

SDR - Systèmes Distribués et Repartis

Mutex par jetons

02 November 2025

Table des matières

1	Introduction	1
1.1	Petit rappel	2
2	Algorithme par jeton	2
2.1	Critères de réussite	3
2.2	Approche naïve (anneau)	3
2.3	Approche avec un arbre (Raymond)	4
2.3.1	Propriétés	4
2.3.2	Demande du jeton	5
2.3.2.1	Demande multiple	5
2.3.3	Resumé	5
2.3.4	Pseudo-code	6

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.

1.1 Petit rappel

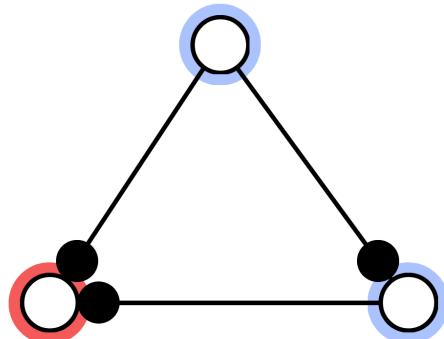
- **Lamport:** horloge logique pour ordonner les événements
- **Ricart & Agrawala:** demande de permission à tous les processus avant d'entrer en section critique
- **Carvalho et Roucairol:** demande de permission à un sous-ensemble de processus (passage de témoin entre 2 processus)

2 Algorithme par jeton

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

Algorithme par jetons

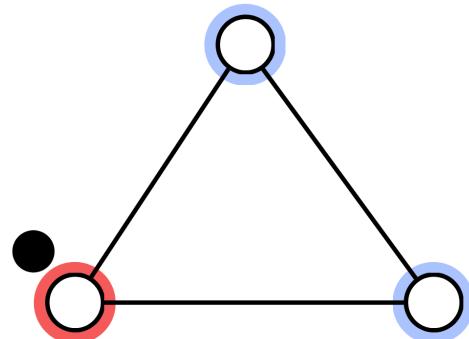
Un jeton par paire de voisins.



"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. 1. – Capture des slides du cours – Différents multi-jetons à jeton unique

2.1 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

2.2 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

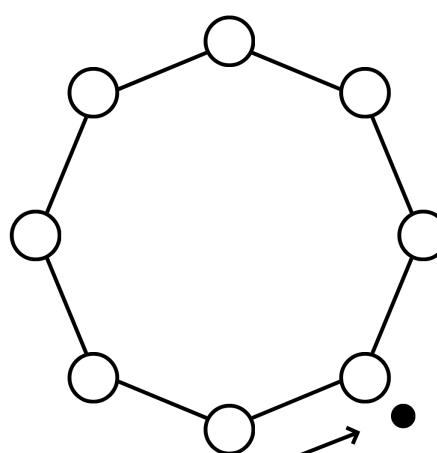


Fig. 2. – Capture des slides du cours – Approche circulaire naïve

⚠ 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.

2.3 Approche avec un arbre (Raymond)

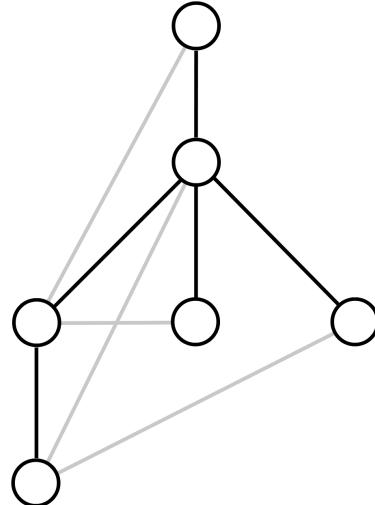
L'avantage de la structure en arbre est que la profondeur de celui-ci est

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

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

2.3.1 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. 3. – 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.

2.3.2 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.

i 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.

2.3.2.1 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.**

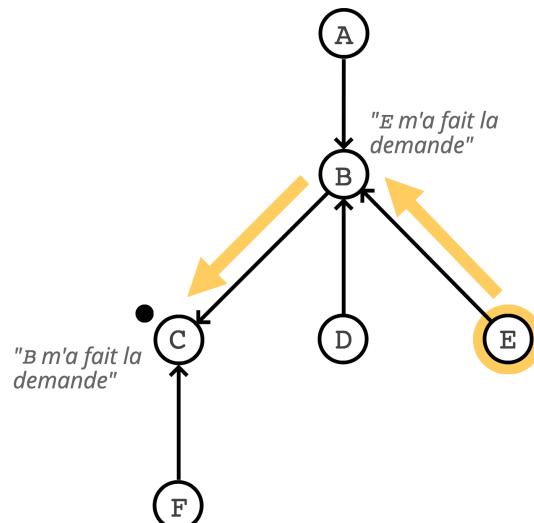


Fig. 4. – Capture des slides du cours – Demande du jeton

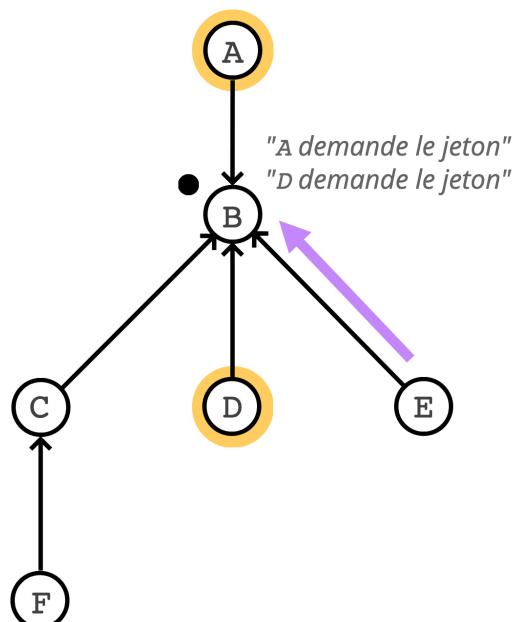


Fig. 5. – Capture des slides du cours – Libération du jeton

Warning

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

2.3.3 Resumé

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**

2.3.4 Pseudo-code

Variables

<code>self</code>	<code>entier, constant</code>	mon numéro de processus
<code>inSC</code>	<code>booléen, init false</code>	ssi je suis actuellement en SC
<code>hasRequested</code>	<code>booléen, init false</code>	ssi j'ai fait une demande de jeton
<code>parent</code>	<code>entier</code>	id du processus parent dans l'arbre
<code>queue</code>	<code>FIFO</code>	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. 6. – 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. 7. – Capture des slides du cours – Pseudo-code de l'algorithme de Raymond - suite

Traitement : Demande de SC de la couche applicative

Envoyer de `REQ` à `self`

Traitement : Réception de `REQ` de la part de `i`

```

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

Fig. 8. – Capture des slides du cours – Pseudo-code de l'algorithme de Raymond - suite

Traitement : Réception de `OK` de la part de `i`

Exécuter `handleJetton`

Traitement : Sortie de SC de la couche applicative

Exécuter `handleJetton`

Fig. 9. – Capture des slides du cours – Pseudo-code de l'algorithme de Raymond - suite

Fonction `handleJetton`

```

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