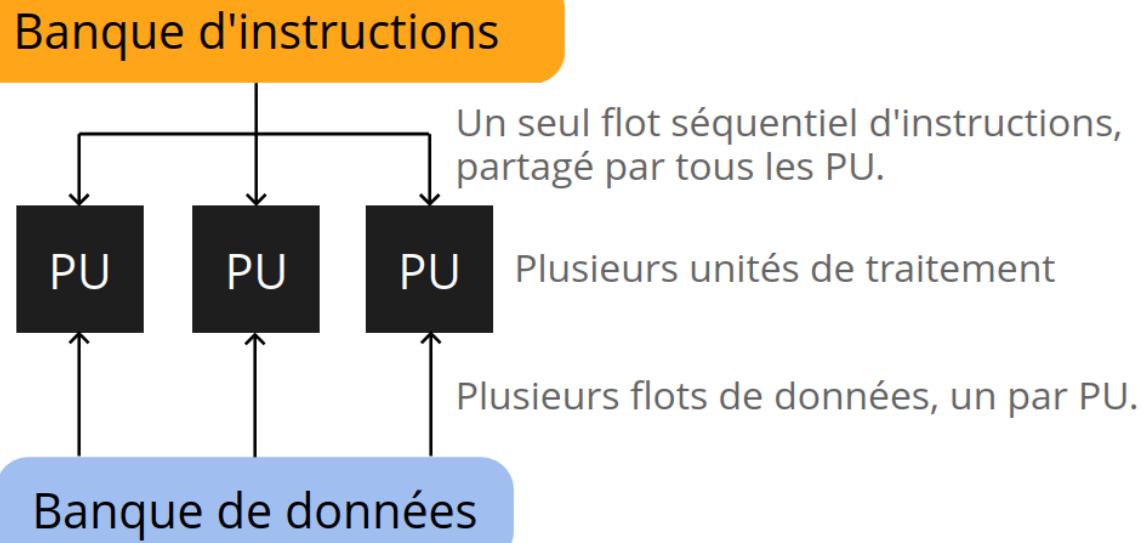


Fig. 6. – Capture des slides du cours – Architecture SISD

1.3.2 SIMD (Single Instruction, Multiple Data)

Une seule instruction s'applique simultanément à plusieurs données (ex: processeurs vectoriels, GPU).

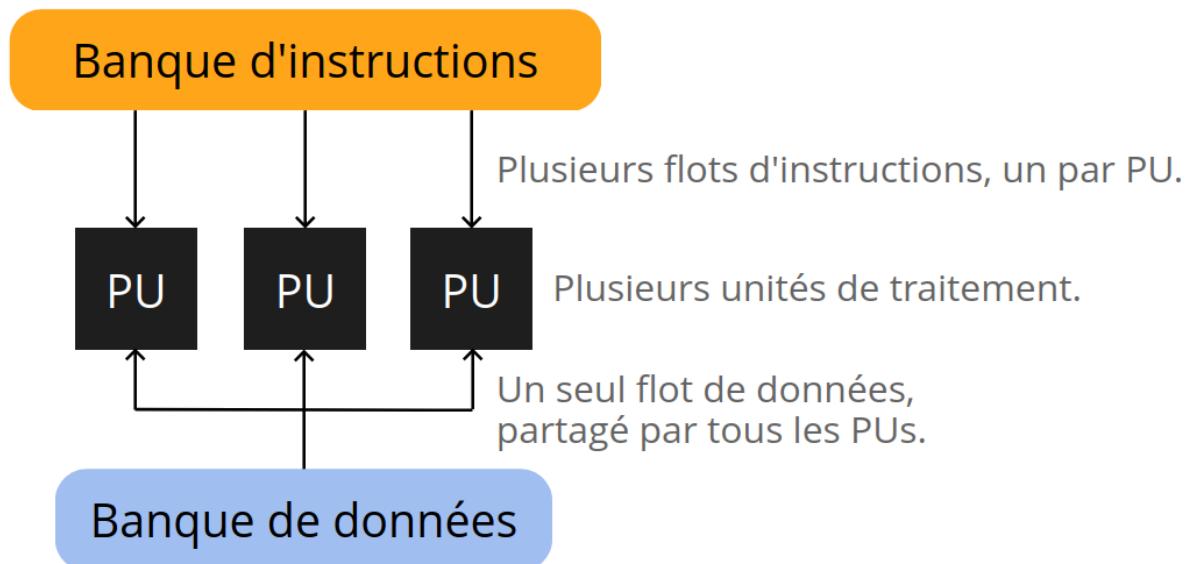


Par exemple pour calcul scientifique (vecteurs et matrices)

Fig. 7. – Capture des slides du cours – Architecture SIMD

1.3.3 MISD (Multiple Instruction, Single Data)

Architecture théorique rarement utilisée en pratique.

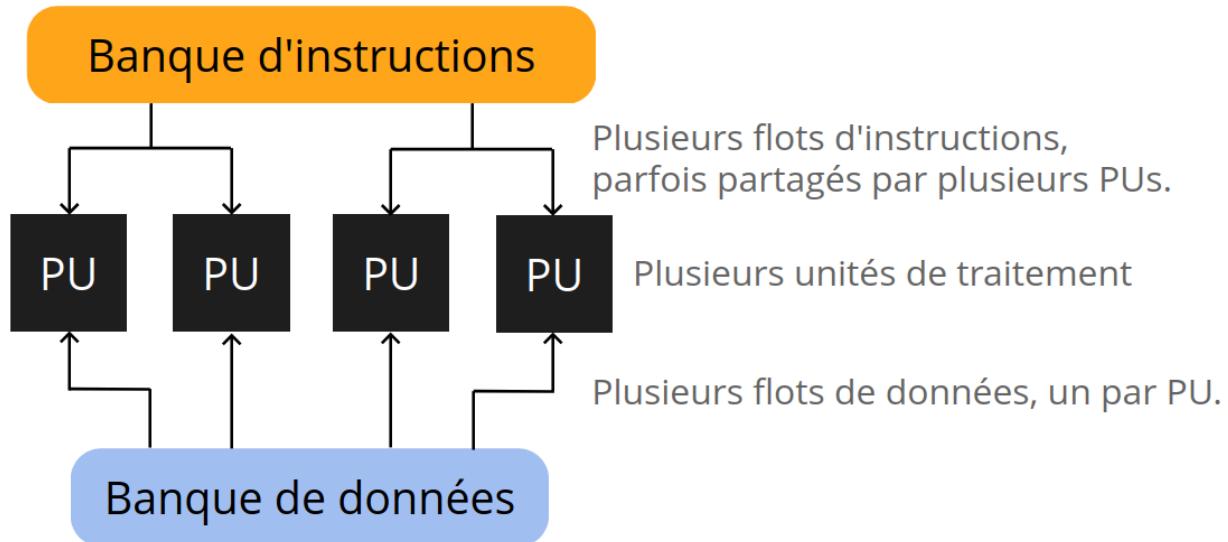


Architecture théorique...

Fig. 8. – Capture des slides du cours – Architecture MISD

1.3.4 MIMD (Multiple Instruction, Multiple Data)

Plusieurs processeurs exécutent différentes instructions sur différentes données. C'est l'architecture des systèmes distribués modernes.



L'exécution peut ici être asynchrone. L'enjeu est la synchronisation des PUs.

Fig. 9. – Capture des slides du cours – Architecture MIMD de base

On distingue deux types de MIMD :

1.3.4.1 MIMD - Shared Memory (Mémoire partagée)

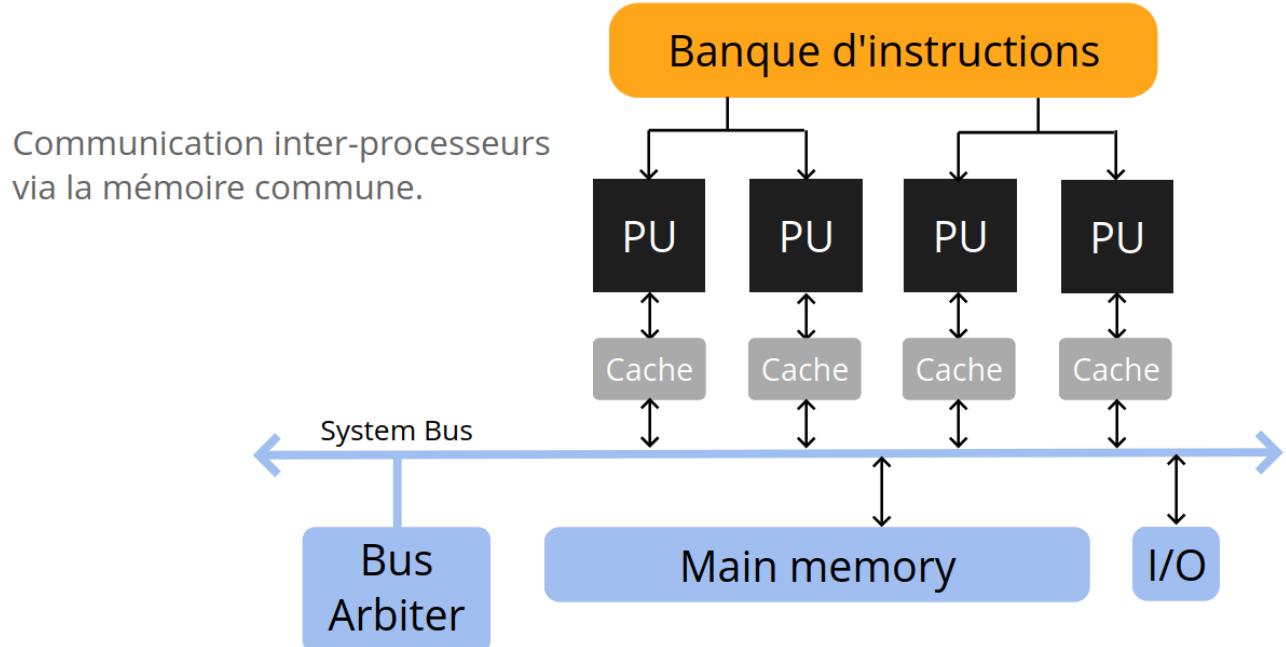
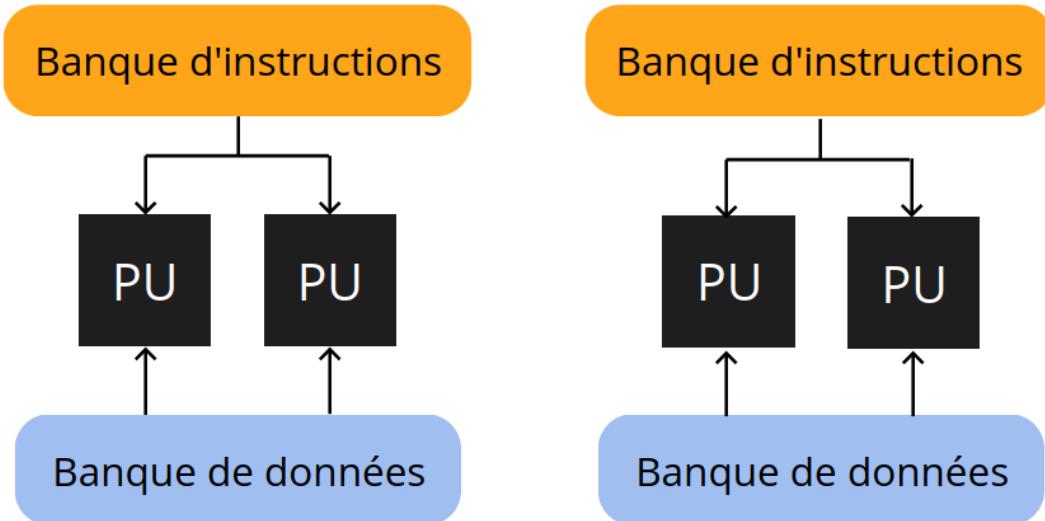


Fig. 10. – Capture des slides du cours – MIMD avec mémoire partagée

1.3.4.2 MIMD - Distributed Memory (Mémoire distribuée)



Communication inter-processeurs via le réseau.

Fig. 11. – Capture des slides du cours – MIMD avec mémoire distribuée

1.4 Couplage : Matériel vs Logiciel

1.4.1 Couplage matériel

Definition

Le **couplage matériel** désigne la quantité et qualité des liens entre les éléments matériels d'un système.

Couplage fort

Beaucoup de liens, rapides.

Couplage faible

Peu de liens, lents.

Shared Memory MIMD

- Débit élevé
- Délai faible

Peut partager beaucoup, rapidement.

Distributed Memory MIMD

- Débit faible
- Délai élevé

Peut partager peu, avec délai.

Fig. 12. – Capture des slides du cours – Couplage matériel

⚠ Warning

En fonction du couplage de l'architecture matérielle ciblée, une même application devra être conçue très différemment.

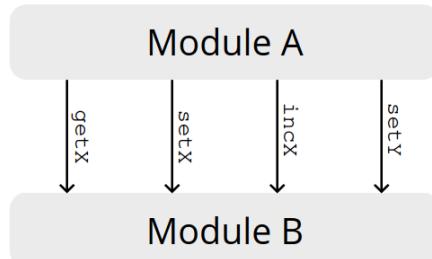
1.4.2 Couplage logiciel

Definition

Le **couplage logiciel** désigne la quantité et qualité des liens entre les modules logiciels d'un système.

Couplage fort

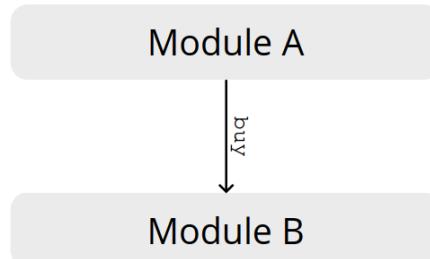
Beaucoup de liens, rapides.



Couplage : 4

Couplage faible

Peu de liens, lents.



Couplage : 1

Fig. 13. – Capture des slides du cours – Couplage logiciel

i Info

Généralement, on vise un couplage logiciel faible :

- Interfaces plus claires
- Risque de bugs moindre

1.4.3 Couplage logiciel et exécution réseau

Dans un contexte d'**exécution réseau** :

- Couplage matériel faible
- Donc coût de communication élevé
- Donc couplage logiciel fort devient coûteux



Fig. 14. – Capture des slides du cours – Couplage et exécution réseau

Avec un couplage logiciel faible, moins d'échanges sont nécessaires entre modules, donc moins de communication sur le réseau. Puisque la communication est coûteuse, c'est un avantage significatif.

1.4.4 Exécution réseau vs Programme réparti

Logiciel réseau simple	Logiciel réseau faiblement couplé	Logiciel réseau réparti
(telnet, wget, ssh)	(NFS, iCloud Drive)	(calcul distribué)
<ul style="list-style-type: none"> • Serveur simple 	<ul style="list-style-type: none"> • Serveur implicitement distribué • Couplage logiciel naturellement faible 	<ul style="list-style-type: none"> • Serveur implicitement distribué • Couplage logiciel à tendance forte <p><i>Conception logicielle combat ce couplage</i></p>

Fig. 15. – Capture des slides du cours – Exécution réseau vs Programme réparti

1.4.5 Couche logicielle de répartition

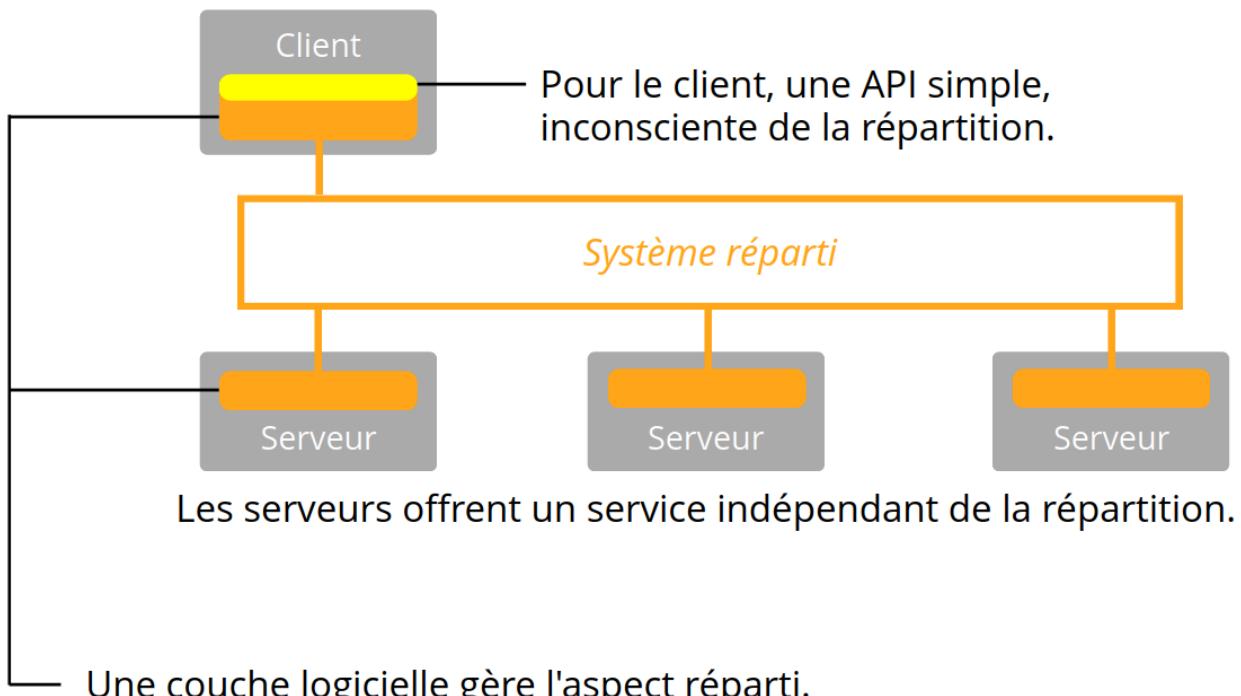


Fig. 16. – Capture des slides du cours – Couche logicielle de répartition

1.4.6 Système réparti

Definition

Un **système réparti** est l'exécution d'une logique nécessitant un **couplage logiciel fort**, sur du matériel limité à un **couplage matériel faible**.

Note

Le challenge est d'optimiser le couplage logiciel effectif pour assurer une performance élevée.

1.5 Propriétés d'un bon système réparti

1.5.1 Abstraction (Transparence)

Emplacement des processus et données

- Pas d'adresses physiques des machines ou des données

Migration des processus et données

- Déplacement de ressource (processus, données) invisible

Duplication des données

- Gestion implicite des copies éventuelles

Cohérence des données

- Gestion implicite de la concurrence

1.5.2 Fiabilité**Disponibilité**

- Résilience aux pannes de machines et de réseau

Cohérence

- État toujours correct (récupération après panne, résistance aux attaques)

1.5.3 Performance**Parallélisme maximal**

- Tirer profit du parallélisme, éviter qu'une machine soit en attente de travail

Communication minimale

- Diminuer le nombre d'échanges de messages

Tradeoff Performance-Fiabilité

- Garantir la fiabilité nécessite des protocoles limitant les performances

1.5.4 Dimensionnement (Scalability)**Extensibilité**

- Ajouter une machine doit être possible et peu coûteux

Complexité algorithmique faible

- Avoir plus de machines ne doit pas rendre le service notablement plus lent

1.6 Gestion des erreurs réseau**1.6.1 Garanties du réseau**

Ce que nous supposerons dans ce cours.

**Pas de perte**

Un message envoyé est reçu.

Pas de duplication

Un message envoyé n'est reçu qu'une fois.

Pas de changement d'ordre

L'ordre de réception est celui d'envoi.

Fig. 17. – Capture des slides du cours – Garanties du réseau

1.6.2 Responsabilités du système réparti**⚠ Warning**

Le système réparti doit :

- Maintenir les garanties du réseau
- **Assurer la résilience aux pannes** (de serveur et du client)
- Implémenter différents **protocoles de fiabilité**

1.7 Protocole Request-Reply-Acknowledge

Le protocole RRA (Request-Reply-Acknowledge) est un protocole de fiabilité permettant de garantir la bonne réception et le traitement des messages dans un système distribué.

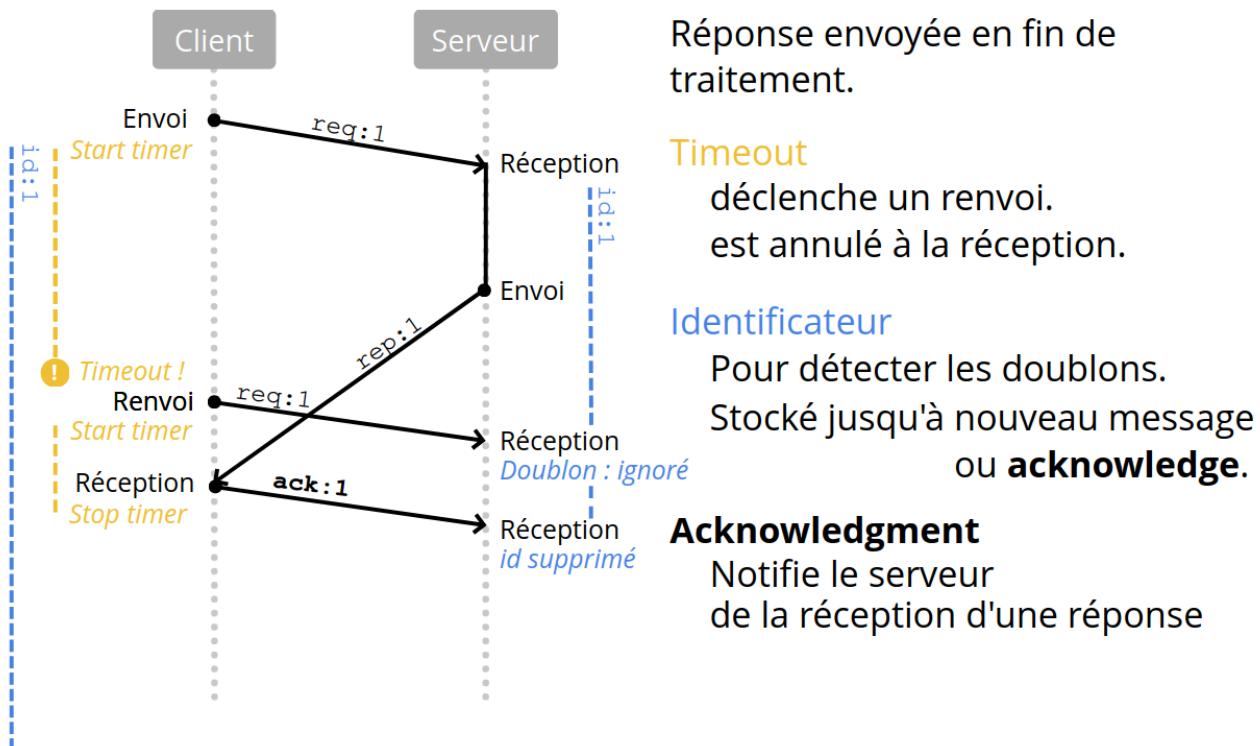


Fig. 18. – Capture des slides du cours – Protocole RRA - Fonctionnement de base

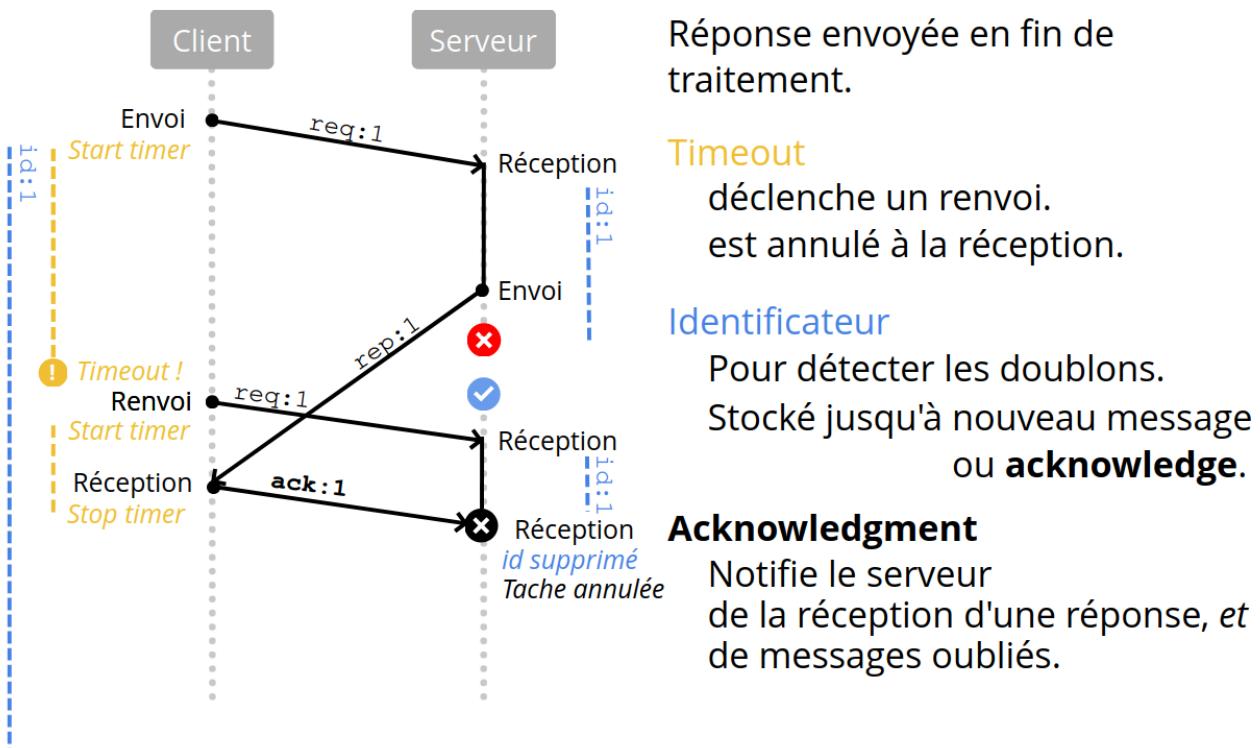


Fig. 19. – Capture des slides du cours – Protocole RRA - Gestion des erreurs

2 Notion de pannes

2.1 Couplage

Dans les systèmes distribués, le couplage fait référence au degré d'interdépendance entre les différents composants ou processus du système. Un couplage faible signifie que les composants sont indépendants les uns des autres, tandis qu'un couplage fort indique une forte dépendance entre eux.

Dans un système distribué, nous souhaiterons plutôt avoir un couplage faible, car cela permet une meilleure tolérance aux pannes et une plus grande flexibilité dans la gestion des ressources. L'objectif est de ne rien partager entre les processus et de profiter au maximum de la communication pour partager les ressources.

2.2 Types de pannes

2.2.1 Panne permanente

Une panne permanente est une défaillance d'un composant du système qui persiste indéfiniment.

Dans notre cas, on dit qu'un processus est correct en terme de panne permanente quand il ne tombera jamais en panne permanente.



Fig. 20. – Capture des slides du cours – Panne permanente

i Info

L'aspect **correct** est une caractéristique théorique.

- Elle servira à analyser les algorithmes.
- Elle n'a aucun sens dans la vraie vie.

2.2.2 Panne récupérable

Une panne récupérable est une défaillance d'un composant du système qui peut être corrigée, permettant au composant de revenir à un état de fonctionnement normal.

Lorsque notre processus revient d'une panne, il se peut qu'il ait perdu des informations, dans ce cas nous parlons d'**amnésie**.



Fig. 21. – Capture des slides du cours – Panne récupérable

On dit qu'un processus est correct en terme de panne récupérable quand il existe un instant T après lequel il ne tombera plus en panne.



Fig. 22. – Capture des slides du cours – Panne récupérable - correct

2.2.3 Panne arbitraire (Byzantine)

Une panne arbitraire, aussi appelée panne byzantine, est une défaillance d'un composant du système qui peut se comporter de manière imprévisible ou malveillante, affectant la fiabilité et la sécurité du système.



Fig. 23. – Capture des slides du cours – Panne byzantine

On dit qu'un processus est correct en terme de panne arbitraire quand il suivra toujours l'algorithme attendu.

Note

Les pannes byzantines sont les plus difficiles et couteuses à gérer dans les systèmes distribués, car elles peuvent impliquer des comportements malveillants ou erratiques.

2.2.4 Autres types de pannes

- **Panne d'omission:** lorsqu'un message devant être envoyé ne l'est pas.
- **Panne d'eavesdropping:** lorsqu'un message peut être lu par une entité extérieure au système.

2.3 DéTECTEUR DE PANNEs

2.3.1 DÉTECTEUR DE PANNEs PARFAIT

Un détecteur de pannes parfait est un mécanisme qui permet d'identifier avec certitude les processus défaillants dans un système distribué.

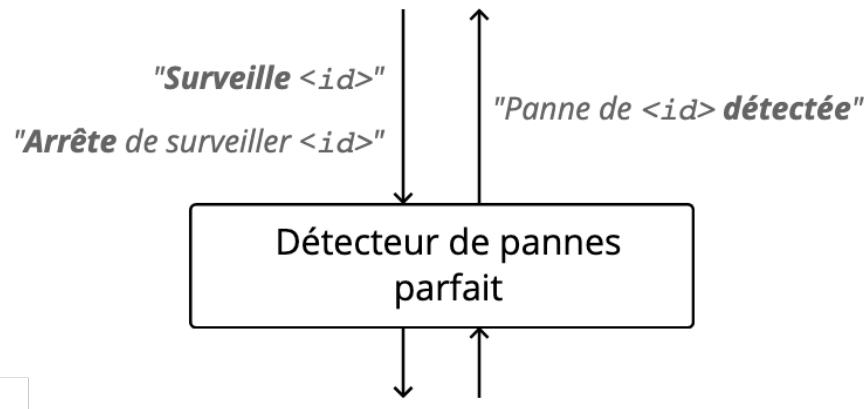


Fig. 24. – Capture des slides du cours – Détecteur de panne parfait

Propriétés:

- **Complétude:** Un jour, tout processus en panne sera détecté par tous les processus corrects.
- **Précision:** Si un processus p est détecté par un quelconque processus, alors p est effectivement en panne.

2.3.2 Heartbeat

Un détecteur de pannes de type **heartbeat** est un mécanisme utilisé dans les systèmes distribués pour surveiller l'état des processus en envoyant périodiquement des signaux (ou « battements de cœur ») entre eux.

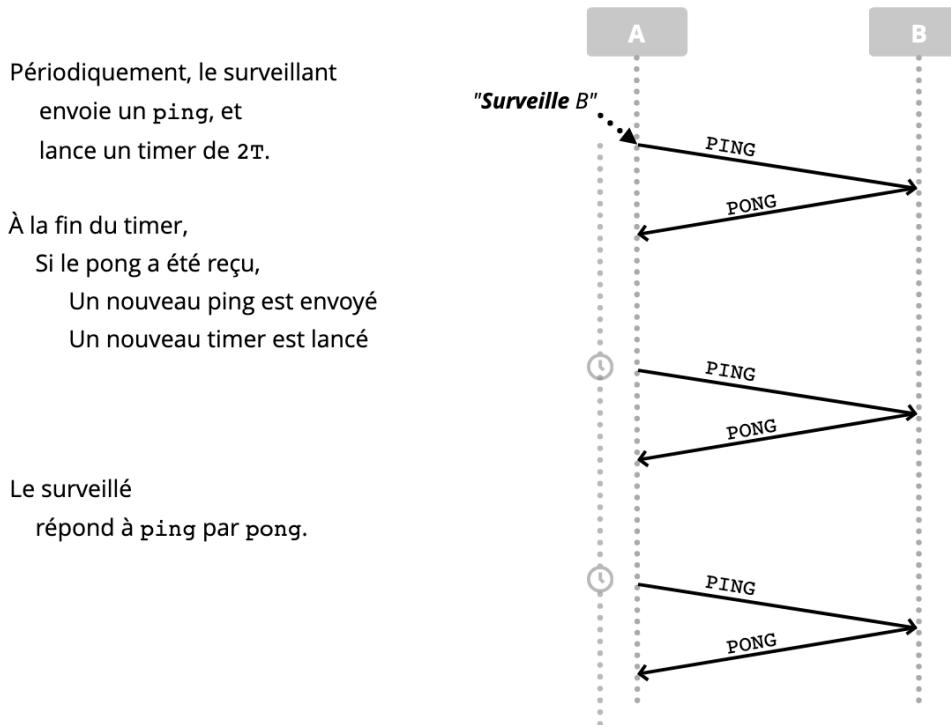


Fig. 25. – Capture des slides du cours – Détecteur de pannes Heartbeat

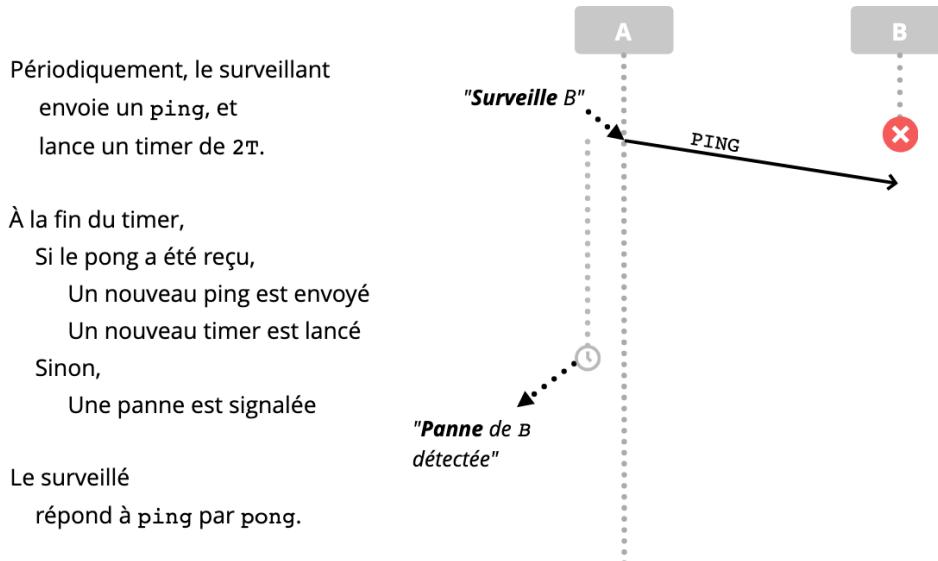


Fig. 26. – Capture des slides du cours – Détecteur de pannes Heartbeat - exemple

Dans ces exemples, nous voyons que notre système répond aux deux propriétés d'un détecteur de pannes parfait.

⚠ Warning

Cependant la supposition d'un système synchrone est irréaliste. Les vrais réseaux ne nous donnent pas de garantie sur la durée de transit du message. C'est pourquoi ce genre de détecteur de type Heartbeat donnera potentiellement des faux positifs, **la notion de τ unité de temps n'existe pas.**

2.3.3 DéTECTEUR DE PANNE PARFAIT UN JOUR

Un détecteur de pannes parfait **un jour** est un mécanisme qui garantit que, après un certain temps, tous les processus corrects auront détecté tous les processus en panne, tout en maintenant la précision.

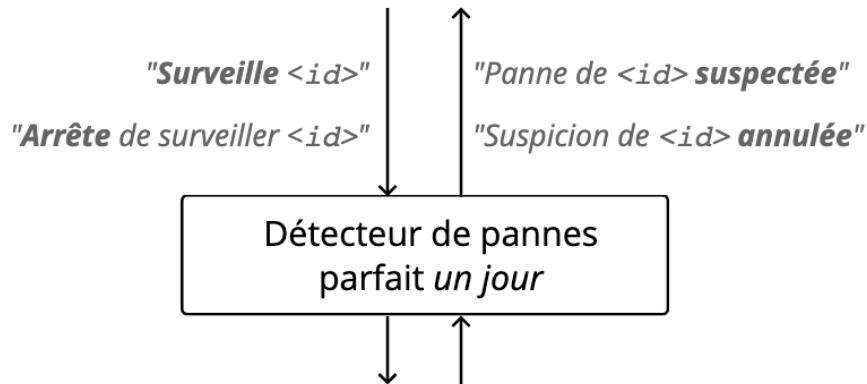


Fig. 27. – Capture des slides du cours – DéTECTEUR DE PANNE PARFAIT UN JOUR

Propriétés:

- **Complétude:** Un jour, tout processus en panne sera détecté par tous les processus corrects.
- **Précision un jour:** **Un jour**, aucun processus correct ne sera suspecté par processus correct.

Timeout dynamique:

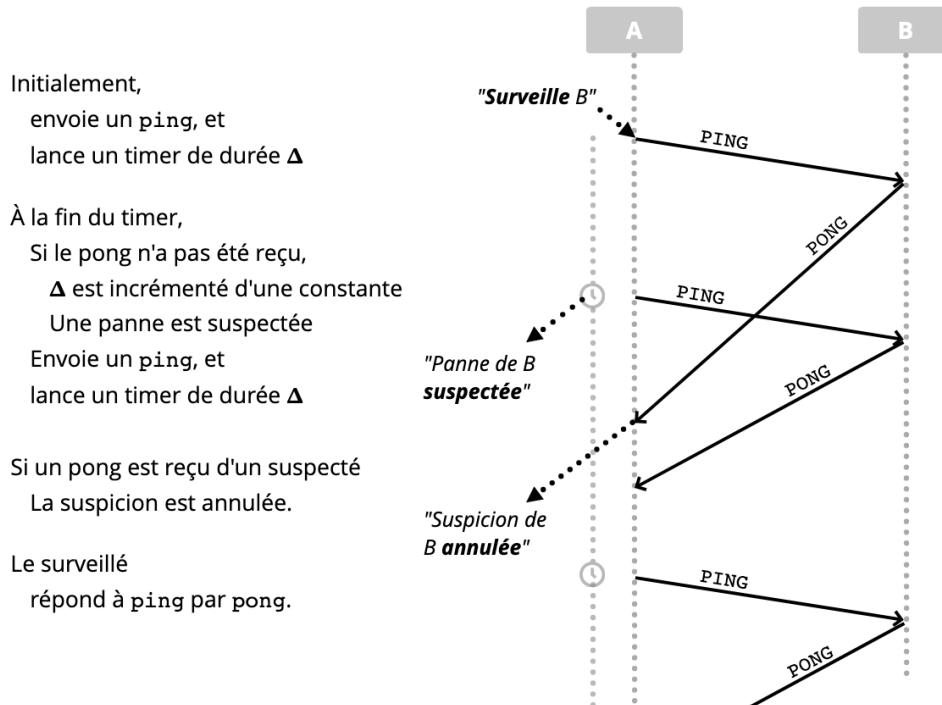


Fig. 28. – Capture des slides du cours – Timeout dynamique

Grâce à cette nouvelle définition, nous pouvons donc affirmer les informations suivantes:

- Complétude: Un jour, tout processus en panne sera détecté par tous les processus corrects.
 - ▶ *Oui, un processus en panne permanente ne répondra plus aux pings, et après Δ , cette panne sera découverte.*
- Précision un jour: Un jour, aucun processus correct ne sera suspecté par processus correct.

- Oui, un processus correct (en terme de panne permanente comme récupérable) répondra aux pings en un temps fini ; quand Δ sera assez grand, plus de panne ne sera suspectée : tout processus sera soit « en panne » et suspecté pour toujours, ou « correct » et plus jamais suspecté.

 **Note**

Pour résoudre ce problème, était-il nécessaire d'avoir un timeout dynamique?

- Oui, sinon on risquerait de constamment suspecter un processus qui est simplement lent mais bien correct.

3 Horloges logiques

3.1 Problématique

Lorsqu'on travaille dans un système distribué, il est crucial de pouvoir ordonner les événements qui s'y produisent. Cependant, en l'absence d'une horloge globale, il devient difficile de déterminer l'ordre exact des événements.

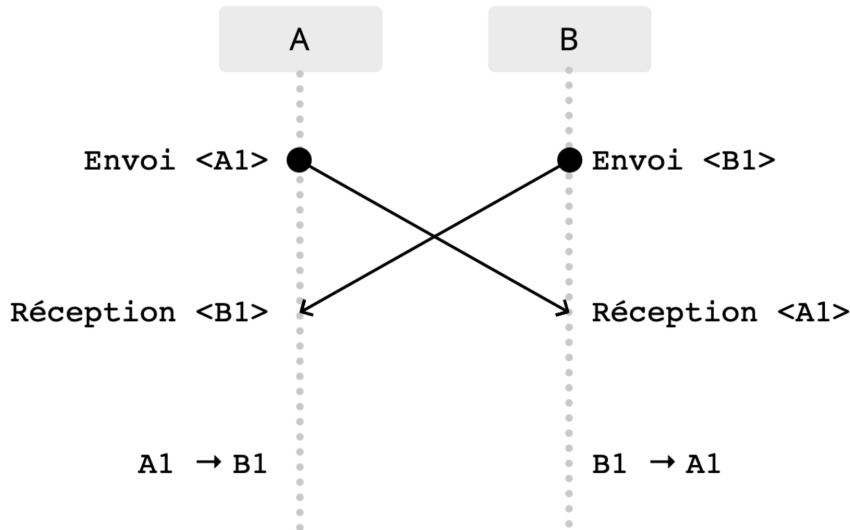


Fig. 29. – Capture des slides du cours – Problème d'ordonnancement

⚠ Warning

Sur cette image, nous voyons la problématique concernant l'ordonnancement des événements dans un système distribué. Chaque système prétend que son événement est le premier.

3.2 Protocole existant

- **NTP:** Network Time Protocol (NTP) ms precision
- **PTP:** Precision Time Protocol (PTP) μs precision

3.3 Résolution

On ne cherche pas l'heure exacte mais plutôt un ordre d'événement.

- Si E1 et E2 sont sur la même machine et E1 arrive avant E2.
- Si E1 est l'émission d'un message et E2 sa réception.

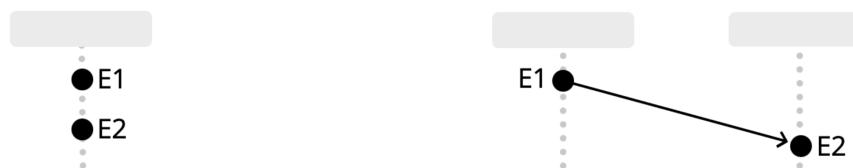


Fig. 30. – Capture des slides du cours – Ordre d'événements

3.4 Propriétés des ordres

Nous pouvons définir trois propriétés pour un ordre d'événements:

- **Transitivité:** Si un événement A précède un événement B, et que B précède C, alors A précède C.
- **Anti-réflexif:** Aucun événement ne peut précéder lui-même.
- **Antisymétrique:** Si un événement A précède B, alors B ne peut pas précéder A.

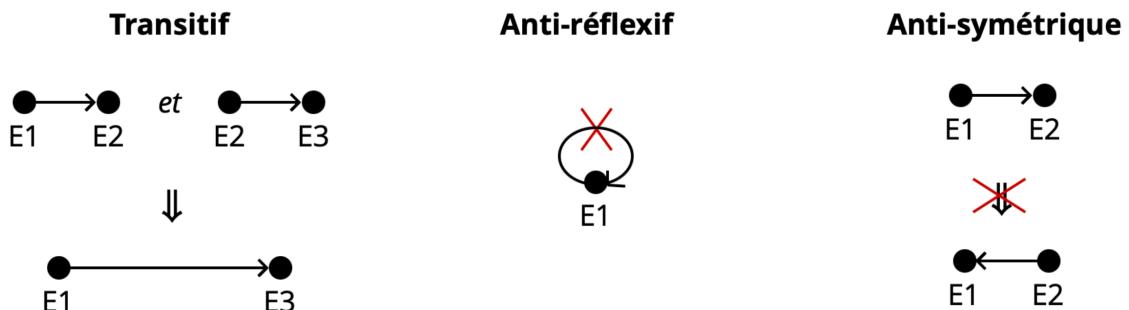


Fig. 31. – Capture des slides du cours – Propriétés des ordres

3.5 Horloge logique

On cherche à définir une fonction H qui associe un entier à chaque événement, de manière à respecter les propriétés d'ordre définies précédemment.

Une idée serait d'utiliser une horloge locale pour chaque processus, qui s'incrémente à chaque événement. Cependant, cela ne garantit pas que les événements soient ordonnés correctement à travers les différents processus. Le risque de décalage entre les horloges locales est trop grand et nous pourrons nous retrouver avec un événement 6 qui arrive en réalité avant un événement 5.

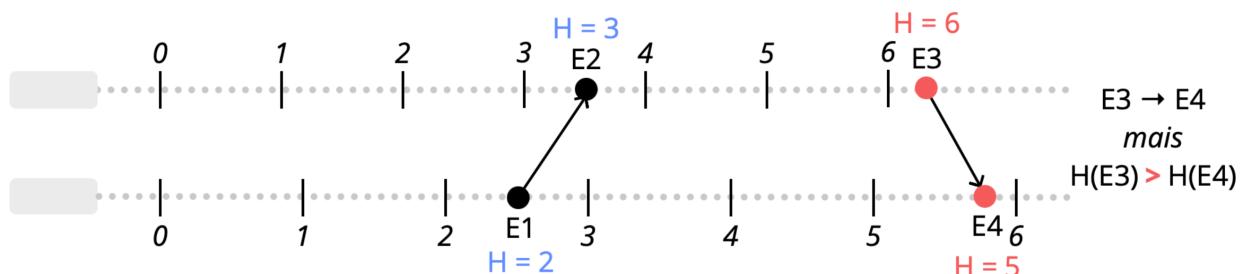


Fig. 32. – Capture des slides du cours – Risque d'utilisation d'horloges locales

3.6 Horloge de Lamport

Pour résoudre ce problème, Lamport propose un algorithme d'horloge logique qui utilise des horloges locales, mais ajoute des règles pour garantir l'ordre des événements à travers les processus.

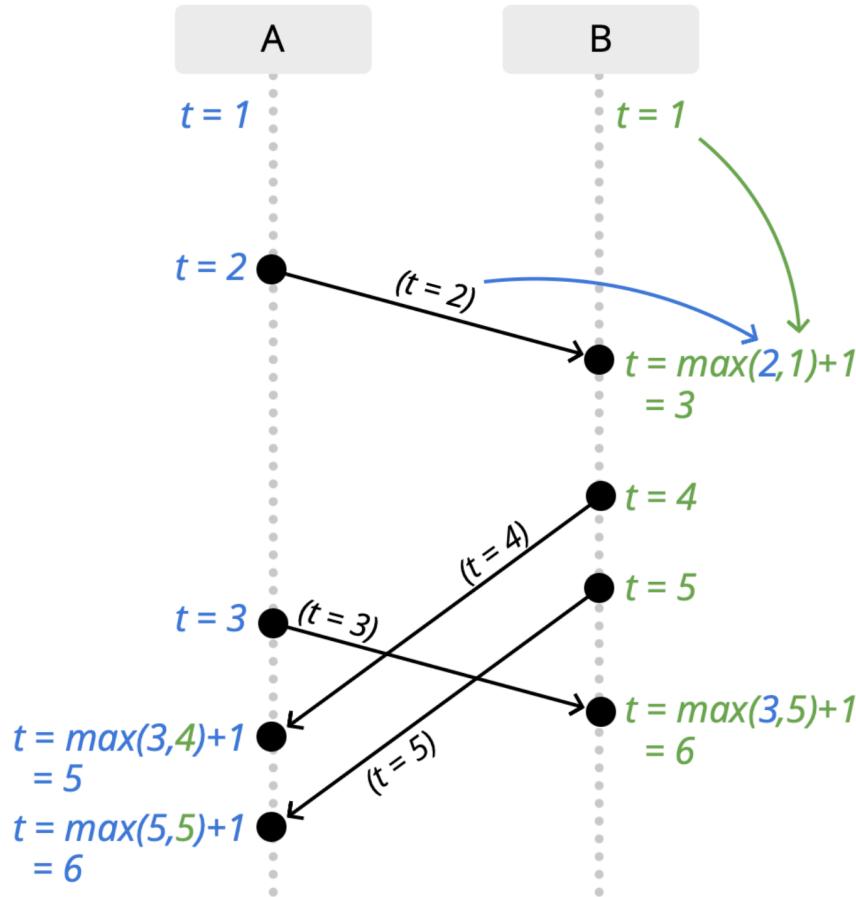


Fig. 33. – Capture des slides du cours – Algorithme de Lamport

- Chaque site maintient un numéro
- À chaque événement local, le site incrémente son numéro
- Le timestamp est attaché au message
- À la réception d'un message, le site met à jour son numéro avec le maximum entre son numéro et le timestamp reçu, puis incrémente ce numéro de 1

⚠ Warning

Cet algorithme n'est pas déterministe, si deux événements sont concurrents, l'ordre peut varier selon les exécutions. **Solution:** priorité à la machine avec l'ID le plus petit

⚠ Warning

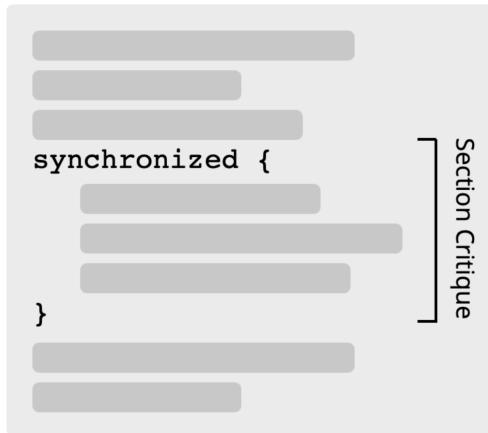
- Les timestamps de Lamport n'ordonnancent pas les messages immédiatement.
- Ils assurent seulement qu'un ordre strict sera obtenu un jour.

4 Exclusion mutuelle

4.1 Problématique

Dans un système distribué, il est crucial de gérer l'accès concurrent aux ressources partagées. L'exclusion mutuelle garantit qu'une ressource ne peut être utilisée que par un seul processus à la fois, évitant ainsi les conflits et les incohérences.

Exemple inspiré de la syntaxe Java



Synchronisation nécessaire

- par locks (**shared memory**), ou
- par messages (**systèmes répartis**).

Exemple

Copie synchronisée d'un même compte en banque. Certaines opérations doivent être séquentielles.

Fig. 34. – Capture des slides du cours – Exclusion mutuelle

4.2 Système centralisé

Un serveur centralisé gère les demandes d'accès aux ressources. Chaque processus doit demander la permission au serveur avant d'accéder à la ressource.

La tête de la file est toujours celle actuellement en SC.

On request

Push demandeur dans la file

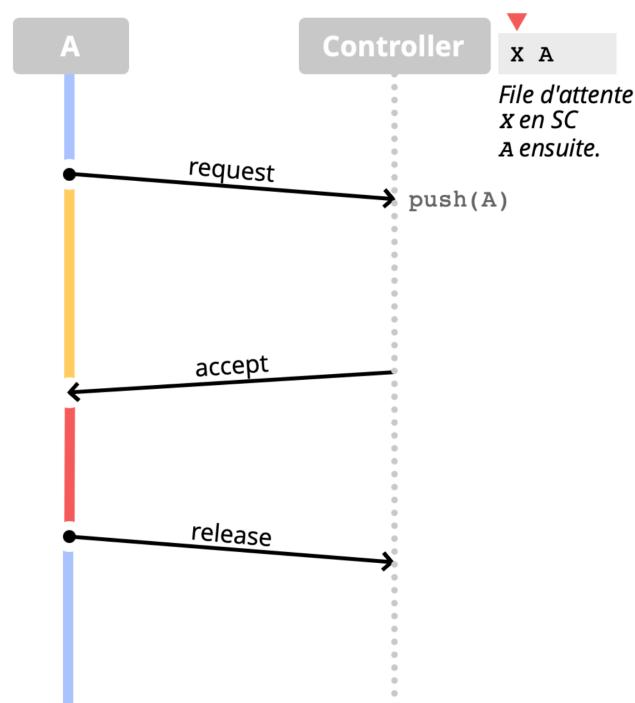


Fig. 35. – Capture des slides du cours – Système centralisé

La tête de la file est toujours celle actuellement en SC.

On request

Push demandeur dans la file

On release

Pop tête de file

Envoi de accept à nouvelle tête

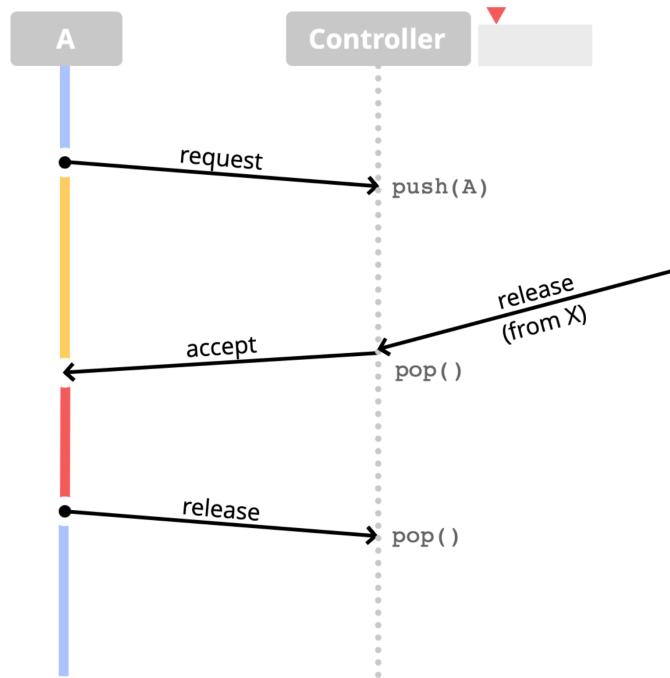


Fig. 36. – Capture des slides du cours – Système centralisé - exemple

4.3 Solution répartie

Chaque processus maintient une file d'attente des demandes d'accès aux ressources, ordonnée par les horloges logiques. Lorsqu'un processus souhaite accéder à une ressource, il envoie une demande à tous les autres processus et attend les réponses.

4.3.1 Version priority queue

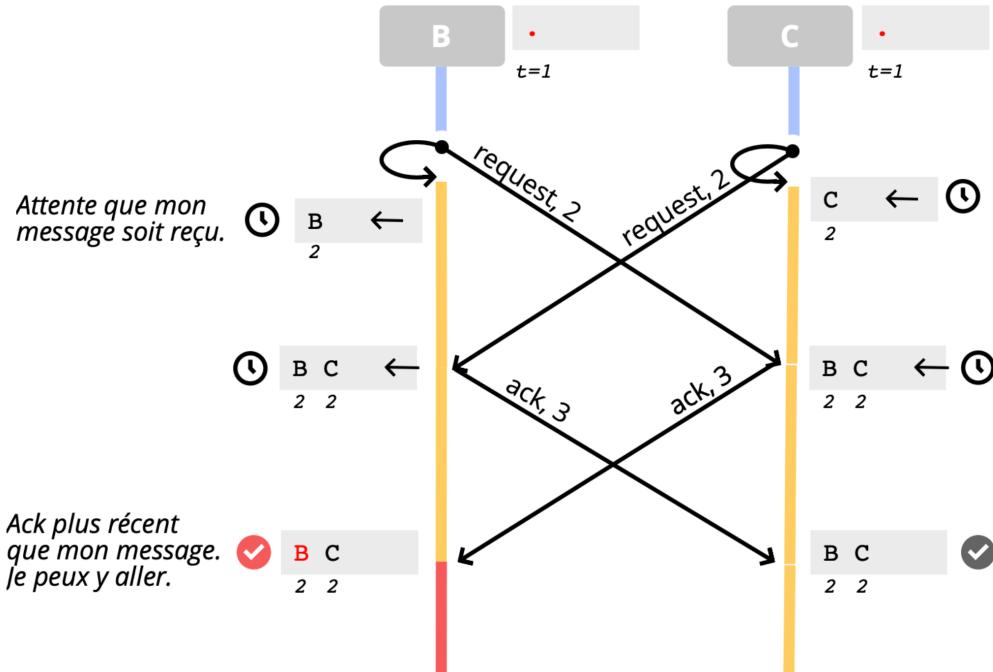


Fig. 37. – Capture des slides du cours – Solution répartie - priority queue

On request:

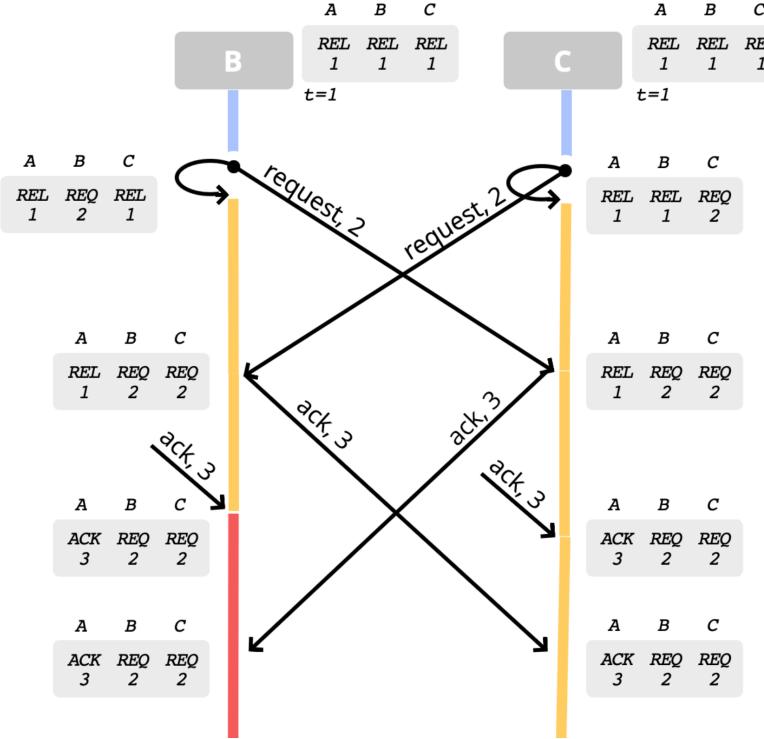
- Push demandeur dans la file d'attente
- Réordonner la file d'attente par heure de Lamport

- Envoie un `ack` avec nouveau timestamp

On release:

- Pop tête de file
- Entrer en section critique
 - ▶ Si je suis la nouvelle tête et que j'ai reçus tous les `acks`

4.3.2 Version tableau



Amélioration :

File remplacée par tableau.

Pour chaque processus,
initialement {REL, 1}, puis

- le dernier message reçu
- le timestamp associé

sauf si ACK remplacerait REQ.

Un processus entre en SC si

- son dernier message est REQ,
- et son heure logique est **strictement la plus petite**.

Fig. 38. – Capture des slides du cours – Solution répartie - version tableau

Chaque processus maintient un tableau des derniers messages reçus ainsi qu'un timestamp pour chaque message, par exemple `(REQ, id)`.

4.4 Propriétés de l'algorithme Lamport

- **Correctness:** Un seul processus peut être en section critique à la fois.
- **Progrès:** Toute demande d'entrée en SC sera autorisée un jour.
- **Complexité**
 - ▶ Communication par SC par processus : $3n(n - 1)$
 - ▶ Calcul par événement :
 - $O(n)$ par réception de message
 - $O(1)$ pour le reste.

5 Mutex par jetons (Ricart & Agrawala)

On passe d'une approche à 3 messages:

- `REQ` : demande d'accès à la section critique
- `ACK` : autorisation d'accès à la section critique
- `REL` : libération de la section critique

i Info

Ce que nous pouvons tirer comme conclusion est que nous sommes assez indirect dans notre approche. Ce que nous voulons est d'entrer en **section critique**.

Jusqu'à présent, nous avions 3 messages, hors, ce que l'on souhaite c'est demander d'entrer en section critique, une fois que l'on a **toutes** les autorisations, on entre en section critique, puis on libère la section critique. On en comprend donc que le message `REQ` est nécessaire, cependant on peut simplifier `ACK` et `REL` en un seul message `OK` qui autorise l'entrée en section critique.

💡 Hint

En cherchant à limiter le nombre de messages, on peut penser à une approche par jetons.

5.1 Amélioration

App veut entrer en SC

J'envoie un `REQ` à tout le monde

Réception d'un `REQ`

Si je ne suis pas en SC

- Si je ne veux pas entrer en SC
 - Je réponds avec `OK`
- Si je veux entrer en SC
 - Si je n'ai pas la priorité, je réponds avec `OK`.
 - Si j'ai la priorité, j'attends d'avoir fini, puis je réponds avec `OK`.

Si je suis en SC

- J'attends d'avoir fini, puis je réponds avec `OK`.

App sort de SC

Je réponds par `OK` à tous les `REQ` en attente.

Réception d'un `OK`

Je le comptabilise.

- Si j'ai le `OK` de tout le monde, alors je peux entrer en SC.

Fig. 39. – Capture des slides du cours – Version améliorée

Grâce à cette nouvelle approche, nous pouvons désormais $2(n - 1)$ messages par processus pour entrer en section critique.

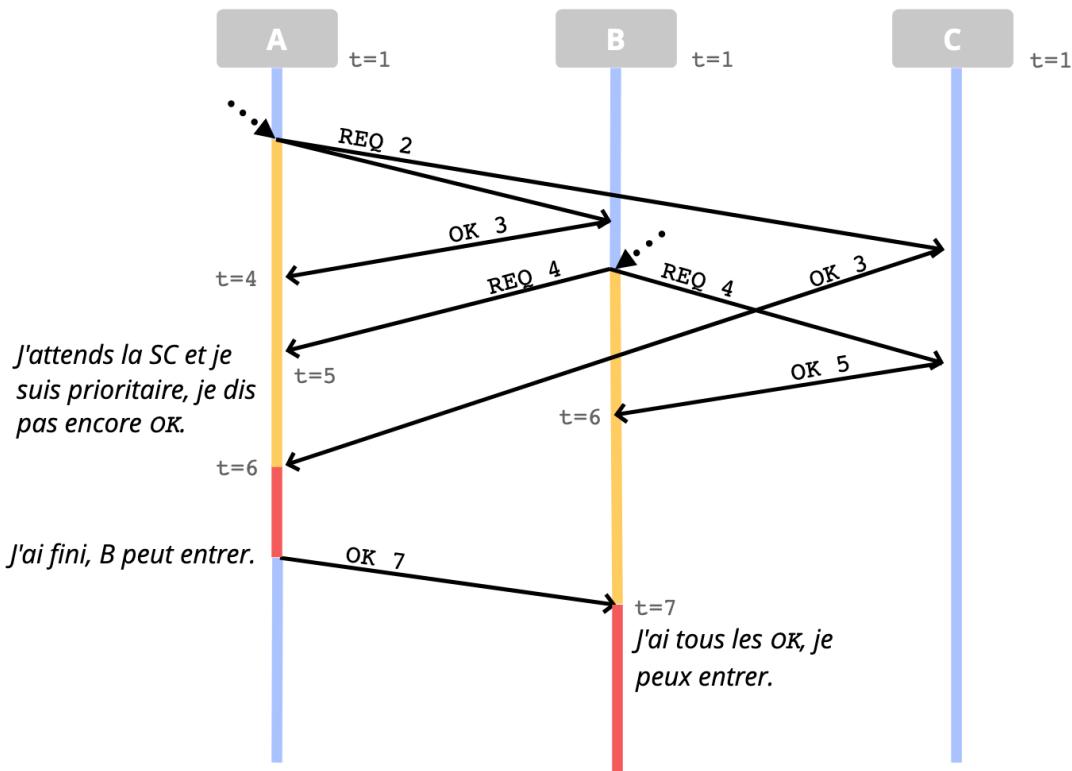


Fig. 40. – Capture des slides du cours – Echanges de messages

5.2 Pseudo-code

n	<i>entier, constant</i>	nombre de processus
self	<i>entier, entre 0 et n-1</i>	mon numéro de processus
ts	<i>entier, init 0</i>	mon timestamp Lamport actuel
hasRequested	<i>booléen, init false</i>	ssi j'ai fait une demande de SC
selfRequestTs	<i>entier</i>	timestamp de cette demande
missingOKs	<i>entier</i>	nombre de OKs que j'attends encore
waitingPs	<i>ensemble d'entiers</i>	numéros des processus qui attendent mon OK.

Fig. 41. – Capture des slides du cours – Variables du pseudo-code

Écouter infiniment les événements suivants:

- Demande de SC de la couche applicative
- Sortie de SC dans la couche applicative
- Passage de `missingOKs` à 0
- Réception de `{REQ, tsi}` du processus i
- Réception de `{OK, tsi}` du processus i

Fig. 42. – Capture des slides du cours – Initialisation du pseudo-code

Traitement : Demande de SC de la couche applicative.

```
ts += 1
hasRequested ← true
selfRequestTs ← ts
missingOKs ← n-1
Envoi de {REQ, ts} à tous les autres processus.
```

Traitement : Passage de `missingOKs` à 0.

Autorisation de la couche application d'entrer en SC.

Traitement : Sortie de SC par la couche applicative.

```
ts += 1
hasRequested ← false
Envoi de {OK, ts} à tous les processus de waitingPs
Réinitialisation de waitingPs
```

Fig. 43. – Capture des slides du cours – Fonctionnement du pseudo-code

Traitement : Réception {REQ, tsi} du processus i.

```

 $ts \leftarrow \max(tsi, ts) + 1$ 
Si hasRequested et
  ( $selfRequestTs < tsi$  ou
  ( $selfRequestTs == tsi$  et  $self < i$ ))
    Ajout de i dans waitingPs
Sinon
  Envoi de {OK, ts} à i
  
```

Traitement : Réception {OK, tsi} du processus i.

```

 $ts \leftarrow \max(tsi, ts) + 1$ 
waitingOKs -= 1
  
```

Fig. 44. – Capture des slides du cours – Fonctionnement du pseudo-code - suite

5.3 Optimisation d'accès

Un autre point est que dans cette approche, si on récupère l'accès pour entrer en section critique, je peux en déduire que j'y ai accès tant que personne ne redemande l'accès.

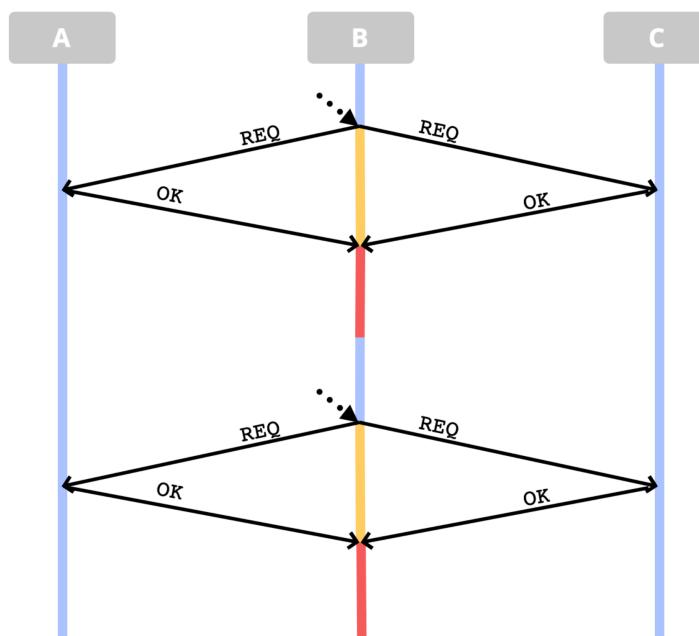


Fig. 45. – Capture des slides du cours –

On peut voir cela un peu comme un jeton que l'on possède. Tant que l'on possède le jeton, on peut entrer en section critique.

5.4 Propriétés

- **Correctness:** Jamais plus d'un processus dans la section critique.
- **Progress:** Toute demande d'entrée en section critique finit par être satisfaite.
- **Complexité de message:** $2(n - 1)$ messages par entrée en section critique.

5.5 Cas particulier

Dans le cas où un processus vient de sortir de SC et qu'il souhaite y rentrer à nouveau, il doit à nouveau envoyer des demandes à tous les autres processus, alors qu'en optimisant il pourrait directement s'y rendre à nouveau.

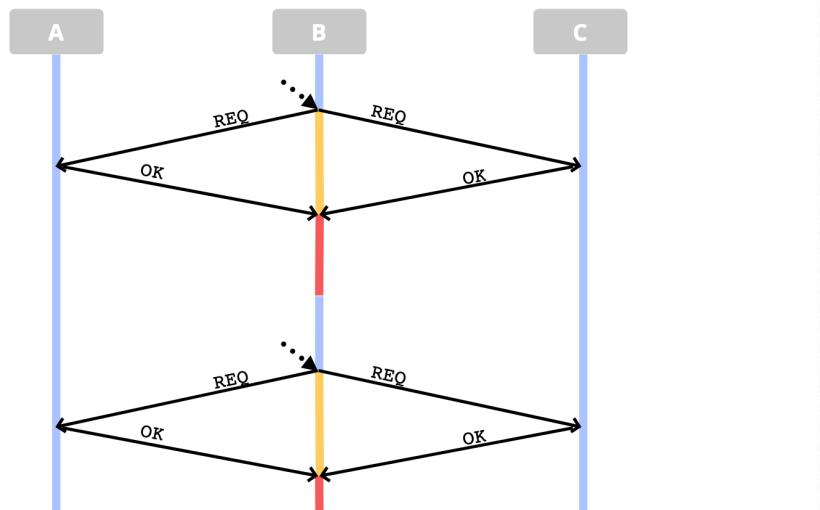


Fig. 46. – Capture des slides du cours –

6 Carvalho & Roucairol

6.1 Amélioration

On pourrait penser à modifier l'algorithme de Ricart & Agrawala en se disant: **Tant qu'on m'a pas redemandé l'accès, je peux y retourner directement sans redemander l'accès.**

Dès ce moment, on peut imaginer que chaque processus possède un jeton. Lorsqu'un processus souhaite entrer en section critique, il doit posséder le jeton. S'il ne le possède pas, il doit le demander au processus qui le possède.

On peut visualiser cet algorithme comme un graph ou chaque processus est un noeud, et entre chaques noeuds, il y a une arête qui représente le jeton.

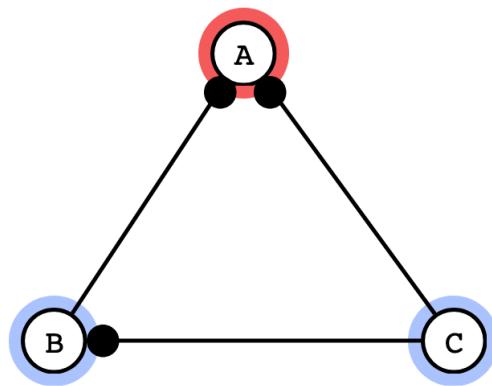


Fig. 47. – Capture des slides du cours – Visualisation graphe

Lorsque qu'un processus souhaite entrer en section critique, il envoie une demande aux noeuds voisins dont il ne possède pas déjà le jeton.

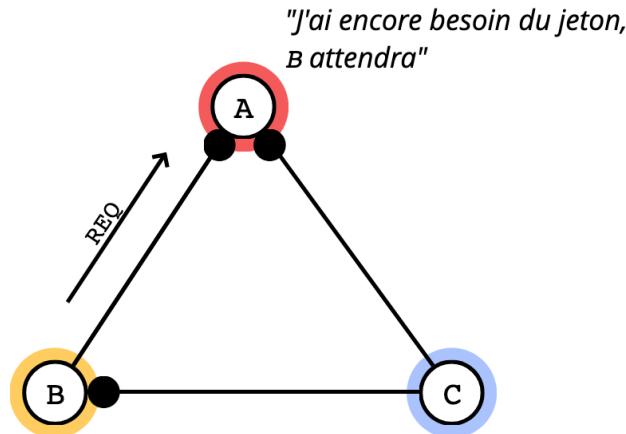


Fig. 48. – Capture des slides du cours – Echanges de messages

Une fois que le processus A a terminé sa section critique, il envoie le jeton au processus B qui lui a fait la demande. Une fois que B a terminé, si personne n'a demandé les jetons, il les garde.

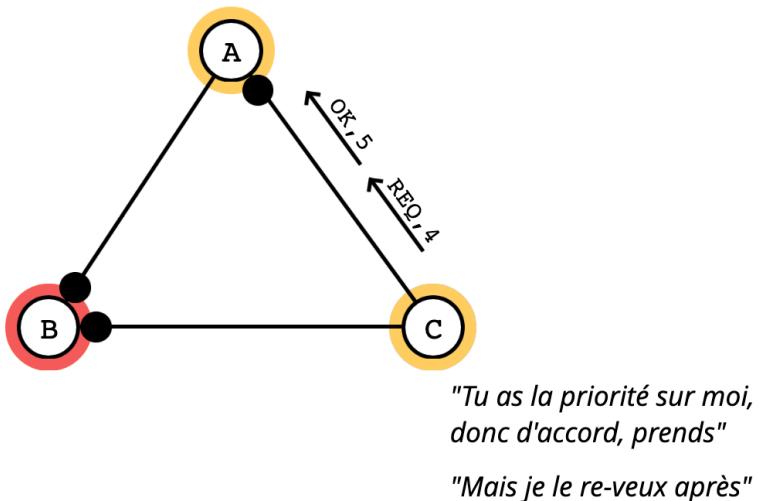


Fig. 49. – Capture des slides du cours – Libération du jeton

Dans le cas où le processus `A` demande le jeton à `C` mais que `C` demande immédiatement après avoir donné son jeton le jeton à `A` pour qu'il lui soit retourné à la fin de la section critique de `A`.

6.2 Propriétés

- Correctness:** Jamais plus d'un processus dans la section critique.
- Progress:** Toute demande d'entrée en section critique finit par être satisfaite.
- Complexité de communication:** Entre 0 et $2(n - 1)$ messages par entrée en section critique.

6.3 Pseudo-code

<code>n</code>	<i>entier, constant</i>	nombre de processus
<code>self</code>	<i>entier, entre 0 et n-1</i>	mon numéro de processus
<code>ts</code>	<i>entier, init 0</i>	mon timestamp Lamport actuel
<code>hasRequested</code>	<i>booléen, init false</i>	ssi j'ai fait une demande de SC
<code>selfRequestTs</code>	<i>entier</i>	timestamp de cette demande
<code>requesters</code>	<i>ensemble d'entiers</i>	numéros des processus qui attendent mon jeton.
<code>jetons</code>	<i>ensemble d'entiers</i>	numéros des processus dont j'ai le jeton. Initialement arbitraire.

Fig. 50. – Capture des slides du cours – Variables du pseudo-code

Écouter infiniment les événements suivants:

- Demande de SC de la couche applicative
- Sortie de SC dans la couche applicative
- Réception de {REQ, tsi} du processus i
- Réception de {OK, tsi} du processus i

Fig. 51. – Capture des slides du cours – Initialisation du pseudo-code

Traitement : Demande de SC de la couche applicative.

```

ts += 1
hasRequested ← true
selfRequestTs ← ts
Pour chaque processus i qui n'est pas dans jetons :
    Envoi de {REQ, ts}
Si jetons a une taille n-1, entrer en SC
```

Traitement : Sortie de SC par la couche applicative.

```

ts += 1
hasRequested ← false
Pour tout i dans requesters :
    Envoi de {OK, ts}
    jetons.remove(i)
Réinitialisation de requesters
```

Fig. 52. – Capture des slides du cours – Fonctionnement du pseudo-code

Traitement : Réception {REQ, tsi} du processus i.

```

ts ← max(tsi, ts) + 1
Si hasRequested et
    (selfRequestTs < tsi ou
    (SelfRequestTs == tsi et self < i))
        Ajout de i dans requesters
Sinon
    Envoi de {OK, ts} à i
    jetons.remove(i)
Si hasRequested
    Envoi de {REQ, selfRequestTs} à i
```

Fig. 53. – Capture des slides du cours – Fonctionnement du pseudo-code - suite

Traitement : Réception {OK, tsi} du processus i.

```

ts ← max(tsi, ts) + 1
jetons.add(i)
Si jetons a une taille n-1 et hasRequested :
    Entrer en SC
```

Fig. 54. – Capture des slides du cours – Fonctionnement du pseudo-code - fin

7 Mutex par jeton unique

7.1 Introduction

Jusqu'à maintenant, nous avons vu des algorithmes de synchronisation basés sur des horloges logiques et des demandes de permission. Cependant, une autre approche intéressante pour gérer l'accès à une section critique dans un système distribué est l'utilisation de jetons uniques.

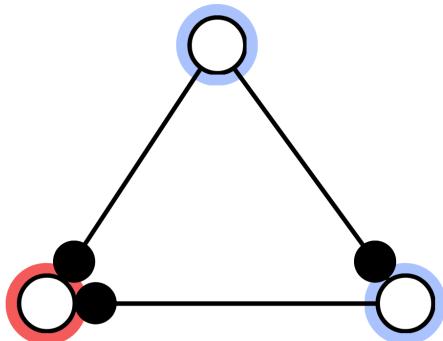
Petit rappel:

- **Lamport:** horloge logique pour ordonner les événements
- **Ricart & Agrawala:** demande de permission à tous les processus avant d'entrer en section critique
- **Carvalho et Roucairol:** demande de permission à un sous-ensemble de processus (passage de témoin entre 2 processus)

7.2 Algorithme par jeton

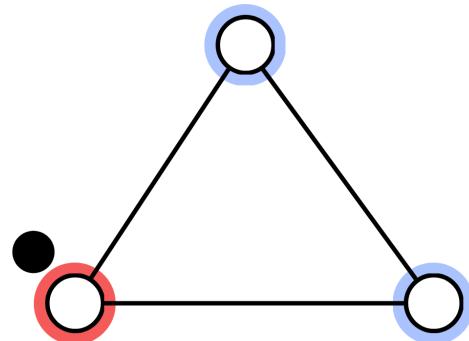
Jusqu'à maintenant les algorithmes que nous avons vu, nécessitaient plusieurs jetons pour fonctionner.

Algorithme par jetons
Un jeton par paire de voisins.



"Il me faut le jeton de tout le monde pour avoir le droit d'être en SC."

Algorithme par jeton X
Un jeton, tout court.



"Il me faut **le jeton** pour avoir le droit d'être en SC."

Fig. 55. – Capture des slides du cours – Différents multi-jetons à jeton unique

7.3 Critères de réussite

Pour résoudre ce problème, en ayant qu'un seul jeton, nous cherchons à respecter 3 critères:

- Unicité: assurer que le jeton ne sera pas dupliqué
- Transmission: gérer le transit du jeton entre les processus
- Progrès: assurer que tout demandeur finira par obtenir le jeton

7.4 Approche naïve (anneau)

Nous pourrions imaginer un système où le jeton est transmis de manière circulaire entre les processus. Chaque processus, lorsqu'il reçoit le jeton, vérifie s'il a une demande en attente pour la section critique. Si oui, il entre dans la section critique; sinon, il transmet le jeton au processus suivant.

- Jeton transite sur la boucle infiniment
- Un demandeur attend que le jeton lui arrive
- Il garde alors le jeton pendant sa SC
- Jusqu'à avoir fini, puis le fait passer plus loin

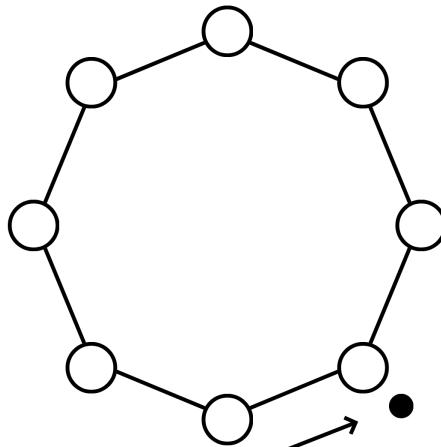


Fig. 56. – Capture des slides du cours – Approche circulaire naïve

⚠ Warning

Si nous prenons en compte les pannes de processus, cette approche devient problématique. En effet, si un processus tombe en panne, le jeton peut être perdu, ce qui empêche tous les autres processus d'accéder à la section critique.

Le temps d'attente n'est pas optimal, car un jeton ayant transmis le jeton et ayant une demande en attente doit attendre que le jeton fasse tout le tour avant de pouvoir entrer en section critique.
La structure en anneau n'est pas efficace.

7.5 Approche avec un arbre (Raymond)

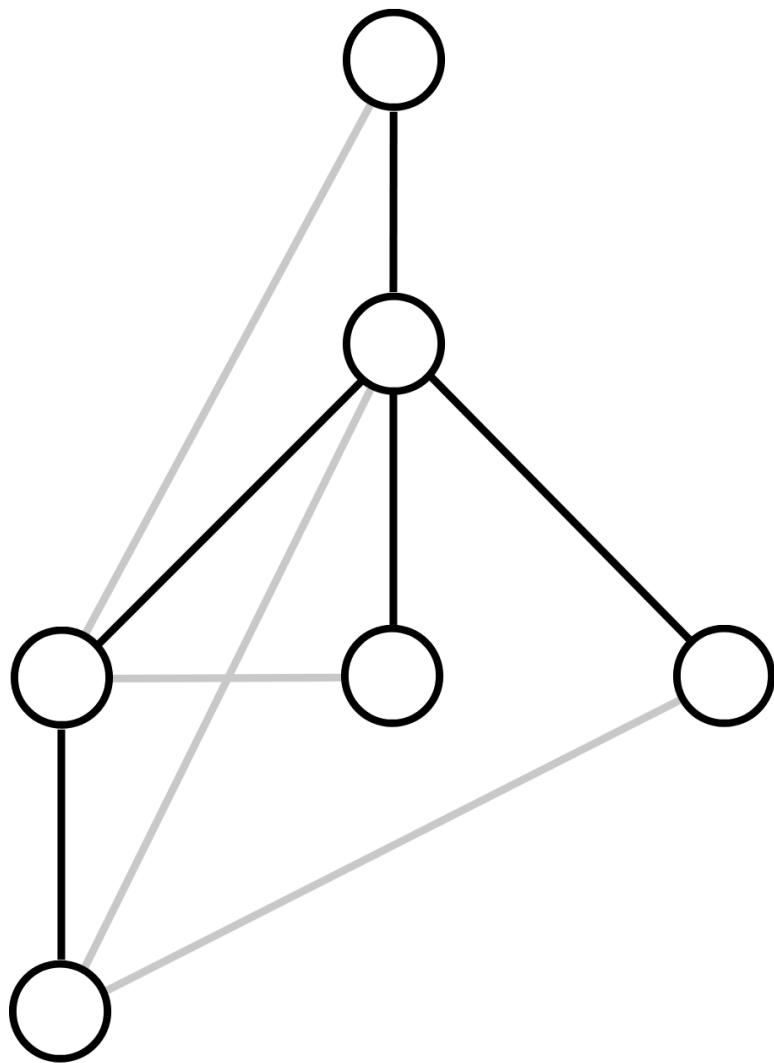
L'avantage de la structure en arbre est que la profondeur de celui-ci est

$$\log(n) \quad (n = \text{nombre de processus})$$

Ainsi, le temps d'attente pour obtenir le jeton est réduit.

7.5.1 Propriétés

- **Efficacité:** S'il est équilibré, distance logarithmique
- **Simplicité:** Un seul chemin entre tous les points
- **Réalisme:** Les vrais réseaux sont rarement des cliques



(Pourra être intégré sur un réseau plus complet)

Fig. 57. – Capture des slides du cours – Approche avec un arbre

Cette approche nous oblige à denouveau gérer de la communication entre les processus pour la demande et la libération du jeton.

7.5.2 Demande du jeton

Dans notre situation, `c` possède le jeton. `e` en fait la demande il prendra donc le chemin qui le mènera vers `c`. Chaque intermédiaire sur le chemin (ici `B`) va stocker la demande de `e` dans une file d'attente locale.

Au moment où `c` libère le jeton, il le transmet à `B` qui le transmettra à `E` (puisque c'est la première demande dans sa file d'attente) et chaque processus **inverse** le sens de son arc.

j Info

Chaque noeud ne connaît que son parent et ses enfants. Lorsqu'un noeud reçoit le jeton, il le transmet au premier demandeur dans sa file d'attente. Donc c ne sait pas que e a fait la demande, il sait juste que b lui a demandé le jeton.

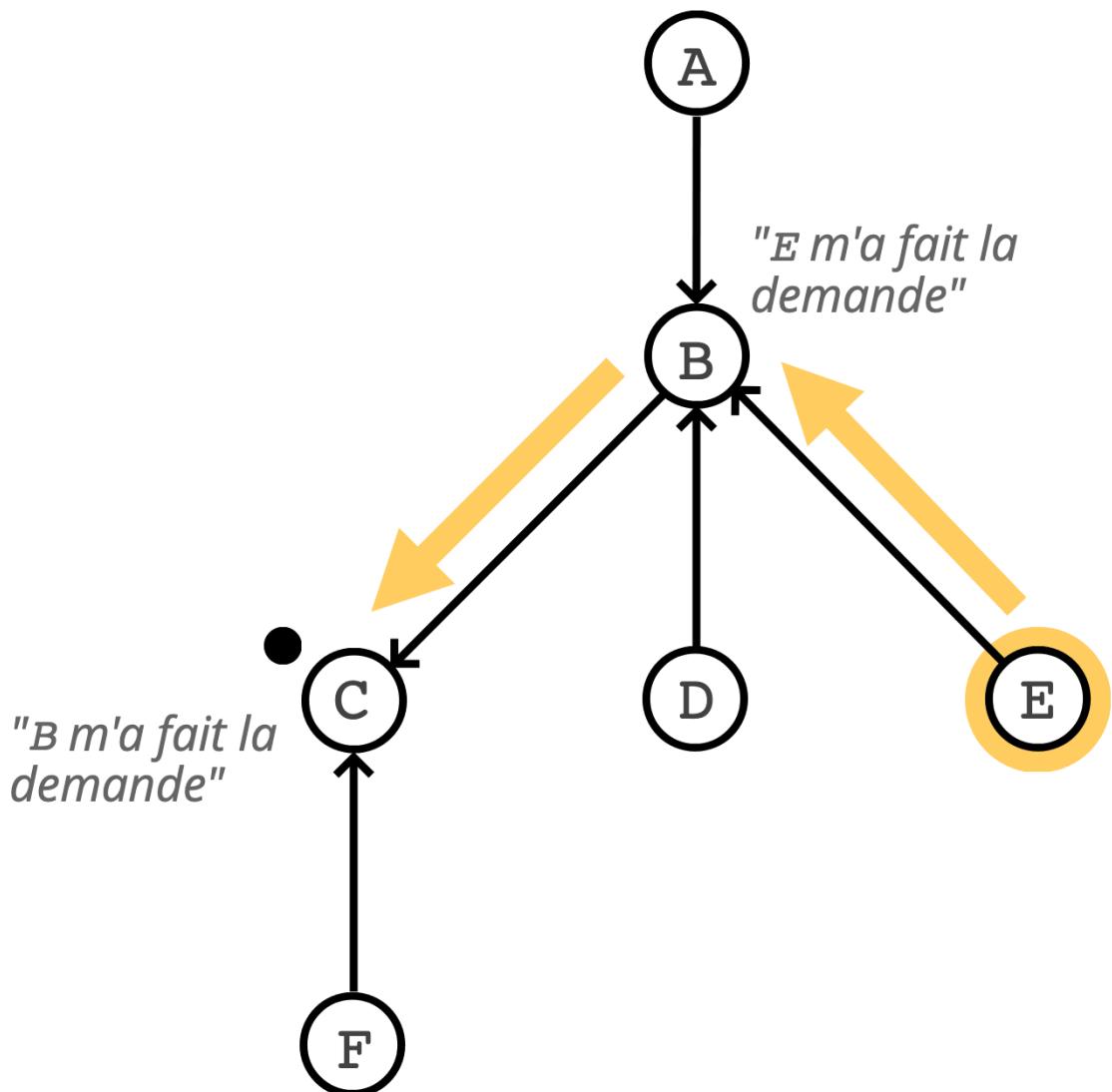


Fig. 58. – Capture des slides du cours – Demande du jeton

7.5.3 Demande multiple

Dans notre cas, si `D` et `A` viennent demander le jeton, `B` va simplement ajouter `A` à sa file d'attente. Lorsque `E` aura terminé, le jeton sera transmis à demandeur le plus ancien dans la file d'attente, ici `D`. En transmettant le jeton à `D`, vu qu'une autre demande est en attente, `B` demandera le jeton à `D` pour le transmettre à `A`.

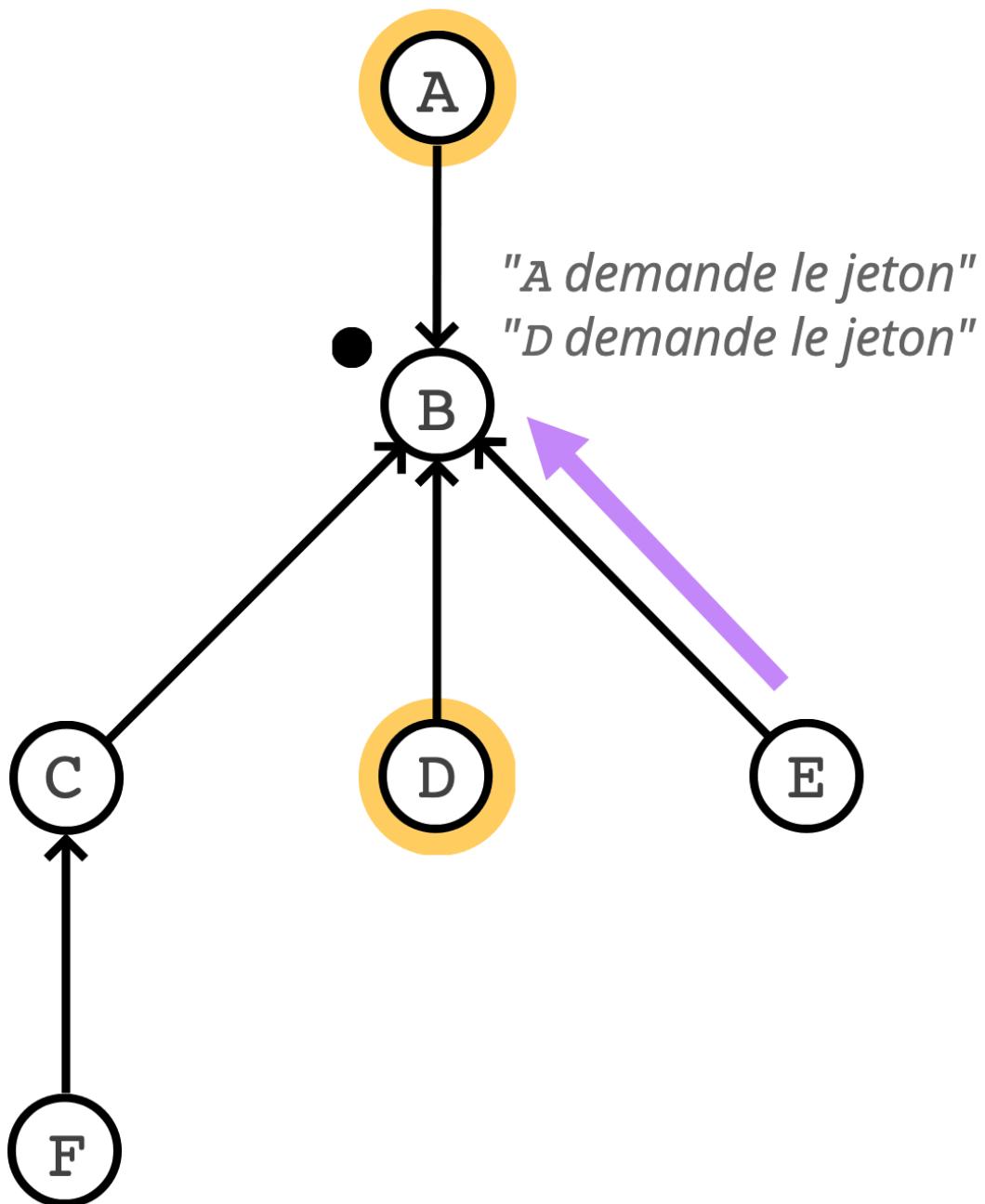


Fig. 59. – Capture des slides du cours – Demande multiple

⚠ Warning

À l’exception des feuilles, cette approche rends notre réseau sensible aux pannes, car si un noeud intermédiaire tombe en panne, **tous ses descendants** sont isolés du reste de l’arbre.

7.5.4 Résumé

Dans cette approche nous aurons besoin de 2 types de messages:

- REQUEST : pour demander le jeton
- OK : pour transmettre le jeton

Les règles à respecter sont:

- Les liens doivent toujours être orientés vers le détenteur du jeton.
- Chaque noeud possède une file d'attente pour stocker les demandes.
- Renvoi d'une requête après le jeton si besoin.
- On ne contacte le parent que si la file d'attente est vide.

Nous pouvons donc déduire les propriétés suivantes:

- **Correctness:** Jamais plus d'un processus sera en SC
- **Progress:** Toute demande finira par être satisfaite
- **Complexity:** $4 \log(n)$ par demande si l'arbre est équilibré

7.5.5 Pseudo-code

Variables

<code>self</code>	<i>entier, constant</i>	mon numéro de processus
<code>inSC</code>	<i>booléen, init false</i>	ssi je suis actuellement en SC
<code>hasRequested</code>	<i>booléen, init false</i>	ssi j'ai fait une demande de jeton
<code>parent</code>	<i>entier</i>	id du processus parent dans l'arbre
<code>queue</code>	<i>FIFO</i>	Processus ayant fait une requête

Note : on appelle parent le processus dans la direction duquel aller pour se diriger vers le jeton.

Ainsi, si parent est nil, alors on peut dire qu'on a le jeton.

Fig. 60. – Capture des slides du cours – Pseudo-code de l'algorithme de Raymond

Initialisation

Écouter infiniment les événements suivants :

- Demande d'entrée en SC
- Sortie de SC
- Réception de REQ
- Réception de OK

Fig. 61. – Capture des slides du cours – Pseudo-code de l'algorithme de Raymond - suite

Traitement : Demande de SC de la couche applicative

```
Envoi de REQ à self
```

Traitement : Réception de REQ de la part de i

```
Push i dans queue
Si parent est nil
    Exécuter handleJeton
Sinon, si hasRequested est false
    Envoyer REQ à parent
    hasRequested ← true
```

Fig. 62. – Capture des slides du cours – Pseudo-code de l'algorithme de Raymond - suite

Traitement : Réception de OK de la part de i

```
Exécuter handleJeton
```

Traitement : Sortie de SC de la couche applicative

```
Exécuter handleJeton
```

Fig. 63. – Capture des slides du cours – Pseudo-code de l'algorithme de Raymond - suite

Fonction handleJeton

```
Si queue est vide, sortir de cette fonction.
p ← queue.pop()
hasRequested ← false
Si p vaut self
    parent ← nil
    Entrer en SC
Sinon
    Envoyer OK à p
    parent ← p
    Si queue n'est pas vide
        Envoyer REQ à p
```

Fig. 64. – Capture des slides du cours – Pseudo-code de l'algorithme de Raymond - fin