République Algérienne Démocratique et Populaire
Ministère de l'Enseignement Supérieur et de la Recherche Scientifique

École nationale Supérieure d'Informatique

Étude et Synthèse d'Algorithmes pour la Synchronisation Répartie

Réalisé par :
Elissa Tagzirt

Encadré par :
Ens Menacer Djamel Eddine

Groupe : 2 CS SIQ1

Année universitaire 2023-2024

Contents

Chapter 1

Introduction

1.1 Contexte et Motivation

La synchronisation répartie est une composante essentielle des systèmes distribués, où plusieurs processus interagissent sans mémoire commune ni horloge globale. Dans ces environnements, les processus doivent coordonner leurs actions en échangeant des messages, ce qui présente des défis uniques en matière de gestion de la cohérence des données, d'évitement des interblocages et de maintien d'une collaboration efficace.

L'importance de la synchronisation dans les systèmes distribués est particulièrement visible dans plusieurs applications pratiques :

- **Bases de Données Distribuées** : La synchronisation est nécessaire pour maintenir la cohérence des données lorsque plusieurs transactions sont exécutées simultanément sur différents nœuds.
- **Systèmes de Fichiers** : L'accès concurrent aux fichiers partagés par plusieurs processus doit être coordonné pour éviter les conflits et les incohérences.
- **Réseaux de Capteurs** : La coordination des capteurs pour collecter et analyser des données de manière synchronisée est essentielle pour obtenir des résultats précis et fiables.
- **Systèmes de Contrôle Industriels** : La synchronisation des actions des machines et des robots est cruciale pour assurer un fonctionnement fluide et sans erreurs dans les environnements industriels.

La nécessité de la synchronisation répartie dans ces contextes souligne l'importance de développer des algorithmes robustes et efficaces pour gérer la coordination dans les systèmes distribués.

1.2 Objectifs de l'Étude

Cette étude vise à :

- **Présenter et Analyser les Principaux Algorithmes de Synchronisation Répartie** :
  - **Horloges Logiques, Vectorielles et Matricielles** : Décrire les horloges de Lamport, les horloges vectorielles de Mattern & Fidge, et les horloges matricielles pour définir un ordre des événements et vérifier la causalité.
  - **Algorithmes utilisant des Jetons (Tokens)** : Présenter les algorithmes de Ricart-Agrawala, Suzuki-Kasami, l'algorithme de Le Lann (LL77), et l'algorithme de Misra pour gérer l'exclusion mutuelle.
  - **Algorithmes de Consensus** : Explorer Paxos et Raft pour parvenir à un accord entre plusieurs nœuds.
- **Comparer ces Algorithmes** : Effectuer une analyse détaillée en fonction de la complexité, de l'évolutivité, de la tolérance aux pannes et des applications pratiques.
- **Discuter des Modèles de Cohérence et de Diffusion** :
  - **Modèles de Cohérence** : Explorer la notion de coupures cohérentes et l'utilisation des horloges vectorielles pour maintenir la cohérence des états.
  - **Diffusion** : Présenter les techniques de diffusion fiable, causale et totalement ordonnée pour la coordination des processus.
- **Analyser les Avantages et les Inconvénients** : Examiner les performances, la complexité et la tolérance aux pannes de chaque algorithme, avec des recommandations d'utilisation pour différents scénarios.
- **Explorer les Perspectives et Travaux Futurs** : Identifier les défis actuels et proposer des directions futures pour la recherche, en tenant compte des nouvelles technologies comme les blockchains et les réseaux de capteurs évolués.

Chapter 2

Concepts de Base

2.1 Systèmes Distribués vs Centralisés

2.1.1 Systèmes Centralisés

- **Horloge** : Une horloge unique synchronise tous les processus, facilitant la coordination des actions. Par exemple, si deux processus veulent accéder à une ressource partagée, l'horloge peut déterminer lequel y accède en premier.
- **Mémoire** : Les processus partagent une mémoire commune, permettant un accès direct et rapide aux données. Cela simplifie la communication et la coordination puisque chaque processus peut lire et écrire dans des emplacements de mémoire partagés.
- **Communication** : La communication se fait directement via la mémoire partagée, ce qui est rapide et efficace. Les processus peuvent communiquer en modifiant les données dans la mémoire commune.

2.1.2 Systèmes Distribués

- **Horloge** : Chaque processus a sa propre horloge, ce qui complique la coordination car il n'y a pas de référence temporelle unique. Les processus doivent synchroniser leurs horloges pour coordonner leurs actions.
- **Mémoire** : Chaque processus dispose de sa propre mémoire locale, ce qui signifie qu'il n'y a pas de mémoire partagée pour l'échange direct d'informations. Les processus doivent échanger des messages pour partager des données.
- **Communication** : Les processus communiquent par l'envoi et la réception de messages. Cela introduit des délais de communication et des incertitudes sur le moment où les messages seront reçus, compliquant la coordination.

2.1.3 Comparaison des Systèmes Centralisés et Distribués

Ce tableau résume en quelques points les principales différences:

Table 2.1: Comparaison entre Systèmes Centralisés et Distribués

Critères	Systèmes Centralisés	**Systèmes Distribués**
Coordination	Coordination simple avec horloge et mémoire partagée	Utilisation d'algorithmes de synchronisation
Tolérance aux Pannes	Défaillance du nœud central entraîne la défaillance du système	Défaillance d'un nœud n'affecte pas l'ensemble du système
Scalabilité	Limitée par les ressources de la machine unique	Répartie sur plusieurs machines, augmentant la capacité de traitement

2.2 Problématiques de la Synchronisation

2.2.1 Absence de Mémoire Commune

- **Complexité** : Les processus doivent échanger des messages pour partager des informations, ce qui ajoute une couche de complexité. Par exemple, pour garantir la cohérence des données, les processus doivent implémenter des protocoles de communication robustes.

- **Délais de Communication** : Les messages peuvent être retardés ou perdus, compliquant la coordination. Les processus doivent gérer ces délais pour éviter des incohérences ou des situations de blocage.

2.2.2 Absence d'Horloge Globale

- **Ordonnancement des Événements** : Il est difficile de déterminer l'ordre exact des événements survenant dans différents processus sans une horloge commune. Les algorithmes doivent donc compenser cette absence en utilisant des horloges logiques ou vectorielles.

- **Cohérence des Données** : Maintenir la cohérence des données est plus difficile sans une horloge globale pour synchroniser les mises à jour et les accès aux données partagées. Par exemple, deux processus peuvent essayer de modifier la même donnée en même temps, ce qui peut entraîner des incohérences.

2.2.3 Asynchronisme des Communications et des Traitements

- **Synchronisation des Processus** : Les processus peuvent s'exécuter à des vitesses différentes et les messages peuvent arriver à des moments imprévisibles. Par exemple, un processus peut attendre indéfiniment une réponse qui est retardée ou perdue.
- **Gestion des Délais** : Les systèmes doivent être conçus pour gérer les délais de communication et les différences de vitesse d'exécution des processus. Cela peut compliquer la conception des algorithmes de synchronisation.
- **Tolérance aux Pannes** : L'asynchronisme complique la détection et la gestion des pannes. Par exemple, il peut être difficile de distinguer une panne de réseau d'un simple retard de communication. Les algorithmes doivent inclure des mécanismes de détection de pannes robustes pour maintenir la fiabilité du système.

2.3 Conclusion des Concepts de Base

Les systèmes distribués présentent des défis uniques par rapport aux systèmes centralisés, principalement en raison de l'absence de mémoire commune et d'horloge globale, ainsi que de l'asynchronisme des communications et des traitements. Ces défis nécessitent des algorithmes sophistiqués pour assurer la synchronisation et la coordination efficace des processus.

Chapter 3

Modèles Temporels

3.1 Modèle Asynchrone

3.1.1 Caractéristiques et Principe

- **Indépendance des Processus** : Chaque processus fonctionne de manière indépendante sans synchronisation explicite avec les autres.
- **Absence de Garanties Temporelles** : Aucun délai de communication ou de traitement n'est garanti, les messages peuvent être retardés ou perdus.
- **Événements Imprévisibles** : Les événements se produisent sans ordre global, rendant la synchronisation difficile.

Exemple : Un processus peut envoyer un message sans savoir quand il sera reçu ni quand une réponse sera envoyée.

3.2 Modèle Synchrone

Les modèles temporels dans les systèmes distribués sont essentiels pour coordonner les actions des processus et garantir la cohérence des données. Deux principaux modèles sont utilisés : les modèles asynchrone et synchrone. Chacun présente des caractéristiques uniques et des défis spécifiques.

3.2.1 Caractéristiques et Principe

- **Synchronisation Temporelle** : Les processus fonctionnent en synchronisation avec une horloge globale ou des horloges locales synchronisées.
- **Garanties Temporelles** : Les délais de communication et de traitement sont bornés et connus, permettant une coordination précise.
- **Ordonnancement des Événements** : Les événements sont ordonnés selon une horloge globale, facilitant la coordination.

Exemple : Un processus peut envoyer un message avec l'assurance qu'il sera reçu dans un délai spécifique et que la réponse arrivera également dans un délai prédéterminé.

3.3 Comparaison

- **Fiabilité des Prévisions** : Les modèles synchrones permettent des prévisions fiables, contrairement aux modèles asynchrones.
- **Complexité des Algorithmes** : Les algorithmes asynchrones sont plus complexes à cause des retards et pertes de messages. Les algorithmes synchrones sont plus simples.
- **Tolérance aux Pannes** : Les modèles asynchrones sont plus tolérants aux pannes. Les modèles synchrones dépendent des garanties temporelles et peuvent échouer en cas de panne.

Pour gérer cette coordination temporelle, divers modèles et mécanismes de synchronisation ont été développés. Les horloges logiques et vectorielles jouent un rôle crucial en permettant d'ordonner les événements dans un système distribué sans horloge globale.

3.4 Horloges Logiques

3.4.1 Algorithme de Lamport

Les horloges logiques, introduites par Lamport en 1978, permettent de dater les événements de manière à respecter l'ordre causal sans nécessiter d'horloge physique commune. Chaque processus maintient une horloge locale, et lors de la communication entre processus, les horloges sont mises à jour pour refléter l'ordre des événements.

Fonctionnement :

- Chaque processus P_i maintient une horloge H_i.
- Lors d'un événement local, H_i est incrémentée : $H_i = H_i + 1$.
- Lors de l'envoi d'un message m, le message est estampillé avec l'horloge actuelle : $m = (m, H_i)$.
- À la réception du message m par le processus P_j, ce dernier met à jour son horloge : $H_j = \max(H_j, H_i) + 1$.

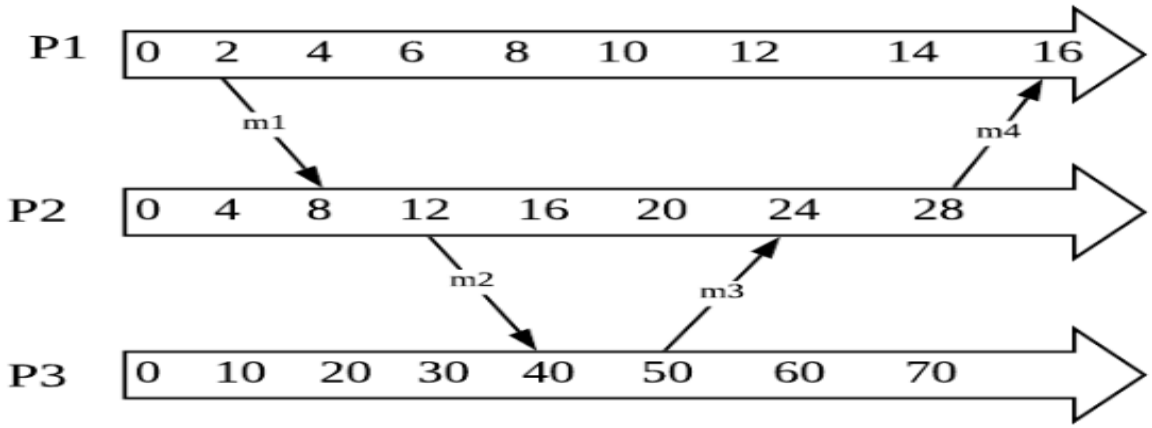

Figure 3.1: Illustration de l'algorithme de Lamport.

Exemple :

- Chaque processus commence avec une horloge initiale.
- P_1 commence à 0 et incrémente son horloge locale de 2 en 2 à chaque événement local.
- P_1 envoie le message m_1 à P_2 avec l'horloge 8.
- À la réception de ce message, P_2 met à jour son horloge à 9 (car $\max(8, 4) + 1 = 9$).
- P_2 envoie le message m_2 à P_3 et l'horloge de P_3 est mise à jour de manière similaire.
- Finalement, P_1 envoie m_4 à P_2.

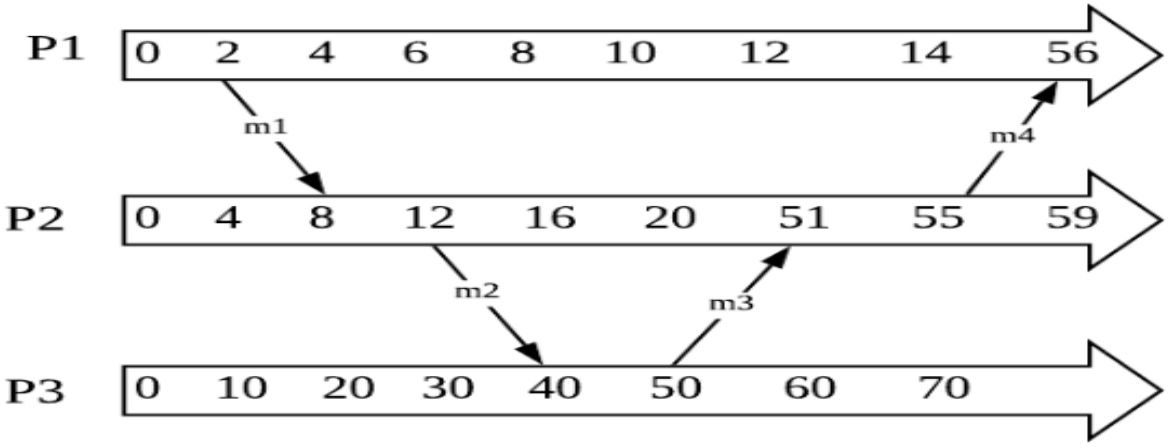

Figure 3.2: Limites de l'algorithme de Lamport avec les horloges logiques.

Limites de l'algorithme de Lamport :

- Les horodatages logiques de Lamport obéissent à la règle de causalité mais ne peuvent pas faire la distinction entre les événements occasionnels et concurrents.
- Les événements sans relation de causalité apparaissent avec des horodatages qui respectent la règle de causalité mais ne permettent pas de déterminer leur ordre exact.

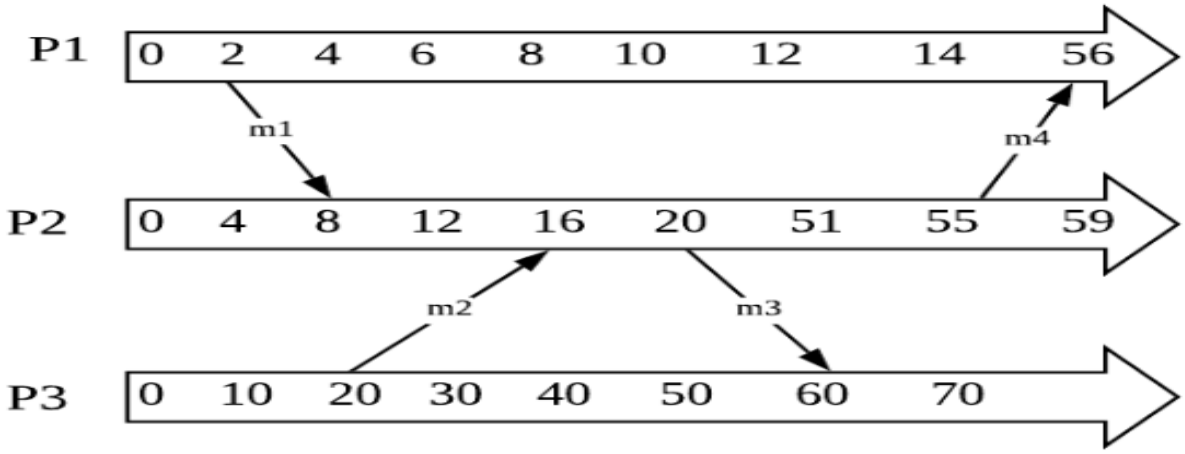

Figure 3.3: Illustration de l'horloge logique de Lamport montrant les événements concurrents.

Dans les figures 3.2 et 3.3, on peut voir que les horloges des processus P_1, P_2 et P_3 reflètent les événements de manière causale, mais ne permettent pas de distinguer les événements concurrents.

Conclusion

L'algorithme de Lamport est efficace pour maintenir l'ordre causal des événements dans un système distribué. Cependant, il présente des limites en ce qui concerne la différenciation entre les événements occasionnels et concurrents, ce qui peut poser des problèmes dans certaines applications distribuées.

3.4.2 Algorithme des Horloges Vectorielles

Les horloges vectorielles, ou horloges de Mattern Fidge, améliorent les horloges logiques en fournissant plus d'informations sur l'ordre des événements à travers les processus.

Fonctionnement :

1. Chaque processus P_i maintient un vecteur d'horloges V_i de taille N (nombre de processus).
2. Lors d'un événement local, l'entrée correspondante du vecteur est incrémentée : $V_i[i] = V_i[i] + 1$.
3. Lors de l'envoi d'un message m, le vecteur entier est estampillé sur le message : $m = (m, V_i)$.

4. À la réception du message m par P_j, ce dernier met à jour son vecteur :
   - $V_j[i] = \max(V_j[i], V_i[i])$ pour chaque i,
   - puis $V_j[j] = V_j[j] + 1$.

Relations entre événements :

- $V_a = V_b$ si et seulement si $V_a[i] = V_b[i]$ pour tout $i = 1, \ldots, N$.
- $V_a \leq V_b$ si et seulement si $V_a[i] \leq V_b[i]$ pour tout $i = 1, \ldots, N$.
- Les événements a et b sont causalement liés si $V_a < V_b$ c'est-à-dire $V_a \leq V_b$ et il existe j tel que $1 \leq j \leq N$ et $V_a[j] < V_b[j]$.
- Les événements a et b sont concurrents si et seulement si $\neg(V_a \leq V_b)$ et $\neg(V_b \leq V_a)$.

Illustration des horloges vectorielles :

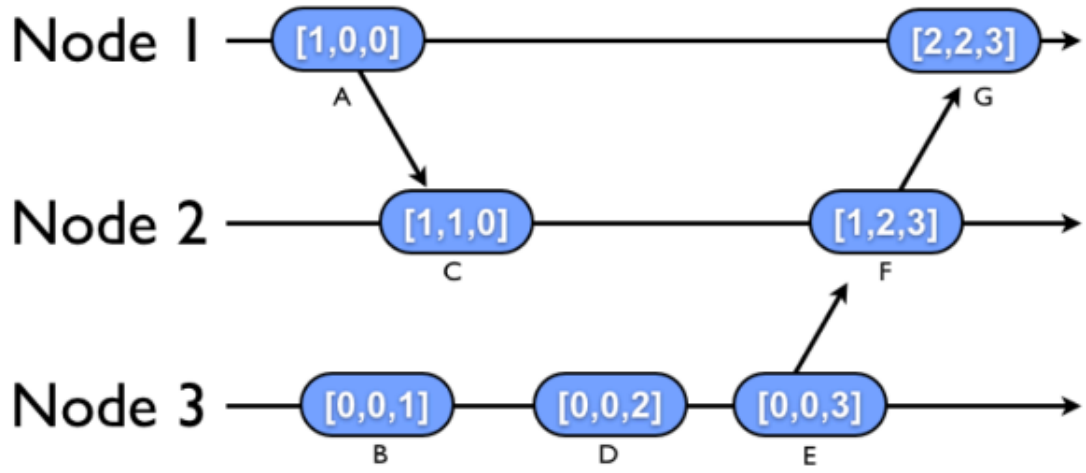

Figure 3.4: Illustration de l'algorithme des horloges vectorielles.

Limites des Horloges Vectorielles :

- Précédence Causale : Les horloges vectorielles ne permettent pas de garantir une délivrance causale des messages. Elles détectent a posteriori la violation de l'ordre causal dans la réception des messages.

3.4.3 Horloges Matricielles

Les horloges matricielles vont encore plus loin en capturant non seulement les horloges locales mais aussi les horloges des autres processus connues par chaque processus. Ce mécanisme permet de résoudre certains problèmes présents dans les horloges vectorielles et de Lamport, tout en assurant la causalité de manière plus robuste.

Fonctionnement

1. **Matrice d'horloges** : Chaque processus P_i maintient une matrice d'horloges M_i de taille $N \times N$, où N est le nombre total de processus dans le système.
2. **Événement local** : Lors d'un événement local, l'entrée correspondante de la matrice est incrémentée : $M_i[i][i] = M_i[i][i] + 1$.
3. **Envoi de message** : Lors de l'envoi d'un message m, la matrice entière est estampillée sur le message : $m = (m, M_i)$.
4. **Réception de message** : À la réception du message m par P_j, ce dernier met à jour sa matrice :
   - $M_j[k][i] = \max(M_j[k][i], M_i[k][i])$ pour chaque k,
   - $M_j[j][i] = M_j[j][i] + 1$.

Vérification de la réception correcte

- **Condition de Délivrance** :
  - Tous les messages causalement antérieurs doivent être délivrés avant de délivrer le message courant.
  - Si $HM_m[j,i] = HM_i[j,i] + 1$ et pour tout $k \neq i, j$, $HM_m[k,i] \leq HM_i[k,i]$, alors le message est délivré. Sinon, il est mis en attente.

Illustration des Horloges Matricielles

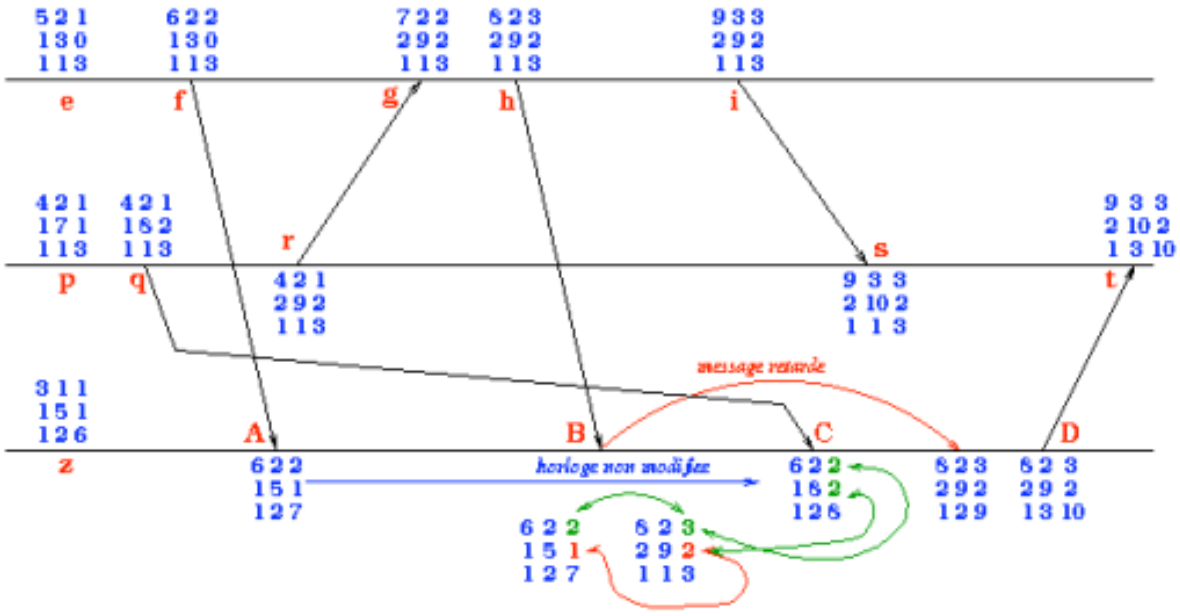

Figure 3.5: Illustration des horloges matricielles

Explication de l'Illustration

1. **Événements Locaux** :
   - Chaque processus A, B, C, et D incrémente son horloge locale dans sa matrice d'horloges respectivement. Par exemple, $M_A[A][A]$ est incrémenté lors d'un événement local dans le processus A.
2. **Envoi et Réception de Messages** :
   - Lorsque A envoie un message à B, la matrice M_A est estampillée sur le message. À la réception, B met à jour sa matrice M_B en prenant le maximum de chaque entrée de M_A et M_B.
3. **Assurance de la Causalité** :
   - L'illustration montre comment les matrices d'horloges sont mises à jour pour respecter la causalité. Par exemple, le message envoyé de B à C ne sera délivré que si toutes les conditions de causalité sont respectées, c'est-à-dire, si les horloges matricielles montrent que tous les messages causalement antérieurs ont été délivrés.
4. **Condition de Délivrance** :
   - La formule $HM_m[j,i] = HM_i[j,i] + 1$ et pour tout $k \neq i,j$, $HM_m[k,i] \leq HM_i[k,i]$, est illustrée comme suit :
     - Supposons que B envoie un message à C. C vérifiera les conditions suivantes avant de délivrer le message :
       * $HM_B[j,i] = HM_A[j,i] + 1$: Cela signifie que la valeur de l'horloge pour le message reçu par B est exactement une unité supérieure à celle de A.
       * Pour tout $k \neq i,j$, $HM_B[k,i] \leq HM_A[k,i]$: Cela assure que les horloges pour tous les autres processus k n'ont pas des valeurs plus grandes, garantissant ainsi la causalité.

Calcul des Valeurs

- **Initialisation** :
  - Avant l'envoi de tout message, supposons que M_A, M_B, M_C, et M_D sont initialisés comme suit :

$$M_A = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}, \quad M_B = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}, \quad M_C = \begin{pmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix},$$

$$\mathrm{M}_D = \begin{pmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

- **Envoi de Message** :
  - Lorsque A envoie un message à B, M_A est estampillée sur le message. Disons que M_A est maintenant

$$M_A = \begin{pmatrix} 2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}.$$

  - B reçoit ce message et met à jour M_B :

$$M_B = \begin{pmatrix} 2 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}.$$

- **Réception de Message et Vérification de Causalité** :
  - Lorsque B envoie un message à C, supposons que M_B est

$$M_B = \begin{pmatrix} 2 & 2 & 0 & 0 \\ 0 & 2 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}.$$

  - C reçoit ce message et vérifie les conditions de causalité avant de le traiter :
    - $HM_C[j,i] = HM_B[j,i] + 1$
    - Pour tout $k \neq i, j$, $HM_C[k,i] \leq HM_B[k,i]$

– Si ces conditions sont satisfaites, C met à jour sa matrice :

$$M_C = \begin{pmatrix} 2 & 2 & 1 & 0 \\ 0 & 2 & 1 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}.$$

Avantages des Horloges Matricielles

1. **Captation des Dépendances Indirectes** : En capturant les horloges de chaque processus, les horloges matricielles assurent que même les dépendances indirectes sont prises en compte.

2. **Assurance de la Causalité** : La mise à jour des matrices d'horloges lors de l'envoi et de la réception des messages garantit que la causalité est préservée, évitant ainsi les anomalies de synchronisation.

3. **Mise à Jour Simultanée** : Chaque processus met à jour sa matrice d'horloges en fonction des messages reçus, assurant une synchronisation correcte et une délivrance des messages dans l'ordre causal.

Chapter 4

Modèles de Cohérence

4.1 Notion d'État et de Coupure (Cut)

4.1.1 Définition de la Coupure

- Une coupure (cut) est une "photo instantanée" du système distribué qui capture l'état de chaque processus à un moment donné, ainsi que les messages en transit.
- **Formellement** :
  - Une coupure est un ensemble d'événements contenant tous les événements précédents pour chaque processus.
  - Si $e \in C$ et e' précède localement e, alors $e' \in C$.

4.1.2 Utilisation de la Coupure

Les coupures sont utilisées pour analyser l'état global d'un système distribué à un instant donné. Elles permettent de figer l'état de chaque processus et les messages en transit.

4.2 Cohérence des Coupures

4.2.1 Définition de la Cohérence

- Une coupure est dite cohérente si elle respecte l'ordre causal :
  - Si $e \in C$ et $e' \to e$, alors $e' \in C$.
- **Propriété** : Tout message reçu dans la coupure a été envoyé dans la coupure.

4.2.2 Importance des Coupures Cohérentes

Les coupures cohérentes sont cruciales pour l'analyse correcte des états des systèmes distribués. Elles garantissent que l'état global observé respecte l'ordre causal des événements, évitant ainsi des incohérences comme des messages "venant du futur".

4.2.3 Horloges Vectorielles pour la Cohérence

Les horloges vectorielles permettent de maintenir et de vérifier l'ordre causal des événements entre les processus, garantissant ainsi la cohérence des coupures dans les systèmes distribués.

- **Cohérence des Coupures par Horloges Vectorielles** :
  - Une coupure est cohérente si, pour chaque événement de réception dans la coupure, l'événement d'émission correspondant est également dans la coupure.
  - L'horloge vectorielle associée à une coupure C est définie par :

    $$EV(C) = \langle EV(e_1)[1], ..., EV(e_j)[j], ..., EV(e_n)[n] \rangle$$

    où e_j est l'événement le plus récent du site S_j appartenant à la coupure.
  - La coupure est cohérente si et seulement si :

    $$EV(C) = \sup(EV(e_1), EV(e_j), ..., EV(e_n))$$

    avec :

    $$\sup(EV(e_1)[k], EV(e_j)[k], EV(e_n)[k])$$

  Cette définition assure que tous les événements antérieurs à un événement dans la coupure sont également inclus dans la coupure, garantissant ainsi la cohérence.

4.3 Relation avec l'Étude et Synthèse d'Algorithmes pour la Synchronisation Répartie

L'étude et la synthèse d'algorithmes pour la synchronisation répartie nécessitent une compréhension approfondie des mécanismes de cohérence dans les systèmes distribués. Les concepts de coupure, de cohérence des coupures, et l'utilisation des horloges vectorielles sont essentiels pour concevoir des algorithmes efficaces de synchronisation. Voici comment ils s'intègrent dans l'étude :

- **Analyse de l'État Global** : Les coupures permettent de capturer l'état global du système à un instant donné, crucial pour analyser et déboguer les algorithmes de synchronisation.
- **Maintien de la Cohérence** : Les coupures cohérentes assurent que l'état global du système respecte l'ordre causal des événements, évitant les incohérences qui pourraient perturber les algorithmes de synchronisation.
- **Ordonnancement des Événements** : Les horloges vectorielles fournissent un mécanisme pour maintenir l'ordre causal des événements, essentiel pour garantir la cohérence et la coordination entre les processus.
- **Détection et Résolution de Conflits** : En utilisant les horloges vectorielles, les algorithmes de synchronisation peuvent détecter les conflits de mise à jour et les résoudre en maintenant un ordre cohérent des événements.

Chapter 5

Algorithmes de Synchronisation

Nous allons diviser les algorithmes de synchronisation en deux grandes catégories : les algorithmes d'exclusion mutuelle et les algorithmes de synchronisation globale.

5.1 Algorithmes d'Exclusion Mutuelle

5.1.1 Algorithmes Basés sur les Horloges Logiques

Algorithme de Lamport (1978)

Principe :

- Chaque processus doit recevoir la permission de tous les autres sites pour entrer en section critique (SC).
- Chaque site maintient une file dattente des processus désirant entrer en SC.
- L'ordre des requêtes est déterminé par les estampilles de Lamport.
- Seul le processus en tête de la file d'attente peut entrer en section critique.

Variables Locales :

- Chaque processus P_i maintient les variables suivantes :
  - h : horloge logique
  - f : tableau de messages reçus de taille n (nombre de processus)

Initialisation :

```
h = 0
f = [(rel, 0, j) pour j dans [0..n-1]]
```

Types de Messages :

- **req** : Requête pour entrer en SC.
- **rel** : Libération de la SC.
- **acq** : Accusé de réception d'une requête.

L'algorithme :

Entrée en Section Critique (SC)

Algorithm 1 Protocole de Demande d'Entrée en Section Critique (SC)

1: Incrémenter l'horloge locale et diffuser un message de requête à tous les autres processus :
2: $h = h + 1$
3: envoyer(req, h, i) à tous les processus
4: $f[i] = (req, h, i)$
5: Attendre de recevoir des acquittements de tous les autres processus avant d'entrer en SC :
6: **while** il existe un j i tel que estampille_de($f[i]$) > estampille_de($f[j]$) **do**
7: attendre
8: **end while**
9: <section critique>

Sortie en Section Critique (SC)

Algorithm 2 Protocole de Sortie de Section Critique

1: Diffuser un message de libération à tous les autres processus :
2: $h = h + 1$
3: envoyer(rel, h, i) à tous les processus
4: $f[i] = (rel, h, i)$

Gestion des Messag

Algorithm 3 À la réception d'un message de requête (req)

1: à la réception de (req, k, j):
2: $h = max(h, k) + 1$
3: $f[j] = (req, k, j)$
4: envoyer(acq, h, i) à j

Algorithm 4 À la réception d'un message de libération (rel)

1: à la réception de (rel, k, j):
2: $h = max(h, k) + 1$
3: $f[j] = (rel, k, j)$

Algorithm 5 À la réception d'un message d'acquittement (acq)

1: à la réception de (acq, k, j):
2: $h = max(h, k) + 1$
3: **if** type($f[j]$) $\neq$ req **then**
4: $\quad f[j] = (acq, k, j)$
5: **end if**

Tolérance aux Pannes

Pour rendre l'algorithme résistant aux pannes, il est nécessaire d'introduire des messages supplémentaires pour gérer les pannes et les redémarrages des sites.

Message d'absence (absent)

- Diffusé par la couche transport du réseau lorsqu'un site tombe en panne.
- Les processus doivent ignorer les requêtes provenant de sites marqués comme absents.

Message de rentrée (rentrée)

- Diffusé par un site lorsqu'il redémarre après une panne.
- Les autres sites doivent répondre avec le dernier message de requête non satisfait ou de libération.

Gestion des pannes

Algorithm 6 Gestion des Pannes

1: à la réception de (absent, HLi, i):
2: marquer le site i comme absent
3: mettre à jour la file d'attente pour ignorer les requêtes de i
4: à la réception de (rentrée, HLi, i):
5: marquer le site i comme actif
6: envoyer le dernier message de requête ou de libération à i

Détection des pannes et redémarrages

Algorithm 7 Détection des Pannes et Redémarrages

```
# La détection de la panne est à la charge de la couche transport
diffuser (absent, HLi, i) à tous les sites actifs
# Lorsqu'un site redémarre
diffuser (rentrée, HLi, i) à tous les sites actifs
```

Nombre de Messages Échangés

L'algorithme de Lamport nécessite :

- 3(n-1) messages pour réaliser une exclusion mutuelle :
  - $n-1$ messages de requête pour indiquer aux autres processus le désir de P_i de pénétrer en section critique.
  - $n-1$ messages d'acquittement pour accuser réception des requêtes.
  - $n-1$ messages de libération pour indiquer aux autres processus que la section critique a été libérée.

Limitations de l'Algorithme de Lamport

- **Tolérance aux pannes** :
  - L'algorithme s'effondre si un processus tombe en panne sans diffuser un message d'absence, car les autres processus continueront à attendre des acquittements qui ne viendront jamais.
  - Le processus qui redémarre doit informer les autres de sa reprise pour rétablir l'état du système.
- **Charge Réseau** :
  - L'algorithme génère 3(n1) messages par demande d'accès à la section critique, ce qui peut devenir un goulot d'étranglement dans les grands systèmes distribués.
- **Latence** :
  - La latence peut être élevée car chaque demande d'accès à la section critique nécessite la réception de $n-1$ acquittements avant de pouvoir entrer en section critique.
- **Synchronisation Asynchrone** :
  - L'algorithme est basé sur un modèle de communication asynchrone, où les messages peuvent arriver dans un ordre quelconque. Cela nécessite une gestion complexe des estampilles logiques pour assurer l'ordre correct des requêtes.
- **Complexité de Mise en Œuvre** :
  - La gestion des horloges logiques et des files d'attente pour chaque processus ajoute une complexité supplémentaire à l'implémentation.

5.1.2 Algorithmes Basés sur les Tokens

Algorithme de Ricart et Agrawala (1981)

Principe :

- Chaque processus doit recevoir la permission de tous les autres sites pour entrer en section critique (SC).
- Les processus échangent des messages de requête et de réponse pour coordonner l'accès à la section critique.

Variables Locales : Chaque processus P_i maintient les variables suivantes :

- **osn** : numéro de séquence de la requête (horloge logique locale).
- **hsn** : plus grand numéro de séquence vu par P_i.
- **nbrepattendues** : nombre de réponses attendues.
- **Scdemandée** : booléen indiquant si la section critique est demandée.
- **Repdifférée** : tableau de booléens pour différer les réponses.
- **Priorité** : booléen pour déterminer la priorité des requêtes.

Initialisation :

```
osn = 0
hsn = 0
nbrepattendues = 0
Scdemandée = Faux
Repdifférée = [Faux pour j dans [1..n]]
Priorité = Faux
```

Types de Messages :

- **req** : Requête pour entrer en SC.
- **rep** : Réponse à une requête.

Protocole de Demande d'Entrée en Section Critique (SC)

Algorithm 8 Protocole de Demande d'Entrée en Section Critique (SC)

1: Scdemandée = Vrai
2: osn = hsn
3: nbrepattendues = n - 1
4: **for** j i **do**
5: envoyer(req, osn, i) à j
6: **end for**
7: attendre (nbrepattendues = 0)
8: <section critique>

Sortie de Section Critique

Algorithm 9 Sortie de Section Critique

```
1: Scdemandée = Faux
2: for j dans [1..n] do
3:    if Repdifférée[j] then
4:       Repdifférée[j] = Faux
5:       envoyer(rep) à j
6:    end if
7: end for
```

Gestion des Messages Reçus

Algorithm 10 Gestion des Messages Reçus

```
1: À la réception d'un message de requête (req) :
2: à la réception de (req, k, j):
3: hsn = max(hsn, k) + 1
4: Priorité = (Scdemandée) et ((k > osn) ou ((k = osn) et (i < j)))
5: if Priorité then
6:    Repdifférée[j] = Vrai
7: else
8:    envoyer(rep) à j
9: end if
10: À la réception d'un message de réponse (rep) :
11: à la réception de (rep):
12: nbrepattendues = nbrepattendues - 1
```

Nombre de Messages Échangés

L'algorithme de Ricart-Agrawala nécessite :

- 2(n-1) messages pour réaliser une exclusion mutuelle :
  - $n-1$ messages de requête pour indiquer aux autres processus le désir de P_i de pénétrer en section critique.
  - $n-1$ messages de réponse pour accorder l'accès à la section critique.

Limitations de l'Algorithme de Ricart-Agrawala

- **Tolérance aux pannes** :
  - L'algorithme suppose que le réseau de transport est sans erreur.
  - La panne d'un processus peut entraîner une attente indéfinie des réponses nécessaires pour entrer en section critique.

- **Charge Réseau** :
  - Bien que l'algorithme soit optimisé par rapport à l'algorithme de Lamport, il génère toujours 2(n-1) messages par demande d'accès à la section critique, ce qui peut devenir un goulot d'étranglement dans les grands systèmes distribués.
- **Synchronisation Asynchrone** :
  - L'algorithme est basé sur un modèle de communication asynchrone, où les messages peuvent arriver dans un ordre quelconque. Cela nécessite une gestion complexe des estampilles logiques pour assurer l'ordre correct des requêtes.
- **Interblocage** :
  - Dans certaines conditions, si plusieurs processus envoient simultanément des requêtes, il peut y avoir un risque de starvation ou d'interblocage si les réponses sont différées de manière excessive.

Algorithme de Le Lann (LL77)

Principe :

Faire circuler un message spécial (jeton) en permanence entre tous les sites. Seul le processus possédant le jeton peut entrer en section critique.

Fonctionnement :

Utilisation d'un anneau logique virtuel pour faciliter la circulation du jeton.

Protocole :

Algorithm 11 Accès à la Section Critique dans l'Algorithme de Le Lann (LL77)

1: attendre (Jeton) venant de VG(i)
2: <section critique>
3: envoyer (Jeton) vers VD(i)

Résistance aux pannes :

Prévoir un nouvel anneau logique en cas de panne d'un site.

Nombre de Messages Échangés :

L'algorithme de Le Lann nécessite 1 message pour accéder à la section critique (le jeton).

Limitations de l'Algorithme de Le Lann :

- Tolérance aux pannes : Si un processus non-détenteur du jeton tombe en panne, il est facile de reformer l'anneau logique. Si le détenteur du jeton tombe en panne, le système doit détecter la perte du jeton et régénérer un nouveau jeton.
- Goulot d'étranglement : Le jeton doit circuler à travers tous les processus, ce qui peut créer un goulot d'étranglement dans les grands systèmes.
- Latence : Le temps d'attente pour entrer en section critique peut être élevé si le jeton doit parcourir un long chemin pour atteindre le processus demandeur.

Algorithme de Misra (1983)

Principe :

Utilisation de deux jetons pour détecter et régénérer la perte de l'autre.

Fonctionnement :

Chaque processus mémorise la valeur associée au dernier jeton vu passer.

Protocle :

Algorithm 12 Régénération de Jeton dans l'Algorithme de Misra

```
1: À la réception de (ping, nbping) :
2: if m = nbping then
3:    <pong est perdu, il est régénéré>
4:    nbping = nbping + 1
5:    nbpong = -nbping
6: else
7:    m = nbping
8: end if
9:
10: À la réception de (pong, nbpong) :
11: <traitement analogue en intervertissant les rôles de ping et pong>
12:
13: À la rencontre de deux jetons :
14: nbping = nbping + 1
15: nbpong = nbpong - 1
```

Nombre de Messages Échangés :

- L'algorithme de Misra nécessite 1 message pour chaque passage de jeton (ping ou pong) entre les processus.

Limitations de l'Algorithme de Misra :

- Tolérance aux pannes : La détection de la perte d'un jeton repose sur l'autre jeton. Si les deux jetons sont perdus simultanément, le système ne peut pas détecter la panne.
- Complexité : La gestion de deux jetons et la vérification de leur état ajoutent une complexité supplémentaire par rapport à un algorithme utilisant un seul jeton.
- Scalabilité : Comme pour l'algorithme de Le Lann, la scalabilité peut être un problème, car le jeton doit circuler à travers tous les processus.

Algorithme de Suzuki-Kasami

Principe :

- Utilise un jeton unique qui circule entre les processus. Seul le détenteur du jeton peut entrer en section critique.

Fonctionnement :

- Chaque processus maintient un tableau local pour suivre les requêtes des autres processus.
- Le jeton contient un tableau qui enregistre la demande la plus récente de chaque processus.
- Lorsqu'un processus veut entrer en section critique, il envoie une requête à tous les autres processus.
- Lorsqu'un processus libère la section critique, il passe le jeton au prochain processus dans la file d'attente.

Variables Locales :

- $RN[i]$: Requête numéro du processus i.
- $LN[i]$: Dernière requête satisfaite du processus i dans le jeton.

Protocole de Demande d'Entrée en Section Critique (SC) :

Algorithm 13 Protocole de Demande d'Entrée en Section Critique (SC) dans Suzuki-Kasami

$RN[i] \leftarrow RN[i] + 1$
Envoyer $REQUEST(i, RN[i])$ à tous les autres processus
Attendre le jeton

Protocole de Sortie de Section Critique :

Algorithm 14 Protocole de Sortie de Section Critique dans Suzuki-Kasami

for chaque processus j **do**
  if $RN[j] = LN[j] + 1$ **then**
    Ajouter j à la file d'attente
  end if
end for
$LN[i] \leftarrow RN[i]$
if file d'attente non vide **then**
  Retirer le premier processus k de la file d'attente
  Envoyer le jeton à k
end if

Gestion des Messages Reçus :

Algorithm 15 Gestion des Messages Reçus dans Suzuki-Kasami

À la réception de $REQUEST(j, k)$:
if $RN[j] < k$ **then**
  $RN[j] \leftarrow k$
end if
if détenteur du jeton **then**
  if $RN[j] = LN[j] + 1$ **then**
    Ajouter j à la file d'attente
  end if
end if

Nombre de Messages Échangés :

- L'algorithme de Suzuki-Kasami nécessite $N - 1$ messages de requête pour chaque demande d'entrée en section critique.

Limitations de l'Algorithme de Suzuki-Kasami :

- Tolérance aux pannes : La perte du jeton peut bloquer le système. Des mécanismes supplémentaires sont nécessaires pour détecter et régénérer un jeton perdu.
- Charge Réseau : Chaque demande nécessite la diffusion de messages à tous les processus, ce qui peut augmenter la charge réseau.
- Latence : La latence peut être élevée si le jeton doit parcourir de nombreux processus avant d'atteindre le demandeur.

5.2 Algorithmes de Consensus

5.2.1 Algorithme Paxos

Principe :

- Atteindre un consensus entre plusieurs nœuds malgré les pannes et les défaillances du réseau.

Fonctionnement :

Algorithm 16 Algorithme Paxos

Phase 1 : Préparation

Proposeur : Choisir un numéro de proposition n et envoyer un message *prepare(n)* à une majorité d'accepteurs.

Accepteurs : Si n est supérieur à tout autre numéro de proposition pour lequel l'accepteur a déjà répondu, envoyer une réponse *promise(n, valeur)* avec la plus haute valeur acceptée jusqu'à présent.

Phase 2 : Acceptation

Proposeur : Si le proposeur reçoit une réponse *promise* de la majorité des accepteurs, envoyer un message *accept(n, valeur)* avec la valeur la plus élevée reçue, sinon proposer une nouvelle valeur.

Accepteurs : Si n est supérieur ou égal à tout autre numéro de proposition pour lequel l'accepteur a déjà donné sa parole, accepter la proposition et envoyer un message *accepted(n, valeur)*.

Phase 3 : Apprentissage

Si un proposeur reçoit des messages *accepted* de la majorité des accepteurs, il peut décider que la valeur est acceptée.

Les autres nœuds apprennent la valeur acceptée via des messages de notification.

Applications :

- Utilisé dans des systèmes de fichiers distribués, bases de données distribuées et services de verrouillage distribués comme Google Chubby.

5.2.2 Algorithme Raft

Principe :

- Simplifier la compréhension et l'implémentation du consensus par rapport à Paxos.

Fonctionnement :

Algorithm 17 Algorithme Raft

Phase d'Élection
Chaque nœud peut devenir candidat et envoyer des requêtes de vote aux autres nœuds.
Les autres nœuds votent pour le candidat s'ils n'ont pas déjà voté pour un autre candidat au cours de la même période d'élection.
Le candidat qui obtient la majorité des voix devient le leader.

Phase de Réplication de Journal
Le leader accepte les nouvelles entrées de journal des clients.
Le leader réplique les entrées de journal aux suiveurs et attend leur confirmation.
Une fois que la majorité des suiveurs ont confirmé l'entrée, elle est considérée comme validée.
Le leader informe les suiveurs de la validation de l'entrée.

Comparaison avec Paxos :

- Raft est plus simple à comprendre et à implémenter, mais offre des performances similaires.

Applications :

- Utilisé dans des systèmes de gestion de configuration comme Consul et Etcd.

5.3 Conclusion du Chapitre

Les algorithmes de synchronisation assurent la cohérence et la coordination dans les systèmes distribués. Ce chapitre a présenté des algorithmes d'exclusion mutuelle comme Lamport, Ricart-Agrawala, et Suzuki-Kasami, ainsi que des algorithmes de consensus comme Paxos et Raft. Chaque algorithme a ses propres avantages et limitations, et le choix de l'algorithme dépend des besoins spécifiques de l'application. Ces algorithmes sont essentiels pour le bon fonctionnement des systèmes distribués.

Chapter 6

Protocoles de Mise en Œuvre

6.1 Sémaphores

Les sémaphores sont des outils de base pour résoudre les problèmes de synchronisation rencontrés dans la mise en œuvre des systèmes. Ils ont été introduits par Dijkstra pour les systèmes centralisés et sont également adaptés aux systèmes distribués.

6.1.1 Définition

Un sémaphore est une variable partagée, initialisée à une valeur non négative S_0, et manipulée uniquement par deux opérations atomiques :

- P(S) : Attendre que $S > 0$, puis décrémenter S.
- V(S) : Incrémenter S.

6.1.2 Intérêt dans les Systèmes Répartis

Les sémaphores sont utiles dans les systèmes distribués pour :

- Coordination des Accès : Contrôler l'accès concurrentiel aux ressources partagées.
- Prévention des Interblocages : Éviter les situations où plusieurs processus se bloquent mutuellement.
- Synchronisation : Assurer que certaines actions se déroulent dans un ordre précis.

6.1.3 Mise en Œuvre du Sémaphore S

Pour implémenter un sémaphore dans un système réparti, il faut décomposer la valeur de S en deux parties :

- $S_0 + \#V(S)$: Nombre d'autorisations de passage délivrées depuis l'initialisation de S.
- $\#P(S)$: Nombre de demandes de passage délivrées depuis l'initialisation de S.

6.1.4 Variables Locales et Gestion du Jeton

Chaque site i qui peut effectuer une opération P sur le sémaphore S maintient une variable locale $nbvs_i$ pour compter le nombre d'opérations V effectuées sur S.

Opération V

Algorithm 18 Opération V

1: $nbvs_i := nbvs_i + 1$
2: diffuser (incr, $nbvs_i$) à tous les autres sites

Opération P

Le compteur $\#P(S)$ est une variable globale utilisée uniquement par les opérations P. Comme celles-ci doivent être exécutées en section critique, un jeton est utilisé pour implémenter cette section critique et transporter la valeur du compteur $\#P(S)$.

Algorithm 19 Opération P

1: acquérir le jeton (contenant $nbps$)
2: attendre ($nbps < S_0 + nbvs_i$)
3: $nbps := nbps + 1$
4: libérer le jeton (avec $nbps$ mis à jour)

Remarques :

- Le jeton est un message spécial qui contient le compteur global $\boldsymbol{nbps}$.
- La réception de messages $\boldsymbol{incr}$ par les autres sites met à jour leurs copies locales de $\boldsymbol{nbvs_i}$.

6.1.5 Sémaphores Particuliers

Sémaphore Privé

Un sémaphore possédé par un site i, où seul ce site peut exécuter l'opération P(S). Seul le possesseur de S maintient une variable $nbvs_i$, le jeton n'est plus nécessaire et peut être remplacé par une variable locale au site i.

Sémaphore Généralisé

Permet d'opérer de façon atomique sur plusieurs sémaphores en même temps.

Algorithm 20 Sémaphore Généralisé

1: attendre ($S1 - k1 \geq 0$) et ($S2 - k2 \geq 0$)
2: $S1 := S1 - k1$
3: $S2 := S2 - k2$

6.1.6 Protocole de Remise à Zéro des Compteurs

Pour borner la croissance des compteurs $nbvs_i$ et $nbps$, un protocole de remise à zéro peut être utilisé :

Algorithm 21 Protocole de Remise à Zéro des Compteurs

1: Lorsque le possesseur du jeton détecte que $nbps$ a franchi un certain seuil x, il diffuse un message `decr(x)` à tous les sites j qui ont une copie $nbvs_j$.
2: Les sites j effectuent alors $nbvs_j := nbvs_j - x$ et renvoient un acquittement.
3: Après réception de tous les acquittements, le site i repositionne à 0 la valeur $nbps$ transportée par le jeton.

6.2 Protocole de Mise à Jour Retardée

Le protocole de mise à jour retardée est utilisé pour assurer la cohérence des copies de variables dans les systèmes distribués. Il permet de retarder la propagation des mises à jour jusqu'à ce que cela soit nécessaire, réduisant ainsi la charge sur le réseau et améliorant la performance.

6.2.1 Description des Minorants

Les minorants sont des valeurs utilisées pour indiquer l'état des mises à jour d'une variable dans un système réparti. Chaque copie d'une variable est associée à un minorant qui enregistre le numéro de version ou le timestamp de la dernière mise à jour reçue.

Algorithm 22 Initialisation des Minorants

1: $minorant[v] = 0$ pour chaque variable v

Algorithm 23 Mise à jour d'une variable

1: mettre à jour la variable v
2: $minorant[v] = minorant[v] + 1$
3: envoyer mise à jour et $minorant[v]$ à tous les nœuds

Algorithm 24 Réception d'une mise à jour

1: **if** minorant[v] reçu $> minorant[v]$ local **then**
2: appliquer mise à jour
3: $minorant[v]$ local $= minorant[v]$ reçu
4: **end if**

6.2.2 Gestion des Mises à Jour

La gestion des mises à jour dans le protocole de mise à jour retardée implique la propagation des changements de manière contrôlée pour maintenir la cohérence sans surcharger le réseau.

- **Propagation des Mises à Jour** : Les mises à jour sont stockées localement et propagées périodiquement ou sur demande.
- **Application des Mises à Jour** : Lorsqu'une mise à jour est reçue, elle est appliquée seulement si elle est plus récente que la dernière mise à jour connue.

6.2.3 Intérêt dans les Systèmes Répartis

Le protocole de mise à jour retardée est avantageux pour :

- Réduction de la Charge Réseau : Limite la fréquence des communications en regroupant les mises à jour.
- Amélioration de la Performance : Retarde les mises à jour jusqu'à ce qu'elles soient nécessaires, réduisant ainsi le temps de latence.
- Maintien de la Cohérence : Utilise des minorants pour garantir que toutes les copies d'une variable restent cohérentes.

6.3 Conclusion du Chapitre

La gestion des pannes est essentielle pour assurer la fiabilité et la robustesse des systèmes distribués. En comprenant les différents types de pannes et en adaptant les algorithmes de synchronisation et de consensus, les systèmes peuvent continuer à fonctionner correctement même en présence de défaillances. Les stratégies de tolérance aux pannes telles que la réplication, la retransmission des messages, l'utilisation de temporisateurs et les mécanismes de consensus robustes sont cruciales pour maintenir la continuité des services et la cohérence des données.

Chapter 7

Gestion des Pannes

La gestion des pannes est cruciale dans les systèmes distribués pour assurer la continuité des services et la cohérence des données. Les types de pannes et les stratégies de tolérance aux pannes doivent être bien définis pour adapter les algorithmes de synchronisation et garantir la fiabilité du système.

7.1 Types de Pannes

Les pannes peuvent être classifiées en fonction de leur nature et de leur impact sur le système.

7.1.1 Pannes Franches (Crash Failures)

- **Définition** : Un processus cesse de fonctionner et ne reprend jamais son activité.
- **Impact** : Le processus est complètement inactif et ne participe plus au système.
- **Stratégies de Tolérance** :
  - Utilisation de réplication pour maintenir la disponibilité des services.
  - Détection et remplacement des processus défaillants par des mécanismes de surveillance.

7.1.2 Pannes par Omission (Omission Failures)

- **Définition** : Un processus omet d'envoyer ou de recevoir des messages.
- **Impact** : Les messages peuvent être perdus, entraînant des incohérences.
- **Stratégies de Tolérance** :
  - Utilisation de retransmissions pour garantir la livraison des messages.
  - Implémentation de protocoles de détection de perte de messages et de correction.

7.1.3 Pannes de Temporisation (Timing Failures)

- **Définition** : Un processus prend plus de temps que prévu pour compléter une opération.
- **Impact** : Les délais peuvent entraîner des incohérences et des erreurs de synchronisation.
- **Stratégies de Tolérance** :
  - Utilisation de temporisateurs pour détecter les délais anormaux.
  - Implémentation de mécanismes de reprise en cas de dépassement de temps.

7.1.4 Pannes Byzantines (Byzantine Failures)

- **Définition** : Un processus fonctionne de manière incorrecte ou malveillante, envoyant des messages erronés ou contradictoires.
- **Impact** : Les pannes byzantines peuvent compromettre la cohérence et la sécurité du système.
- **Stratégies de Tolérance** :
  - Utilisation de l'algorithme de consensus byzantin (p. ex. : Algorithme de Byzantine Fault Tolerance, BFT).
  - Réplication et vérification croisée entre plusieurs processus pour détecter les comportements anormaux.

7.2 Algorithmes Tolérants aux Pannes

Les algorithmes de synchronisation doivent être adaptés pour gérer les pannes et garantir la fiabilité du système.

7.2.1 Adaptation des Algorithmes de Synchronisation

Les algorithmes de synchronisation doivent inclure des mécanismes pour détecter et gérer les pannes des processus participants.

Adaptation de l'Algorithme de Lamport

Algorithm 25 Détection des Pannes

```
1: à la réception de (absent, HLi, i):
2: marquer le site i comme absent
3: mettre à jour la file d'attente pour ignorer les requêtes de i
```

Algorithm 26 Reprise après Panne

```
1: à la réception de (rentrée, HLi, i):
2: marquer le site i comme actif
3: envoyer le dernier message de requête ou de libération à i
```

Adaptation de l'Algorithme de Ricart-Agrawala

Algorithm 27 Gestion des Messages Perdus

```
1: si réponse non reçue après délai:
2: retransmettre la requête
```

7.2.2 Algorithmes de Consensus Tolérants aux Pannes

Les algorithmes de consensus doivent être robustes pour fonctionner correctement même en présence de pannes.

Algorithme Paxos

- Paxos est conçu pour tolérer les pannes franches et par omission.
- Les processus peuvent proposer des valeurs et les accepteurs peuvent accepter ou rejeter ces propositions.
- Si un leader échoue, un nouveau leader est élu pour continuer le processus de consensus.

Algorithme Raft

- Raft est conçu pour être compréhensible et tolérant aux pannes.
- Les rôles fixes (leader, suiveur, candidat) simplifient la gestion des pannes.
- Le mécanisme d'élection garantit qu'un nouveau leader est rapidement élu en cas de panne du leader actuel.

7.3 Conclusion du Chapitre

La gestion des pannes est essentielle pour assurer la fiabilité et la robustesse des systèmes distribués. En comprenant les différents types de pannes et en adaptant les algorithmes de synchronisation et de consensus, les systèmes peuvent continuer à fonctionner correctement même en présence de défaillances. Les stratégies de tolérance aux pannes telles que la réplication, la retransmission des messages, l'utilisation de temporisateurs et les mécanismes de consensus robustes sont cruciales pour maintenir la continuité des services et la cohérence des données.

Chapter 8

Diffusion

8.1 Diffusion Fiable et Causale

8.1.1 Diffusion Fiable

Principe : Assure que tous les messages envoyés par un processus sont finalement reçus par tous les autres processus.

Propriétés et Garanties :

- **Livraison Fiable** : Si un message est envoyé par un processus, il est éventuellement livré à tous les processus.
- **Aucun Doublon** : Un message n'est jamais livré plus d'une fois à un même processus.
- **Aucune Perte** : Tous les messages envoyés sont reçus par tous les processus non défaillants.

Algorithm 28 Algorithme de Diffusion Fiable

```
1: Envoi d'un message :
2: envoyer(message, destinataire)
3: Accusé de réception :
4: à la réception de (message):
5:     envoyer(ACK, émetteur)
6: Gestion des accusés de réception :
7: si tous les ACKs reçus avant délai:
8:     confirmer la réception
9: sinon:
10:    retransmettre(message)
```

8.1.2 Diffusion Causale

Principe : Respecte l'ordre causal des événements. Si un événement A cause un événement B, alors tous les processus doivent recevoir A avant B.

Propriétés et Garanties :

- **Causalité** : Si un message m_1 est causé par un message m_2, alors m_2 doit être livré avant m_1 à tous les processus.

Algorithm 29 Algorithme de Diffusion Causale avec Horloges Vectorielles

Incrémentation locale :
$V_i[i] = V_i[i] + 1$
Envoi d'un message :
envoyer(message, V_i, destinataire)
Réception d'un message :
à la réception de (message, V_i):
    pour k de 1 à N:
        $V_j[k] = \max(V_j[k], V_i[k])$
    $V_j[j] = V_j[j] + 1$

8.2 Diffusion Totalement Ordonnée

Principe : Assure que tous les messages sont reçus dans le même ordre par tous les processus.

Propriétés et Garanties :

- **Ordre Total** : Tous les processus reçoivent les messages dans le même ordre.

Algorithm 30 Algorithme de Diffusion Totalement Ordonnée (Utilisation de Numéros de Séquence Globaux)

Émission d'un message :
numéro_séquence_global = obtenir_numéro_séquence()
envoyer(message, numéro_séquence_global, destinataire)
Réception et commande des messages :
à la réception de (message, numéro_séquence_global):
    ajouter à la file_d'attente
    trier file_d'attente par numéro_séquence_global

Algorithm 31 Formule pour le Numéro de Séquence

Un serveur de séquence distribue des numéros de séquence uniques et croissants :
numéro_séquence_global = numéro_séquence_global + 1

8.3 Problèmes et Solutions de Diffusion

8.3.1 Problèmes

- **Latence** : Les délais peuvent être significatifs, surtout dans les réseaux larges.
- **Pannes de Réseau** : Les pannes peuvent empêcher la livraison des messages.

- **Ordre des Messages** : Maintenir l'ordre peut être complexe.
- **Scalabilité** : La diffusion à grande échelle peut engendrer une charge importante.

8.3.2 Solutions

Algorithmes de Diffusion Fiable :

- **Bimodal Multicast** : Utilise une combinaison de diffusion épidémique et de retransmissions déterministes.

Algorithm 32 Algorithme de Bimodal Multicast

```
1: diffuser(message, épidémique)
```

Algorithmes de Diffusion Causale :

- **Protocole de Birman-Schiper-Stephenson** : Utilise des horloges vectorielles pour maintenir l'ordre causal.

Algorithm 33 Protocole de Birman-Schiper-Stephenson

```
1: V_i[i] = V_i[i] + 1
2: envoyer(message, V_i, destinataire)
```

Algorithmes de Diffusion Totalement Ordonnée :

- **Algorithme de Totem** : Utilise des anneaux logiques pour ordonner les messages.

Algorithm 34 Algorithme de Totem

```
1: numéro_séquence = obtenir_numéro_séquence()
2: envoyer(message, numéro_séquence, destinataire)
```

Tolérance aux Pannes :

- **Réplication et Retransmission** : Utilise des copies multiples et des retransmissions pour assurer la livraison.

Algorithm 35 Réplication et Retransmission

```
1: si message_non_reçu:
2:     retransmettre(message)
```

8.4 Conclusion du Chapitre

La diffusion fiable, causale et totalement ordonnée est cruciale pour la cohérence et la fiabilité des systèmes distribués. Les algorithmes et formules décrits assurent une gestion efficace de la diffusion, même en présence de pannes et de défis de scalabilité. La compréhension et l'application de ces mécanismes permettent de concevoir des systèmes distribués robustes et performants.

Chapter 9

Comparaison des Algorithmes

9.1 Critères de Comparaison

Pour évaluer et comparer les différents algorithmes de synchronisation et de consensus dans les systèmes distribués, plusieurs critères sont utilisés :

- Complexité :
  - Temporelle : Temps nécessaire pour atteindre un consensus ou une synchronisation.
  - Spatiale : Mémoire nécessaire pour stocker les états et les messages.
- Évolutivité : Capacité de l'algorithme à fonctionner efficacement avec un grand nombre de processus ou de nœuds.
- Tolérance aux Pannes : Capacité de l'algorithme à continuer de fonctionner correctement en présence de défaillances des processus ou du réseau.
- Performance : Latence et bande passante nécessaires pour la communication entre les nœuds.
- Simplicité : Facilité de mise en œuvre et de compréhension de l'algorithme.

9.2 Tableau Comparatif

Le tableau ci-dessous compare plusieurs algorithmes de synchronisation et de consensus en fonction des critères définis.

Algorithme	**Complexité Temporelle**	**Complexité Spatiale**	**Évolutivité**	**Tolérance aux Pannes**	**Performance**	**Simplicité**
Horloge de Lamport	$O(n)$	$O(1)$	Bonne	Faible	Bonne	Simple
Horloges Vectorielles	$O(n)$	$O(n)$	Moyenne	Moyenne	Moyenne	Moyennement simple
Algorithme de Le Lann (LL77)	$O(1)$	$O(1)$	Bonne	Faible	Bonne	Simple
Algorithme de Ricart-Agrawala	$O(n)$	$O(n)$	Bonne	Moyenne	Moyenne	Moyennement simple
Algorithme de Suzuki-Kasami	$O(n)$	$O(n)$	Moyenne	Faible	Moyenne	Moyennement simple
Paxos	$O(n^2)$	$O(n)$	Bonne	Excellente	Moyenne	Complexe
Raft	$O(n)$	$O(n)$	Bonne	Excellente	Bonne	Moyennement simple

Table 9.1: Comparaison des Algorithmes de Synchronisation et de Consensus

9.3 Détails des Algorithmes

9.3.1 Horloge de Lamport

- **Complexité** : Linéaire par rapport au nombre de processus.
- **Tolérance aux Pannes** : Faible, ne gère pas les pannes de manière efficace.
- **Usage** : Simple à implémenter pour ordonner les événements.

9.3.2 Horloges Vectorielles

- **Complexité Spatiale** : Augmente avec le nombre de processus.
- **Tolérance aux Pannes** : Moyenne, les processus peuvent détecter les messages manquants.
- **Usage** : Utilisées pour maintenir l'ordre causal.

9.3.3 Algorithme de Le Lann (LL77)

- **Complexité Temporelle et Spatiale** : Constante.
- **Tolérance aux Pannes** : Faible, nécessite un nouveau jeton en cas de perte.
- **Usage** : Simple et efficace pour un nombre limité de nœuds.

9.3.4 Algorithme de Ricart-Agrawala

- **Complexité** : Linéaire par rapport au nombre de processus.
- **Tolérance aux Pannes** : Moyenne, gère les retransmissions mais pas les défaillances totales.
- **Usage** : Réduit le nombre de messages par rapport à Lamport.

9.3.5 Algorithme de Suzuki-Kasami

- **Complexité** : Linéaire par rapport au nombre de processus.
- **Tolérance aux Pannes** : Faible, dépend de la présence du jeton.
- **Usage** : Efficace pour des systèmes de taille modérée.

9.3.6 Paxos

- **Complexité** : Quadratique en temps, linéaire en espace.
- **Tolérance aux Pannes** : Excellente, conçu pour résister aux pannes de nœuds.
- **Usage** : Algorithme de consensus robuste, utilisé dans les systèmes critiques.

9.3.7 Raft

- **Complexité** : Linéaire.
- **Tolérance aux Pannes** : Excellente, facile à comprendre et à implémenter.
- **Usage** : Alternative à Paxos avec des concepts simplifiés pour la tolérance aux pannes.

9.4 Conclusion du chapitre

Le choix de l'algorithme de synchronisation ou de consensus dépend des besoins spécifiques du système distribué, tels que la tolérance aux pannes, la complexité, l'évolutivité, et la performance requise. Les horloges logiques et les algorithmes basés sur les tokens sont souvent plus simples à mettre en œuvre, tandis que les algorithmes de consensus comme Paxos et Raft offrent une tolérance aux pannes et une robustesse supérieures, bien qu'ils soient plus complexes.

Chapter 10

Études de Cas et Applications

10.1 Exemples Pratiques

Les algorithmes de synchronisation et de consensus sont utilisés dans de nombreux systèmes distribués pour assurer la cohérence des données et la coordination des processus. Voici quelques exemples réels d'utilisation de ces algorithmes.

Exemple	**Description**	**Algorithme Utilisé**	**Application**
Google Chubby	Service de verrouillage distribué utilisé pour la coordination dans les infrastructures de Google.	Paxos	Coordination des accès aux ressources partagées, bases de données et systèmes de fichiers.
Apache ZooKeeper	Service de coordination distribué pour les applications distribuées.	Zab (ZooKeeper Atomic Broadcast)	Coordination des tâches distribuées, configuration de services, gestion de métadonnées.
Raft dans Consul et Etcd	Systèmes de gestion de configuration et de découverte de services.	Raft	Stockage et distribution des configurations dynamiques, orchestration des microservices.

Table 10.1: Exemples Pratiques d'Utilisation des Algorithmes de Synchronisation et de Consensus

10.2 Analyse des Performances

Exemple	**Performance**	**Complexité**	**Résultats**
Google Chubby	Tolérance aux Pannes : Excellente	Complexité élevée, justifiée par la robustesse et la fiabilité du service.	Coordination efficace des ressources avec haute disponibilité et faible latence.
Apache ZooKeeper	Tolérance aux Pannes : Bonne	Capable de gérer des centaines de clients simultanément.	Coordination efficace des tâches, gestion de la configuration avec forte cohérence des données.
Raft dans Consul et Etcd	Tolérance aux Pannes : Excellente	Plus facile à comprendre et à implémenter par rapport à Paxos, avec des performances similaires.	Gestion de configuration dynamique et découverte de services fiable, avec forte cohérence et faible latence.

Table 10.2: Analyse des Performances des Algorithmes de Synchronisation et de Consensus

10.3 Conlusion du chapitre

Les algorithmes de synchronisation et de consensus jouent un rôle crucial dans les systèmes distribués modernes. Les études de cas montrent que des solutions telles que Google Chubby, Apache ZooKeeper, Consul et Etcd utilisent des algorithmes comme Paxos, Zab, et Raft pour assurer la cohérence, la tolérance aux pannes, et la performance des systèmes distribués. Ces exemples pratiques illustrent l'importance de choisir le bon algorithme en fonction des exigences spécifiques de l'application et des contraintes du système.

Chapter 11

Avantages et Inconvénients

11.1 Analyse Critique

Algorithme	**Avantages**	**Inconvénients**
Horloge de Lamport	Simplicité, Facile à implémenter	Tolérance aux pannes faible, Scalabilité limitée
Horloges Vectorielles	Précision de l'ordre causal, Peut détecter les messages manquants	Complexité spatiale élevée, Plus complexe à implémenter
Algorithme de Le Lann (LL77)	Simplicité, Faible latence pour l'accès à la section critique	Tolérance aux pannes faible, Scalabilité limitée
Algorithme de Ricart-Agrawala	Réduction du nombre de messages, Maintient un ordre total des requêtes	Complexité temporelle élevée, Tolérance aux pannes moyenne
Algorithme de Suzuki-Kasami	Simplicité, Utilisation d'un unique jeton pour la synchronisation	Tolérance aux pannes faible, Scalabilité limitée
Paxos	Excellente tolérance aux pannes, Robuste	Complexité élevée, Latence et overhead élevés
Raft	Facilité de compréhension et de mise en œuvre, Excellente tolérance aux pannes	Latence potentiellement élevée sous haute charge, Peut nécessiter des ajustements pour la scalabilité

Table 11.1: Comparaison des Avantages et Inconvénients des Algorithmes de Synchronisation et de Consensus

11.2 Recommandations

- **Horloge de Lamport** : Utiliser pour des systèmes simples avec des exigences minimales en matière de tolérance aux pannes et de scalabilité.
- **Horloges Vectorielles** : Préférer pour des systèmes nécessitant un suivi précis de l'ordre causal des événements.
- **Algorithme de Le Lann (LL77)** : Adapté pour des systèmes de petite taille avec des exigences simples de synchronisation.
- **Algorithme de Ricart-Agrawala** : Utiliser lorsque la réduction du nombre de messages est importante et que la tolérance aux pannes est moyennement critique.
- **Algorithme de Suzuki-Kasami** : Idéal pour des systèmes de taille modérée où la simplicité et l'efficacité sont importantes.
- **Paxos** : Choisir pour des systèmes critiques nécessitant une tolérance élevée aux pannes et une forte robustesse.
- **Raft** : Préférer pour des systèmes où la simplicité de mise en œuvre est importante, tout en nécessitant une tolérance élevée aux pannes.

11.3 Conclusion du chapitre

En conclusion, chaque algorithme de synchronisation et de consensus présente ses propres avantages et inconvénients. Le choix de l'algorithme le plus approprié dépend des exigences spécifiques du système distribué, telles que la tolérance aux pannes, la complexité, l'évolutivité et la performance requise. Les algorithmes plus simples comme l'horloge de Lamport et l'algorithme de Le Lann (LL77) sont faciles à implémenter mais offrent une tolérance aux pannes limitée. En revanche, des algorithmes comme Paxos et Raft offrent une robustesse et une tolérance aux pannes excellentes, mais au prix d'une complexité et d'une latence plus élevées. Les recommandations fournies aident à orienter le choix en fonction des besoins spécifiques du système.

Chapter 12

Perspectives et Travaux Futurs

12.1 Défis Actuels

- **Scalabilité** : Les algorithmes actuels doivent être améliorés pour gérer efficacement un grand nombre de processus sans compromettre la performance.
- **Tolérance aux Pannes Byzantines** : Les systèmes doivent être capables de gérer les pannes byzantines de manière plus efficace et sécurisée.
- **Complexité** : La réduction de la complexité des algorithmes de consensus comme Paxos reste un défi majeur.
- **Latence** : La minimisation de la latence dans les algorithmes de synchronisation et de consensus est cruciale pour les applications en temps réel.

12.2 Directions Futures

- **Algorithmes Hybrides** : Développer des algorithmes hybrides qui combinent les avantages des algorithmes existants pour améliorer la tolérance aux pannes, la scalabilité, et la performance.
- **Machine Learning pour la Synchronisation** : Utiliser des techniques de machine learning pour prédire et gérer les pannes, optimiser les chemins de diffusion, et améliorer la performance globale.
- **Blockchain et Consensus Distribué** : Explorer l'utilisation des technologies de blockchain pour la synchronisation et le consensus dans les systèmes distribués.
- **Optimisation de la Consommation d'Énergie** : Développer des algorithmes de synchronisation et de consensus qui minimisent la consommation d'énergie, particulièrement pour les systèmes IoT (Internet of Things).
- **Sécurité et Confidentialité** : Intégrer des mécanismes de sécurité et de confidentialité plus robustes dans les algorithmes de synchronisation pour protéger les données sensibles.

12.3 Conclusion du chapitre

Les perspectives pour les algorithmes de synchronisation et de consensus sont prometteuses, mais de nombreux défis restent à relever. La scalabilité, la tolérance aux pannes byzantines, la complexité et la latence sont des domaines nécessitant des améliorations continues. Les directions futures incluent le développement d'algorithmes hybrides, l'intégration du machine learning, l'exploration des technologies de blockchain et l'optimisation de la consommation d'énergie. La sécurité et la confidentialité des données restent également des priorités majeures. Ces avancées contribueront à rendre les systèmes distribués plus robustes, performants et sécurisés.

Chapter 13

Conclusion

13.1 Résumé des Points Clés

Cette étude a exploré en détail les principaux algorithmes de synchronisation répartie. Nous avons analysé les horloges logiques, vectorielles et matricielles, et leurs rôles dans le maintien de l'ordre des événements et la vérification de la causalité. Les algorithmes utilisant des jetons tels que Ricart-Agrawala, Suzuki-Kasami, et le jeton circulaire ont été étudiés pour leur efficacité dans la gestion de l'exclusion mutuelle. Les protocoles de consensus, y compris Paxos et Raft, ont été examinés pour leur capacité à parvenir à un accord entre plusieurs nœuds. Une comparaison détaillée a été effectuée pour évaluer ces algorithmes en termes de complexité, évolutivité, tolérance aux pannes et applications pratiques.

13.2 Impact de la Synchronisation Répartie

La synchronisation répartie est essentielle dans les systèmes distribués modernes pour garantir la cohérence des données et coordonner efficacement les processus. Elle est cruciale dans divers domaines, notamment les bases de données distribuées, les systèmes de fichiers, les réseaux de capteurs et les systèmes de contrôle industriels. L'utilisation de protocoles robustes de synchronisation améliore la résilience et la performance des systèmes distribués, tout en facilitant leur évolution et leur maintenance. Cette étude met en lumière l'importance de choisir le bon algorithme de synchronisation en fonction des exigences spécifiques du système et des contraintes opérationnelles.

Bibliography

[1] L. Lamport, *Time, Clocks, and the Ordering of Events in a Distributed System*, Communications of the ACM, vol. 21, no. 7, pp. 558-565, 1978.

[2] F. Mattern, *Virtual Time and Global States of Distributed Systems*, in Proceedings of the Workshop on Parallel and Distributed Algorithms, pp. 215-226, 1989.

[3] C. J. Fidge, *Timestamps in Message-Passing Systems that Preserve the Partial Ordering*, Australian Computer Science Communications, vol. 10, no. 1, pp. 56-66, 1988.

[4] G. Ricart and A. K. Agrawala, *An Optimal Algorithm for Mutual Exclusion in Computer Networks*, Communications of the ACM, vol. 24, no. 1, pp. 9-17, 1981.

[5] I. Suzuki and T. Kasami, *A Distributed Mutual Exclusion Algorithm*, ACM Transactions on Computer Systems, vol. 3, no. 4, pp. 344-349, 1985.

[6] A. Tanenbaum and M. van Steen, *Distributed Systems: Principles and Paradigms*, Prentice Hall, 2007.

[7] L. Lamport, *Paxos Made Simple*, ACM SIGACT News, vol. 32, no. 4, pp. 18-25, 2001.

[8] D. Ongaro and J. Ousterhout, *In Search of an Understandable Consensus Algorithm*, in Proceedings of the USENIX Annual Technical Conference, pp. 305-319, 2014.

[9] N. Lynch, *Distributed Algorithms*, Morgan Kaufmann Publishers, 1996.

[10] T. Chandra and S. Toueg, *Unreliable Failure Detectors for Reliable Distributed Systems*, Journal of the ACM, vol. 43, no. 2, pp. 225-267, 1996.