AVANT-PROPOS

Ce polycopié de cours, conforme au programme enseigné, s'adresse aux étudiants 1ère année Master RISR (Réseaux Informatiques et Systèmes Répartis) de l'Université de Saida « Dr Tahar Moulay ». il s'adresse plus généralement à toute personne désireuse d'apprendre et approfondir les connaissances dans le domaine des systèmes distribués.

On y étudiera les paradigmes de l'algorithmique répartie tout en traitant en détails les algorithmes distribués nécessaires à la gestion d'un système réparti. Dans l'objectif d'apprendre à aborder de manière rigoureuse les problèmes de distribution et leurs solutions.

Ce polycopié s'articule autour de six chapitres :
Le premier chapitre étudie les notions fondamentales des systèmes répartis.
Le deuxième chapitre est consacré aux temps et état dans un système distribué.
Le troisième chapitre est abordé les différents algorithmes d'exclusion mmutuelle réparti.
Le quatrième chapitre pésente les algorithmes d'eléction de leader dans un système réparti.
Le cinquième chapitre porte sur la terminaison réparti et ses algorithmes.
Le dernier chapitre est réservé aux consensus réparti.

Table des matières

Chapitre 1

Introduction aux Systèmes distribués

1.1. Les progrès technologiques

L'histoire moderne des ordinateurs correspond avec l'invention des programmes stockés en mémoire.

- **La première génération -1950-1959 :**La première génération est caractérisée par l'utilisation des tubes à vide comme éléments actifs. Ces ordinateurs étaient dotés du stockage électrostatique, des tubes de William, des tambours magnétiques, des ferrites.
  Exemple : L'UNIVAC était le premier ordinateur produit en série et vendu à un client et utiliser pour faire du comptages, des statistiques et d'autre travaux relevant du traitement de l'information. La vitesse était de 2.25 MHz [1].

- **La Deuxième Génération 1959-1963 :**La deuxième génération est caractérisée par l'utilisation des transistors et les mémoires à tore de ferrites. Cette génération verra l'utilisation des langages de programmations évolués comme le Fortran et le Cobol, même si le langage assembleur était toujours présent. Le remplacement des tubes par les transistors avait pour objectif de résoudre le problème de fiabilité, de dissipation thermique et de consommation énergétique.
  Exemple :IBM 1401 introduit en 1960 [1].

- **La Troisième Génération 1964-1971 :** la série de machine de cette génération n'a pas utilisé au départ les circuits intégrés, mais des cartes modulaires d'éléments discrets sur un substrat de céramique. IBM a voulu utiliser des circuits intégrés, mais il a finalement opté pour fabriquer en grande quantité, une série de 6 machines compatibles [1].

- **Le quatrième génération : de 1971 à 1990 :** La caractéristique principale de la quatrième génération est l'intégration à grande échelle des circuits qui donnés naissance aux microprocesseurs. Il est désormais possible de mettre plusieurs millions d'équivalent transistor sur un seul circuit intégré. Cette génération est caractérisé essentiellement par la révolution micro-informatique.
  Exemple :Cray-1 (1975), Cray X-MP (1982) contient 2 à 4 processeur, Cray-2 (1983) contient 8 processeurs [1].

- **La cinquième génération de 1990 à 2000 :** fort retrait des supercalculateurs entre 1990 et 1995 dû aux nombreuses faillites et disparition des architectures originales.
  Ce retrait était le resultat du manque de réalisme, faible demande en supercalculateurs, coût d'achat et d'exploitation trop élevés, une utilisation peu

pratique, systèmes d'exploitation propriétaires, difficulté d'apprentissage, Manque ou absence d'outils et difficulté d'exploitation [1].

- **Vers 2000 : Réseaux d'ordinateurs à large échelle (Grilles)** avec les grilles informatiques qui se développent à partir de la fin des années 90, surtout dans la communauté scientifique, on n'est plus très loin du service public de calcul. Le terme grille provient de l'analogie avec le réseau électrique qu'on appellegrid aux états-Unis. La puissance de calcul et de stockage est fournie par des centres de calcul et de données qui sont de différents types et situés dans différents endroits. Et l'utilisateur accède à ces ressources à traversle réseau Internet. Dans cette vision, les terminaux d'accès ont été remplacés par les ordinateurs personnels et les ressources informatiques, auxquels un utilisateur peut accéder, peuvent être situées dans un ou plusieurs centres de calcul qui ont conclu des accords de mutualisation des ressources [1].

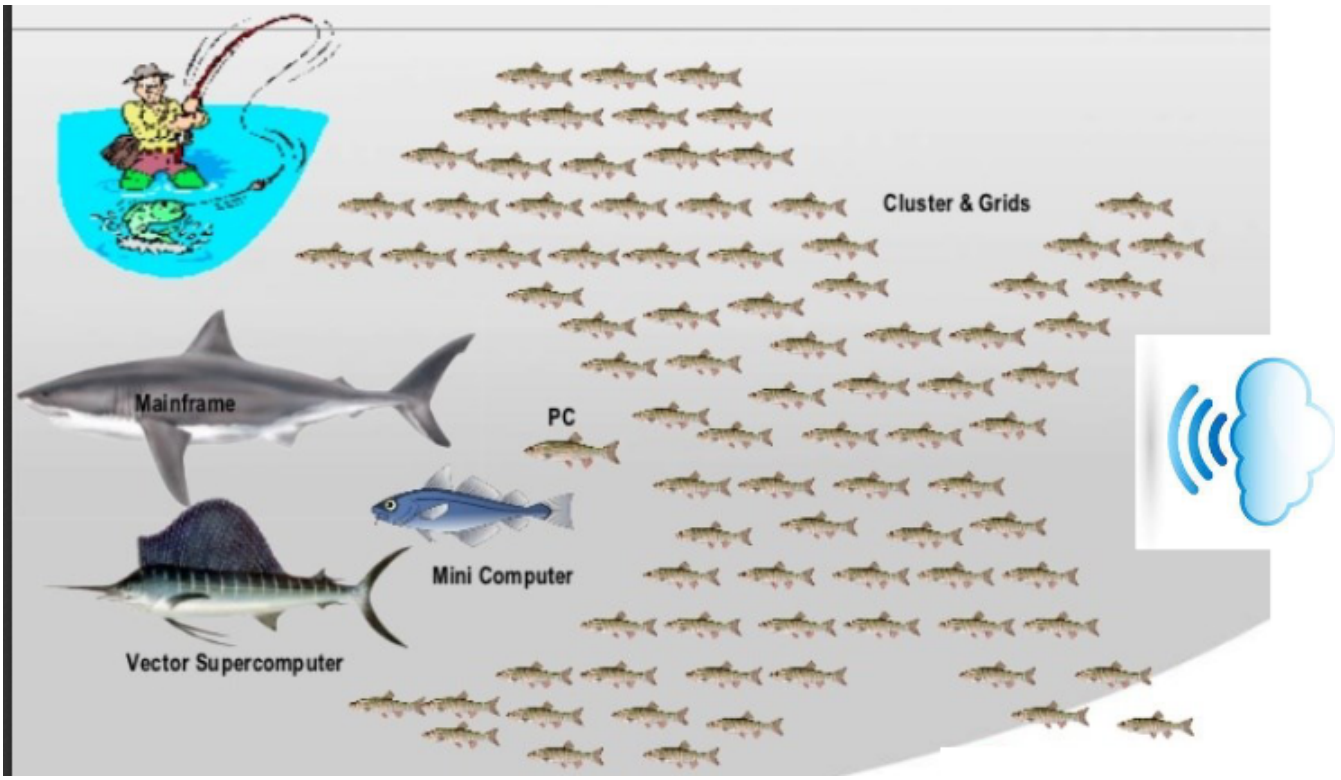


FIGURE 1.1 – *Evolution du materiel informatique [1].*

- **Emergence de nuage informatique (Cloud) :** c'est finalement dans le

monde commercial que sont nés les nuages informatiques. Le terme nuage informatique ou cloud est apparu au milieu des années 2000, très probablement en 2006 dans la bouche d'éric Schmidt lors d'une conférence. Ce terme a immédiatement été repris par Amazon dans le nom de son service EC2, pour Elastic Compute Cloud. Avec les nuages informatiques, vous ne payez que pour ce que vous utilisez.
Depuis lors, le marché du cloud computing est foisonnant avec de très nombreux acteurs, petits et grands, qui proposent différents types de services à une clientèle très variée, qui va du commun des mortels jusqu'aux entreprises et autres institutions en passant par la communauté scientifique [1].

1.2. Définitions d'un Système distribué

L'informatique distribué renvoie à l'idée même de l'utilisation des systèmes distribués qui sont généralement plusieurs ordinateurs reliés entre eux par des réseaux informatiques pour traiter en collaboration un objectif commun. Diverses définitions légèrement différentes existent, mais contentons-nous de ce qui suit :

- Selon **Leslie Lamport 28 Mai, 1987** : " You know you have a distributed system when the crash of a computer you've never heard of stops you from getting any work done ".
  Un système distribué est un système qui vous empêche de travailler quand une machine dont vous n'avez jamais entendu parler tombe en panne. On peux déduire que lamport se focalise sur la tolérance aux pannes des systèmes distribués.

- Selon **M. Singhaland & N. Shivaratri 1994** [4] : "A collection of computers that do not share common memory or a common physical clock, that communicate by a messages passing over a communication network, and where each computer has its own memory and runs its own operating system. Typically the computers are semi-autonomous and are loosely coupled while they cooperate to address a problem collectively".
  Ensemble d'ordinateurs qui ne partagent pas une mémoire commune ou une horloge physique commune, qui communiquent par un message passant sur un réseau de communication, et où chaque ordinateur a sa propre mémoire et exécute son propre système d'exploitation. En règle générale, les ordinateurs sont semi-autonomes et sont faiblement couplés pendant qu'ils coopèrent pour résoudre un problème collectivement. On peu dire qu'ils ont concentré sur le type de mémoire (mémoire distribuée), Absence d'une horloge physique et la communication s'effectue par envoie de message.

- Selon **A. Tanenbaumand & M. Van Steen 2001** [5] : "A collection of independent computers that appears to the users of the system as a single coherent computer".
  Un ensemble d'ordinateurs indépendants qui apparait aux utilisateurs du système comme un seul ordinateur cohérent. Cette définition s'intéresse à la transparence par rapport aux utilisateurs.

- **Tentative de définition :** " Un système distribué est un ensemble d'entités autonomes de calcul (ordinateurs, organes d'E/S, mémoires, processeurs spécialisés, dispositif de commande ou de mesure, ...etc) interconnectées en réseaux et équipé d'un logiciel (Middleware=intergiciel) qui permet aux différents composants du système de coordonner leurs activités d'une telle façon que l'utilsateurs perçoivent le système comme une capacité de traitement unique, cohérente et intégrée".

1.3. Propriétés d'un Système distribué[2]

1. Non unicité des lieux (séparation géographique) :Il n'est pas nécessaire que les processeurs se trouvent dans le même WAN
2. Absence d'horloge physique commune : asynchronisation entre processeurs.
3. Absence de mémoire commune (partagée : communication par message) : Les systèmes distribués peuvent fournir une abstraction concernant un espace d'adressage commun via une mémoire partagée distribuée.
4. Autonomie et hétérogénéité (différentes caractéristiques matérielles et logicielles) : Les processeurs sont faiblement couplés. Ils ont des vitesses différentes et ils utilisent des systèmes d'exploitation différents, ils n'appartiennent pas à un système dédié mais coopèrent pour résoudre un problème

1.4. Caractéristiques d'un Système distribué [2]

1. **Hétérogénéité :**
   - Hétérogénéité des ressources de calcul
   - Hétérogénéité des réseaux de communication
   - Hétérogénéité des systèmes & applications

2. **Ouverture :** il est important que le système distribué soit flexible pour laisser ouverte toutes les options d'extension et d'amélioration du système.

3. **Tolérance aux pannes :** dans un système réparti la probabilité d'une défaillance partielle ne peut être négligée.

4. **Transparence :** le but est de réaliser un système distribué à image unique qui donne l'illusion à l'utilisateur que l'ensemble de machines ne constituent qu'un seul système.
   Le concept transparence est appliqué à plusieurs aspects dans un système distribué :

   - Transparence d'accès : uniformité d'accès locaux et distants.
   - Transparence de localistion : localisation des ressources non perceptible, le nom de la ressource ne doit pas englober sa localisation.
   - Transparence de migration : la migration est possible sans interférence avec la localisation physique
   - Transparence à la réplication : le nombre de copies de ressources est invisible pour l'utilisateur.
   - Transparence à la concurrence : accès partagé aux ressources non perceptible.
   - Transparence au parallelisme : invisibilité des activités parallèles offertes par l'environnement d'exécution.

5. **Sécurité :** vise à garantir le maintien de la confidentialité et de l'intégrité de l'information et le respect des règles d'accès aux services.

6. **Scalabilité :** c'est la capacité du système à s'adapter à l'augmentation de charge des ressources.

7. **Fiabilité & performance :**

   - Disponibilité : fraction de temps pendant laquelle le système est resté utilisable.
   - Cohérence : garantir des informations valides quelque soient les évènements qui peuvent survenir.
   - Efficacité : temps de réponse.
   - Robustesse : détection des failles et reprise.

1.5. Organisation d'un Système distribué [2]

1.5.1. Organisation matérielle :

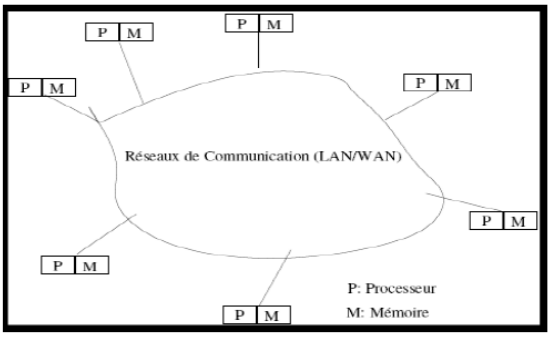


Figure 1.2 – *Organisation matérielle [2].*

1.5.2. Organisation logicielle :

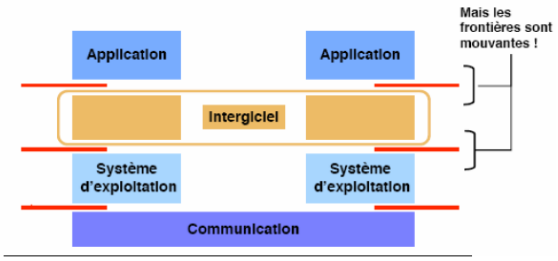


Figure 1.3 – *Organisation logicielle [2].*

1.6. Avantages et Inconvenients d'un Système distribué [2]

1.6.1. Avantages

- Aspect économiques :
- Adaptation aux applications distribuées :
- Partage de ressources & accès distants :
- Fiabilité accrue :
- Accélération des calculs :
- Passage à l'échelle :

1.6.2. Inconvénients

- Insuffisance de software.
- Problème causé par les réseaux.
- Problème de sécurité.

1.7. Architecture d'un Système distribué[3]

Le calcul parallèle est un calcul dans lequel les travaux sont divisés en parties discrètes qui peuvent être exécutées simultanément. Chaque partie est ensuite décomposée en une série d'instructions. Les instructions de chaque partie s'exécutent simultanément sur différentes CPU. Les systèmes parallèles traitent de l'utilisation simultanée de plusieurs ressources informatiques pouvant inclure un seul ordinateur avec plusieurs processeurs, un certain nombre d'ordinateurs connectés par un réseau pour former un cluster de traitement parallèle ou une combinaison des deux. Les systèmes parallèles sont plus difficiles à programmer que les ordinateurs avec un seul processeur car l'architecture des ordinateurs parallèles varie en conséquence et les processus de plusieurs processeurs doivent être coordonnés et synchronisés.
Les processeurs sont au coeur du traitement parallèle. En fonction du nombre d'instructions et de flux de données pouvant être traités simultanément, les systèmes informatiques sont classés en quatre grandes catégories parallèle (Taxonomie de Flynn-Tanenbaum) :

1.8. Taxonomie de FLYNN [3]

Cette classification est basée sur les notions de flot de contrôle (deux premières lettres, I voulant dire Instruction) et flot de données (deux dernières lettres, D voulant dire "Data").

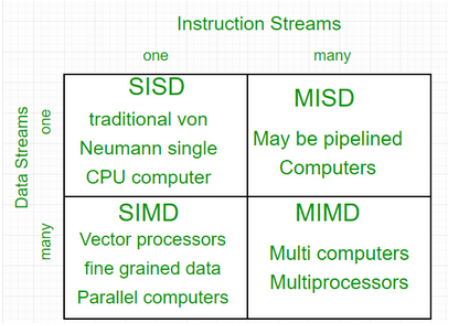


Figure 1.4 – *Taxonomie de FLYNN [3].*

- **Single Instruction Single Data (SISD)** : Un système informatique SISD est une machine monoprocesseur qui est capable d'exécuter une seule instruction, fonctionnant sur un seul flux de données. Dans le SISD, les instructions machine sont traitées de manière séquentielle et les ordinateurs adoptant ce modèle sont communément appelés ordinateurs séquentiels. La plupart des ordinateurs conventionnels ont une architecture SISD. Toutes les instructions et données à traiter doivent être stockées dans la mémoire primaire.

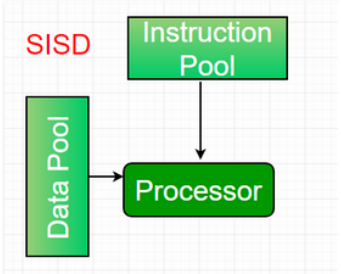


Figure 1.5 – *SISD [3].*

- **Single-instruction, multiple-data (SIMD)** :

  Un système SIMD est une machine multiprocesseur capable d'exécuter la même instruction sur toutes les CPU mais fonctionnant sur des flux de données différents. Les machines basées sur un modèle SIMD sont bien adaptées au calcul scientifique car elles impliquent de nombreuses opérations vectorielles et matricielles. Afin que les informations puissent être transmises à tous les éléments de traitement (PE), les éléments de données organisés des vecteurs peuvent être divisés en plusieurs ensembles (N-ensembles pour N systèmes PE) et chaque PE peut traiter un ensemble de données.

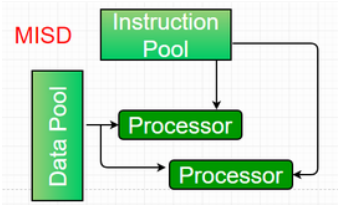


FIGURE 1.6 – *SIMD [3].*

- **Multiple-instruction, single-data (MISD)** :

  Un système informatique MISD est une machine multiprocesseur capable d'exécuter différentes instructions sur différents PE mais toutes fonctionnant sur le même jeu de données.

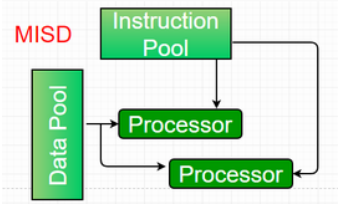


FIGURE 1.7 – *MISD [3].*

- **Multiple-instruction, multiple-data (MIMD)** : Un système MIMD est une machine multiprocesseur capable d'exécuter plusieurs instructions sur plusieurs ensembles de données. Chaque PE du modèle MIMD possède des instructions et des flux de données séparés ; par conséquent, les machines construites en utilisant ce modèle sont capables de tout type d'application. Contrairement aux machines SIMD et MISD, les PE des machines MIMD fonctionnent de manière asynchrone (voir Figure 1.8).

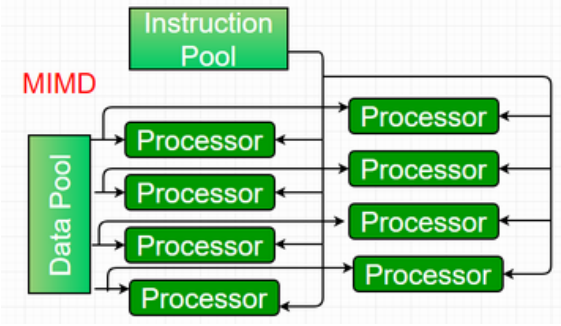


FIGURE 1.8 – *MIMD [3].*

1.8.1. INSUFFISANCES DE CLASSIFICATION DE FLYNN

- Absence de l'organisation de la mémoire (Partagée, distribué ou hybride).
- Réseaux d'interconnexion (type d'interconnexion ?).

Chapitre 2

Temps et état dans un système distribué

2.1. Modèle évènementielle d'un calcul réparti

2.1.1. Programme distribué

Un programme distribué est composé d'un ensemble de processus $\{P_1, P_2, ...\}$ asynchrones communiquant par envoie de messages à travers un réseau et ne partagent pas de mémoire commune.

2.1.2. Modèle d'exécution répartie

On définit quelques concepts d'une exécution répartie :

- **Exécution d'un processus :** est une exécution séquentielle d'une succession d'évènement chacun d'eux se produisant sur un site.
- **Evènement :** est un calcul local sur un site (interne), emission d'un message ou réception d'un message
- **Exécution d'un évènement est atomique :** toutes les opérations qu'il contient doivent être exécutées avant de passé à l'évènement suivant

2.1.3. Principe de base

- e_i^x : le x^{ieme} évènement du processus P_i ;
- Send(m) et Rec(m) : emission et réception de message ;
- Les évènements produits par un même processus sont totalement ordonnés ;
- Si deux évènements se produisent sur des processus distincts, on ne peux pas dire que l'un d'eux a eu lieu avant l'autre.
- Il est bien évident que si un des évènement contient l'émission d'un message et que l'autre contient la réception du même message, alors le premier a eu lieu avant le second. Ceci fournit la base d'une relation d'ordre partiel sur les évènements, appelé **relation de causalité message.**

2.1.4. Modélisation

L'évolution d'une exécution répartie est modélisée par un diagramme temporel (voir Figure 2.1) :

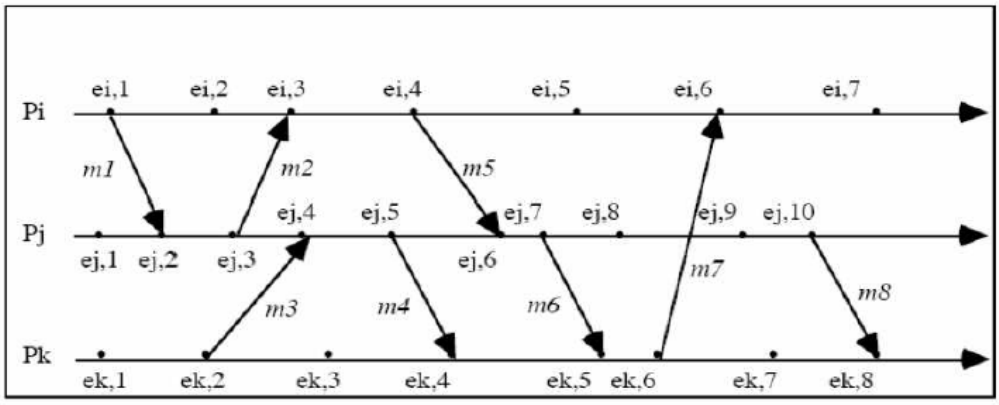


FIGURE 2.1 – *Modélisation d'une exécution répartie.*

2.1.5. Relation de précédence causale (→)

Un évènement **e** → un évènement **f** ⇔

1. Ou bien **e** et **f** se dérooule sur le même processus dans cet ordre ;
2. Ou bien **e** contient l'émission d'un message **m** et **f** contient la réception du même message ;
3. **Transitivité** : **e** → **f** et **f** → **g** alors **e** → **g**.

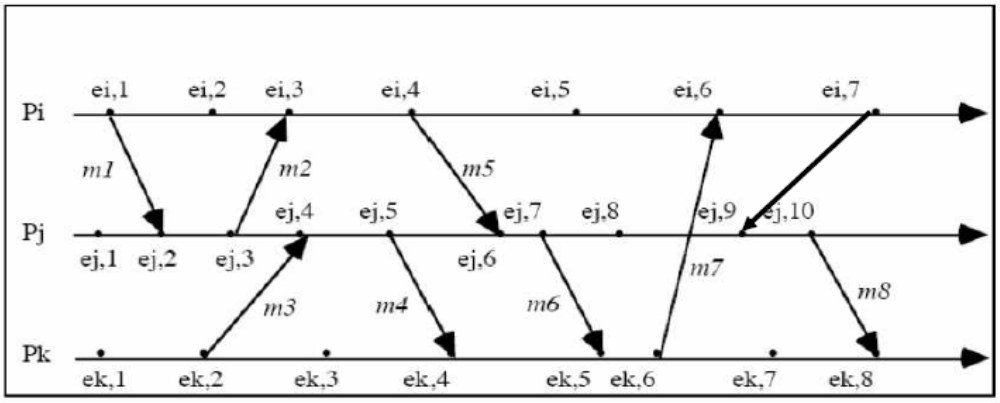
Remarques :

- **e Non** → **f** signifie que f ne dépend pas de e ni directement ni par transitivité. Exemple **ei,5 Non** → **ej,8**
- Deux évènements non comparables sont dits concurrents (concomitants) ou causalement indépendants : e // f ⇔ Non (**e** → **f**) et Non (**f** → **e**).

2.1.6. Modèles de communications

Il existe trois principaux modèles de communications :

1. **FIFO** : l'ordre de remise des message préserve l'ordre d'émission de message par la source ;
2. **No FIFO** : le canal se comporte comme un ensemble où l'émetteur lui ajoute des messages et le récepteur lui soustrait dans un ordre aléatoire.



ei,1 → ei,2→ ei,3 → ei,4→ ej,6 → ej,7→ ek,5 ;
ej,6 → ek,7 ;
ei,1 → ek,8

FIGURE 2.2 – *Exemple de la relation de précédence.*

3. **Ordre Causal** : basé sur la relation de précédence :
   pour tout message m_{ij} et m_{kj} : si **send**(m_{ij}) → **send**(m_{kj}) alors **rec**(m_{ij}) → **rec**(m_{kj}).
   Ce type de modèle est très utilisé dans la conception d'algorithme réparti.

2.1.7. HISTOIRE D'UN ÉVÈNEMENT,COUPURE & FRONTIÈRE

- **Histoire d'un évènement :** pour tout évènement e, le passé (histoire) de e est l'ensemble des évènements qui lui sont inférieurs par la relation d'ordre de précédence causale.
  HIST(e)=e ∪ ensemble des évènements é tels que **é** → **e**

- **Notion de Coupure :** c'est une photographie instantanée d'un système obtenue en prenant un évènement par site et tout les évènements du site qui le précèdent [6].
  C=sous ensemble fini de H tel que pour e,f ∈ H.
  Formulation : f ∈ C et (e → f) ⇒ e ∈ C.

  Chaque coupure correspond à un état global, on distinct deux types de coupure [6] :

  1. Coupure cohérente : est une Coupure qui respecte les dépendances causales des événements du système(aucun message ne vient du futur).

Passé de e13 = **e11 e12 e13 e21 e31 e32 e33**

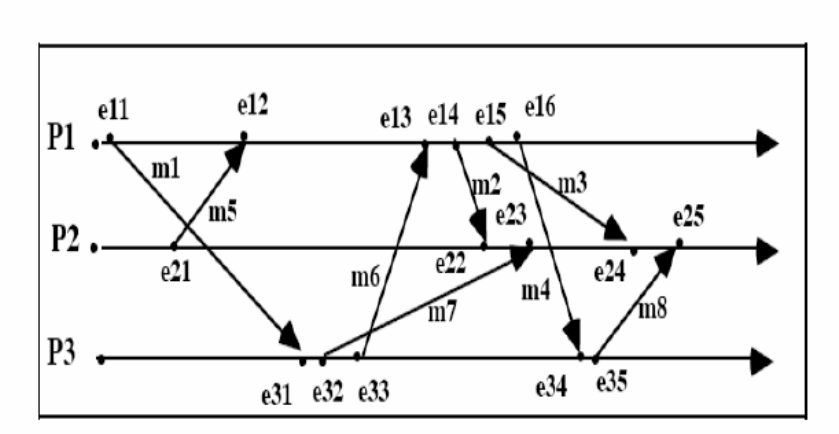


FIGURE 2.3 – *Exemple d'histoire d'un évènement.*

2. Coupure incohérente : est une Coupure qui ne respecte pas les dépendances causales des évènements du système (il existe au moin un message qui traverse la coupure du futur vers le passé) [3].

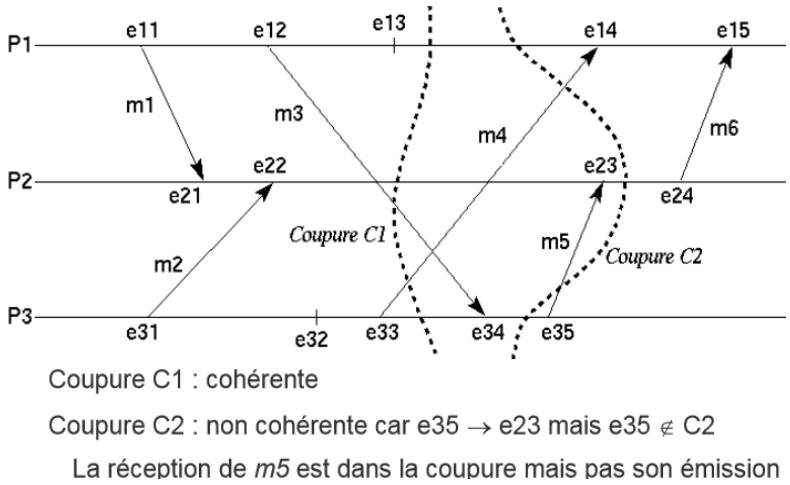


FIGURE 2.4 – *Exemple d'une coupure cohérente et incohérente.*

- **Frontière** : frontière d'un F d'une coupure C est l'ensemble des évènements les plus récents de la coupure, un par site [5].
  Formulation : e ∈ F ⇔ e ∈ C et il n'existe pas è ∈ C tel que e → è.

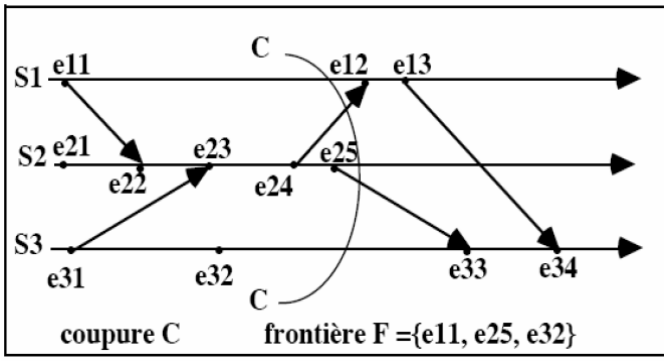


FIGURE 2.5 – *Exemple d'une frontière.*

2.2. Horloges Logiques

Un système réparti est constitué de N composants (processus ou sites) communiquant par messages (et uniquement de cette manière). Chacun de ces composants agit comme un automate : il réalise des opérations qui modifient son état. Les opérations réalisées par un des composants sont naturellement ordonnées par l'ordre dans lequel elles sont réalisées : si il s'agit d'un processus abritant plusieurs activités, sur un système monoprocesseur, c'est l'ordre de l'exécution des instructions sur ce processeur qui ordonne les événements. La définition de l'ordre des événements sur un système multi-processeurs (fortement et a fortiori faiblement couplés) est plus problématique du fait de la difficulté de maintenir une notion de temps absolu cohérente.
Utiliser les horloges matérielles de chaque site, risque d'avoir une datation incohérente de l'évènement de réception d'un message : la date de réception précède la date d'émission. C'est pour cette raison on associe à chaque site une horloge logique locale.
Une horloge logique est un dispositif logiciel qui sert à établir et mesurer la notion de temps établie selon la relation de causalité dans un système réparti asynchrone .

2.2.1. Horloge logique scalaire (Lamport 1978)[7]

- H : ensemble des événements de l'application, muni de l'ordre partiel *to*

- T : domaine de temps, muni de l'ordre partiel $prec$

1. Chaque site gère un compteur dont la valeur est un entier, initialisée à 0. Les valeurs des horloges sont comparables en tant que valeurs entières.
2. Pour chaque événement local de Pi : $H_i = H_i + 1$: on incrémente l'horloge locale
3. Émission d'un message par Pi : On incrémente H_i de 1 puis on envoie le message avec (i, Hi) comme estampille
4. Réception d'un message m avec estampille (s, nb) : $H_i = max(H_i, nb) + 1$ et marque l'événement de réception avec H_i(Hi est éventuellement recalée sur l'horloge de l'autre processus avant d'être incrémentée).

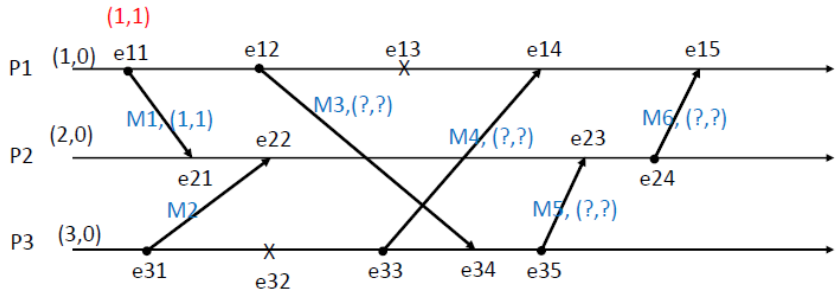


FIGURE 2.6 – *Exemple d'horloge de Lamport.*

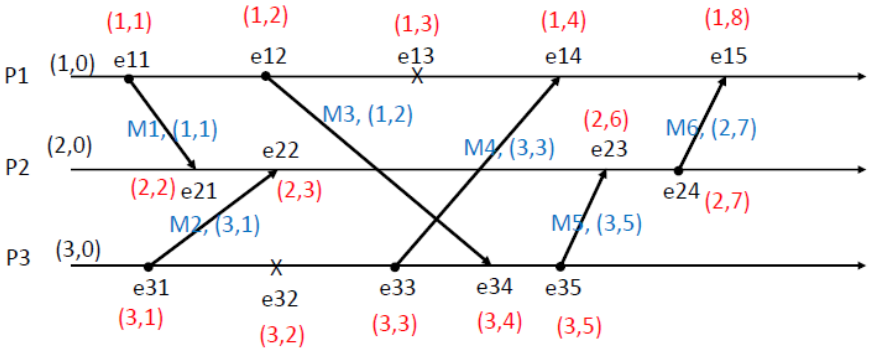


FIGURE 2.7 – *Solution avec l'horloge de Lamport*

2.2.1.1. Propriétés de l'horloge scalaire

- L'horloge H respecte la causalité :$(a \rightarrow b) \Rightarrow H(a) < H(b)$
  1. vrai pour deux événements locaux
  2. vrai pour une émission et une réception
  3. vrai dans tous les cas par induction le long du chemin causal
- H capture la causalité mais ne la caractérise pas : $H(a) < H(b) \not\Rightarrow a \rightarrow b$ Exemple : $H(e_{12}) = 2 < H(e_{23}) = 4$ et pourtant $e_{12}||e23$
- H(e) = n indique que (n - 1) événements se sont déroulés séquentiellement avant e.

Exemples utilisation de l'horloge de Lamport : - Exclusion mutuelle répartie (la datation logique permet d'ordonner totalement les requêtes pour garantir la vivacité de l'algorithme).

2.2.2. Horloge logique vectorielle (Fidge et Mattern 1989-1991) [8]

C'est une horloge qui assure la réciproque de la dépendance causale $H(e) < H(e') \Rightarrow e \rightarrow e$, elle permet également de savoir si 2 événements sont parallèles (non dépendants causalement).

2.2.2.1. Principe

- Utilisation de vecteur V de taille égale au nombre de processus
- Localement , chaque processus Pi a un vecteur Vi ;
- Un message est envoyé avec un vecteur de date ;
- Pour chaque processus P_i , chaque case $V_i[j]$ du vecteur contiendra des valeurs de l'horloge du processus P_j ;
- Initialisation : pour chaque processus P_i , $V_i = (0, ..., 0)$;
- Pour un processus P_i , à chacun de ses événements (émission, réception) ;
  1. $V_i[i] = V_i[i] + 1$;
  2. Si émission d'un message, alors V_i est envoyé avec le message ;

- Pour un processus P_i , à la réception d'un message m contenant un vecteur V_m , on met à jour les cases $j \neq i$ de son vecteur local V_i

  1. $\forall j : V_i[j] = max(V_m[j], V_i[j]$:
  2. Mémorise le nombre d'événements sur P_j qui sont sur P_j dépendants causalement par rapport à l'émission du message ;
  3. La réception du message est donc aussi dépendante causalement de ces événements sur P_j ;

2.2.2.2. Exemple : chronogramme d'application horloge de mattern

Même exemple que pour horloge de Lamport :

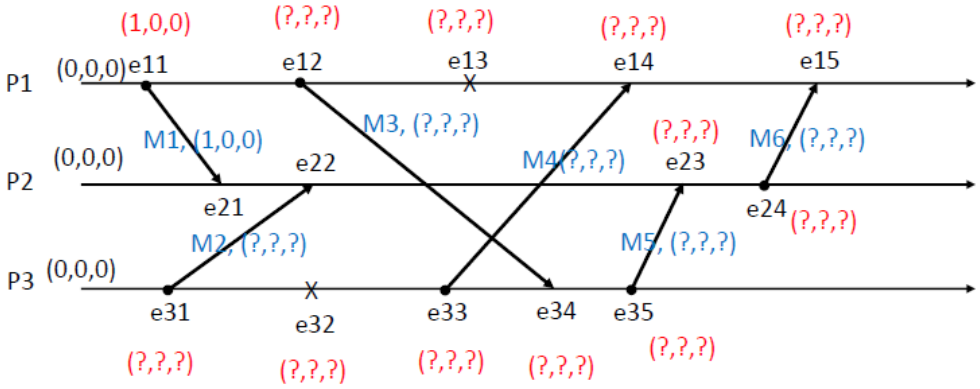


Figure 2.8 – *Exemple d'application horloge vectorielle.*

Solution de l'exemple :

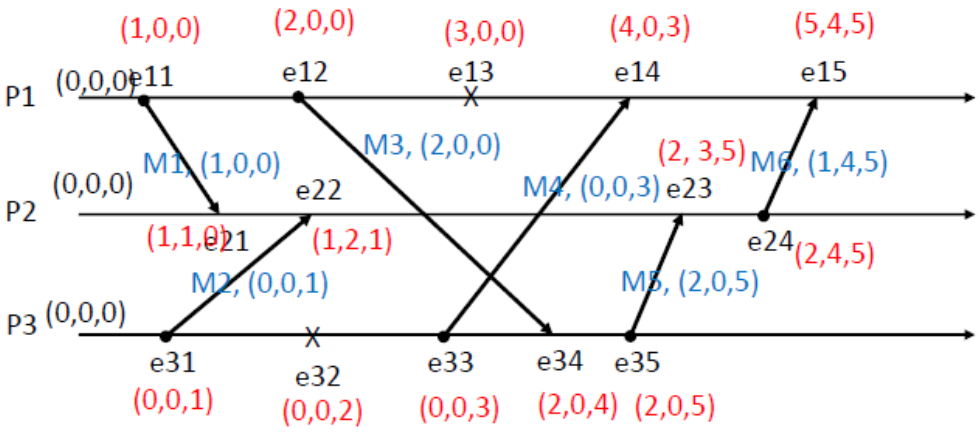


FIGURE 2.9 – *Solution.*

2.2.2.3. RELATION D'ORDRE PARTIEL SUR LES DATES

1. $V \leq V'$ défini par $\forall i : V[i] \leq V'[i]$
2. $V < V'$ défini par $V \leq V'$ et $\exists j$ tel que $V[j] < V'[j]$
3. $V||V'$ défini par $\neg(V < V') \wedge \neg(V' < V$

2.2.3. HORLOGE LOGIQUE MATRICIELLE [9]

L'estampille vectorielle ne corrige pas la défaillance vis-à-vis la délivrance causale des messages.

2.2.3.1. PRINCIPE DES ESTAMPILLES MATRICIELLES

Dans un système de n sites, les horloges d'un site i et les estampilles des événements (et des messages) sont des matrices carrées d'ordre n.
HM_i désigne l'horloge matricielle du site S_i ;
EM_m désigne l'estampille matricielle du message m ;

- **Sur le site S_i,la matrice HM_i va** :

  1. mémoriser le nombre de messages que le site S_i a envoyé aux différents autres sites : cette information est fournie par la $i^{ème}$ ligne de la matrice.

2. L'élément diagonal représente la connaissance qu'a Si du nombre d'événements locaux qui se sont déjà produits sur les différents sites S_j :correspond à l'estampille vectorielle.
3. Mémoriser, pour chacun des autres site s_j, le nombre de messages émis par ce site dont le site i a connaissance (i.e dont le site S_i sait qu'il sont été envoyés).
4. Ainsi sur le site i, la valeur $EM_i[j,k]$ donne le nombre de messages en provenance du site S_j délivrables sur le site S_k dont le site S_i a connaissance(i.e dont l'envoi est causalement antérieur à tout ce qui arrivera dorénavant sur S_i).

- **La modification synchronisation des horloges des différents sites est réalisée de la manière suivante :**

1. Lorsqu'un événement local se produit sur le site Si : $HM_i[i,i]$ est incrémenté ;
2. Lorsqu'un message est expédié à partir du site Si vers le site Sj : $HM_i[i,i]$ et $HM_i[i,j]$ sont incrémentés ;
3. Lorsqu'un message m en provenance du site S_j est reçu sur le site S_i , il faut s'assurer que tous les messages envoyés antérieurement au site S_i y sont effectivement arrivés. Cela suppose donc que Si ait reçu :
   (a) d'une part tous les messages précédents de S_j
   (b) d'autre part tous ceux envoyés plus tôt causalement depuis d'autres sites : cela correspond aux conditions suivantes (à vérifier dans l'ordre) :
   1. $HM_m[j,i] = HM_i[j,i] + 1$(ordre FIFO sur le canal (j,i)) ;
   2. pour tout $k \neq i,j$, $HM_m[k,i] \leq HM_i[k,i]$ (tous les messages en provenance des sites différents de S_j ont été reçus).

- Si toutes ces conditions sont vérifiées, le message est délivrable et l'horloge du site S_i est mise à jour :

1. $HM_i[i,i]++$ (incrémentation) ;
2. $HM_i[j,i]++$ (incrémentation) ;
3. pour le reste de la matrice : $HM_i[k,l] = max(HM_i[k,l], EM_m[k,l])$

- Si les conditions ne sont pas toutes vérifiées, la délivrance du message est différée jusqu'à ce qu'elles le deviennent et l'horloge n'est pas mise à jour.
- La délivrance d'un message pourra ainsi provoquer celle des messages arrivés prématurément.

2.2.3.2. Exemple d'application de l'horloge matricielle

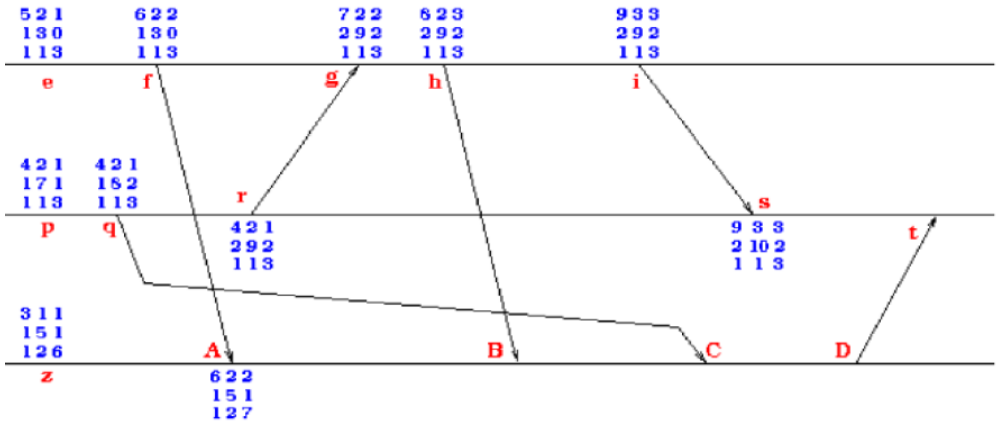


Figure 2.10 – *Exemple d'application d'horloge matricielle.*

La figure suivante correspond à ce qui se passe en utilisant les horloges matricielles dans la fin du scénario de l'exemple lorsque l'ordre d'arrivée des messages n'est plus satisfaisant :

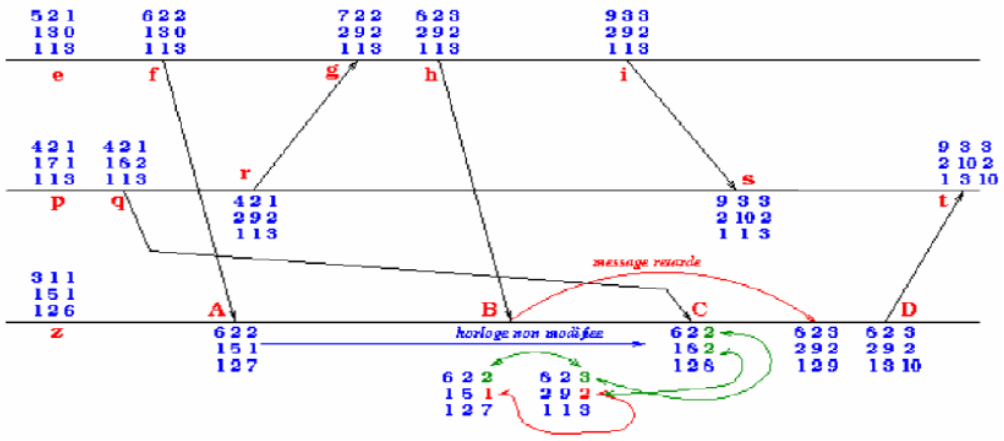


Figure 2.11 – *Solution.*

On peut remarquer que la première condition est vérifiée :
$HM_m[j,i] = HM_i[j,i] + 1 \Rightarrow HM_m[1,3] = HM_i = 3[1,] + 1 \Rightarrow 3 = 2 + 1$ (Verifiée voir la figure 2.11 en vert)
Mais, la deuxième condition n'est pas vérifié :
pour tout $k \neq i, j$, $HM_m[k,i] \leq HM_i[k,i]$: k=2, $HM_m[2,3] \leq HM_3[2,3] \Rightarrow 2 \leq 1$ (Non Verifiée voir la figure 2.11 en rouge), ce qui implique de retarder le message.

2.3. Exercices

2.3.1. Exercice 01 :

Complétez le chronogramme suivant :

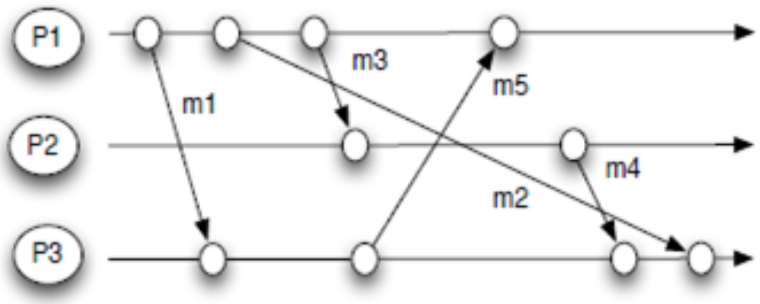


Figure 2.12 – *Chronogramme d'une exécution répartie.*

- Avec l'horloge de Lamport
- Avec l'horloge de Mattern

2.3.2. Exercice 02 : Coupure

On considère les deux coupures désignées C1 et C2 dans la figure ci-dessous :
- Les coupures C1 et C2 sont-elles cohérentes ? Justifiez vos réponses.

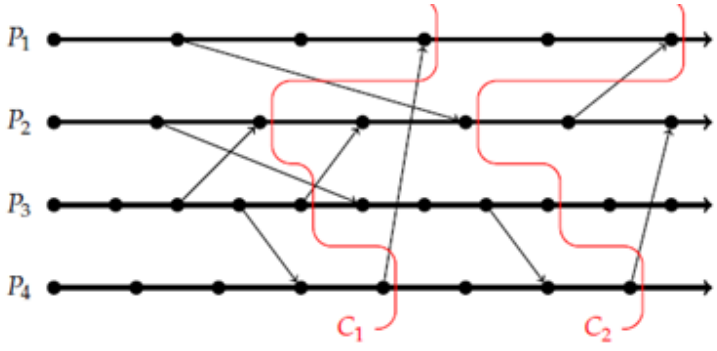


Figure 2.13 – *Coupure.*

2.3.3. EXERCICE 03 : HORLOGE MATRICIELLE

On considère un système distribué constitué de trois sites nommés P1, P2 et P3, utilisant des horloges matricielles pour dater les événements de chaque processus.

L'état initial du chaque processus est : $\begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$

Après quelques instants d'exécution, les horloges des processus P1, P2 et P3 indiquent les dates suivantes :

$$HM1 = \begin{bmatrix} 4 & 0 & 1 \\ 1 & 1 & 0 \\ 1 & 0 & 2 \end{bmatrix} \quad HM2 = \begin{bmatrix} 0 & 0 & 0 \\ 1 & 3 & 1 \\ 0 & 0 & 0 \end{bmatrix} \quad HM3 = \begin{bmatrix} 3 & 0 & 1 \\ 1 & 3 & 1 \\ 1 & 0 & 4 \end{bmatrix}$$

1. Au total, combien de messages ont été échangés ?
2. Au total, combien d'événements ont eu lieu ?
3. Au total, combien d'événements locaux ont eu lieu ?

Chapitre 3

Algorithmes d'exclusion mutuelle réparti.

3.1. Introduction

Dans un système distribué, les processus sont susceptibles d'accéder à une ou plusieursressources partagées (variables, pages mémoires, fichiers, etc.). Pour éviter les incohérencesdues aux accès concurrents des processus, il est indispensable que ceux-ci synchronisentleurs accès. lE paradigme de l'exclusion mutuelle assure qu'à un instant donné, au plus un processus peut exécuterune partie d'un code concurrent, appeléesection critique.

3.2. Definition

Une ressource partagée ou une section critique n'est accédée que par un processus à la fois, donc un processus est dans les trois états possibles, par rapport à l'accès à la ressource :demandeur, dedans et dehors. La couche middleware gère le passage de l'état demandeur à l'état dedans, par contre, le changement d'états par un processus se fait de l'état dehors à à l'état demandeur et de l'état dedans à l'état dehors [11].

3.3. Propriétés [11]

L'accès en exclusion mutuelle doit respecter 4 propriétés :
1- Sûreté : un processus au maximum en section critique (état dedans)
2- Vivacité : toute demande d'accès à la section critique est satisfaite en un temps fini.
3-Symétrique : les processus jouent le même rôle.
4- Equitable : pas de processus favori.

3.4. Mécanismes d'exclusion mutuelle distribuée

En système distribué, nous présentons deux classes de mécanismes d'exclusion mutuelle
- les mécanismes centralisés ;
- les mécanismes distribués.

3.4.1. Mécanisme de synchronisation centralisé [12]

Le mécanisme centralisé de synchronisation en système distribué regroupe les algorithmes qui utilisent un coordinateur pour gérer l'exclusion mutuelle. Ce coordinateur :

1. Centralise les requêtes des différents processus de l'application.
2. Réalise l'exclusion mutuelle en accordant la permission d'entrée en Section Critique.
3. Gère une file des requêtes en attente de Section Critique.

3.4.1.1. Algorithme

- Coté Processus P_i

- Quand un processus P_i veut entrer en Section Critique, il envoie au coordinateur un message de "demande d'entrée en Section Critique". Il attend la permission du coordinateur, avant d'entrer en Section Critique.
- Lorsque P_j reçoit la permission, il entre en Section Critique.
- Quand P_i quitte la Section Critique, il envoie un message de sortie de Section Critique au coordinateur

- Coté Coordinateur

- Si personne en Section Critique, à la réception d'une demande d'accès de P_i , il retourne "permission d'accès à P_i" .
- Si un processus P_j est déjà en Section Critique. Le coordinateur refuse l'accès. La méthode dépend de l'implantation :

  1. soit il n'envoie pas de réponse, le processus P_j est bloqué.
  2. soit il envoie une réponse : "Permission non accordée".

  Dans les deux cas, le coordinateur dépose la demande de P_j dans une file d'attente.

- A la réception d'un message de sortie de Section Critique, le coordinateur prend la première requête de la file d'attente de la Section Critique et envoie au processus un message de permission d'entrée en Section Critique.

3.4.1.2. Avantages

1. Algorithme garantit l'exclusion mutuelle,
2. Algorithme juste (les demandes sont accordées dans l'ordre de réception),
3. Pas de famine (aucun processus ne reste bloqué),
4. Solution facile à implanter
5. Complexité : maximum 2 ou 3 messages.

3.4.1.3. Inconvénients

1. Si coordinateur a un problème, tout le système s'effondre
2. Si processus bloqués lors de demande d'accès à une Section Critique occupée : impossibilité de détecter la panne du coordinateur.
3. Dans de grands systèmes, le coordinateur = goulet d'étranglement.

3.4.2. Mécanisme de synchronisation distribué

Plusieurs chercheurs décrivent une classification des algorithmes d'exclusion mutuelle. Ces classifications, classent les algorithmes en deux grandes catégories :

- **Les algorithmes à permissions :** le site demandeur doit recevoir l'accord d'un ensemble d'autres sites pour accéder à la section critique.
    1. Permissions individuelles : le processus désirant rentrer en section critique doit obtenir la permission de tous les processus formant le système . Un processus peut aussi donner sa permission à plusieurs processus à la fois. Exemple : Alg de Lamport , Alg de Ricart et Agrawala, Alg Carvalho et Roucairol.
    2. Permission par arbitre : Un processus ne donne qu'une permission à la fois. Il joue le rôle d'arbitre. Exemple : Alg de Maekawa .
- **Les algorithmes à jeton :** un jeton unique circule sur l'ensemble des sites et donnele droit à son possesseur d'entrer en section critique. L'unicité du jeton assure la sûreté. Exemple :Algorithme de Le Lann , Algorithme de suzuki-kasami, Algorithme de Raymond.

3.4.2.1. Algorithme de Lamport(1978) [11]

Cet algorithme vise à satisfaire les demandes des différents processus dans l'ordre où elles ont été formulées. ceci suppose donc l'existence d'un ordre dans les requêtes.A cet effet , l'horloge logique de Lamport est utilisée.

Hypothèses

- Chaque processus a un identifiant unique
- Les canaux de communication respectent la priorité FIFO : les messages sont reçus par P_i dans le même ordre qu'ils ont été émis par les autres processus P_j.
- Les canaux de communication sont fiables et synchrones : chaque message émis sur un canal est délivré correctement à son destinataire et ceci en un temps borné.
- Chaque processus utilise une horloge scalaire logique : cette horloge est synchronisée lors de la réception de messages en provenance des autres processus.
- Les processus sont corrects.

Variables pour chaque processus Pi :
Horloge locale Hi, Tableau des horloges des autres processus, initialisé à 0, Chaque processus maintient un tableau de requêtes : état de demande des autres processus (ACK, REL, REQ), Initialement : chaque processus connaît tous les autres participants : liste des voisins.

Messages utilisés :
REQ(H) : demande d'entrée en SC, ACK(H) : acquittement d'une demande d'entrée en SC, REL(H) : sortie de SC

Règle 1 entrée en SC
Accepte la demande de Section Critique **Faire**
Hi := Hi + 1
∀j ∈ Vi Envoyer REQ(Hi) à j
F_Hi[i] := Hi
F_Mi[i] := REQ
Attendre
∀j ∈ Vi((F_Hi[i] < F_Hi[j]) ∨ ((F_Hi[i] = F_Hi[j]) ∧ i < j))
<Section Critique>
Fait

Règle 2 Réception de (REQ)
Accepte le message REQ(H) depuis j **Faire**
Hi := Max(Hi,H) + 1
F_Hi[j] = H
F_Mi[j] = REQ
Envoyer ACK(Hi) à j
Fait

Règle 3 Réception de (ACK)
Accepte le message ACK(H) depuis j **Faire**
Hi := Max(Hi,H) + 1
Si F_Mi[j] != REQ **Alors**
F_Hi[j] = H
F_Mi[j] = ACK
Fsi
Fait

Règle 4 Sortie de la Section Critique
Accepte la libération de la Section Critique **Faire**
Hi := Hi + 1
∀j ∈ Vi Envoyer REL(Hi) à j
F_Hi[i] = Hi
F_Mi[i] = REL
Fait

Règle 5 Réception de REL(H)
Accepte le message REL(H) depuis j **Faire**
Hi := Max(Hi,H) + 1
F_Hi[j] = H
F_Mi[j] = REL
Fait

Complexité : 3 * (N - 1) messages par entrée en Section Critique, où N est le nombre de processus :
N - 1 Requêtes
N - 1 Acquittements
N - 1 Libérations.

3.4.2.2. Algorithme Ricart et Agrawala [13]

Algorithme proposé dans le but de diminuer le nombre de messages par rapport à l'algorithme de Lamport.
Le changement est de regrouper l'information de sortie de SC et accord, pas de retour systématique de ACK et n'informer que les processus en attente à la sortie de SC.

Principe de l'algorithme :

- Lorsqu'un processus P_i demande la Section Critique, il diffuse une requête

datée à tous les autres processus.

- Lorsqu'un processus P_i reçoit une requête de demande d'entrée en Section Critique de P_j, deux cas sont possibles :

  1. Cas 1 : si le processus P_i n'est pas demandeur de la Section Critique, il envoie un accord à P_j
  2. Cas 2 : si le processus P_i est demandeur de la Section Critique et si la date de demande de P_j est plus récente que la sienne, alors la requête de Pjest différée, sinon un message d'accord est envoyé à P_j.

- Lorsqu'un processus P_i sort de la Section Critique, il diffuse à tous les processus dont les requêtes sont différées, un message de libération

Variables utilisées par chaque processus Pi :

- Hi : entier, estampille locale.
- Initialement Hi = 0
- H_d_i : entier, estampille de demande de SC. Initialement $H\_d_i = 0$
- Di : booléen, le processus est demandeur de SC. Inità Di= FAUX
- differei[j] : processus dont l'accord est différé.
- $rep\_attendues\_i$: entier, nombre d'accords attendus. Init à $rep\_attendues\_i = 0$.
- Vi : voisinage de i.

Messages utilisés :
- Accès : demande d'entrée en SC
- OK : message de permission

- Procédure d'acquisition
  $hi \leftarrow hi + 1$;
  $Di \leftarrow vrai$;$h\_di \leftarrow hi$;
  $rep\_attendues_i \leftarrow n - 1$;
  pour tout ($j \in V \setminus \{i\}$) faire
  envoyer <Accès,h_di> à j

Réception de <Accès,h> venant de j→
1. $hi = max(hi, h) + 1$;
2. si (non Di) ou (h_di, i) > (h, j) alors envoyer <ok> *à j*
3. sinon $differe_i[j] \leftarrow vrai$;

- Réception de <ok> venant de j→
1. $hi = max(hi, h) + 1$;
2. $rep\_attendues_i \leftarrow rep\_attendues_i - 1$;
3. si ($rep\_attendues_i = 0$) alors ACCEDER A LA RESSOURCE

- Procédure Libération
  $Di \leftarrow faux$; $hi \leftarrow hi + 1$;
  pour tout $j \in V \setminus \{i\}$ faire
  Si $differe_i[j] = vrai$
  envoyer <ok> à j ;
  $differe_i[j] \leftarrow faux$;

Complexité : Une utilisation de la Section Critique nécessite $2 * (N - 1)$ messages :
N - 1 Requêtes
N - 1 Permissions

3.4.2.3. Algorithme de Carvalho et Roucairol [16]

Principe de l'algorithme :

- Soit le cas où P_i demande l'accès à la Section Critique plusieurs fois de suite (état demandeur plusieurs fois de suite) alors que P_j n'est pas intéressé par celle-ci (état dehors)
- Avec l'algorithme de Ricart et Agrawala, P_i demande la permission de P_j à chaque nouvelle demande d'accès à la Section Critique
- Avec l'algorithme de Carvalho et Roucairol, puisque P_j a donné sa permission à P_i, ce dernier la considère comme acquise jusqu'à ce que P_j demande sa permission à P_i : P_i ne demande qu'une fois la permission à P_j

3.4.2.4. Algorithme de Lann, 1977 [17]

Un jeton unique circule en permanence entre les processus via une topologie en anneau. Quand un processus reçoit le jeton : S'il est dans l'état demandeur : il passe dans l'état dedans et accède à la ressource. S'il est dans l'état dehors,

il passe le jeton à son voisin.
Quand le processus quitte l'état dedans, il passe le jeton à son voisin.

Inconvénients

- La circulation du jeton consomme des ressources inutilement quand aucun processus n'est demandeur de la sc (échange de msg)
- Perte de jeton
- Lenteur du jeton si anneau unidirectionnel
- Si le processus i+1 a le jeton et que le processus i veut accéder à la ressource et est le seul à vouloir y accéder, il faut quand même attendre que le jeton fasse tout le tour de l'anneau

Avantages

- Très simple à mettre en oeuvre
- Intéressant si nombreux sont les processus demandeurs de la ressource

3.5. Exercices

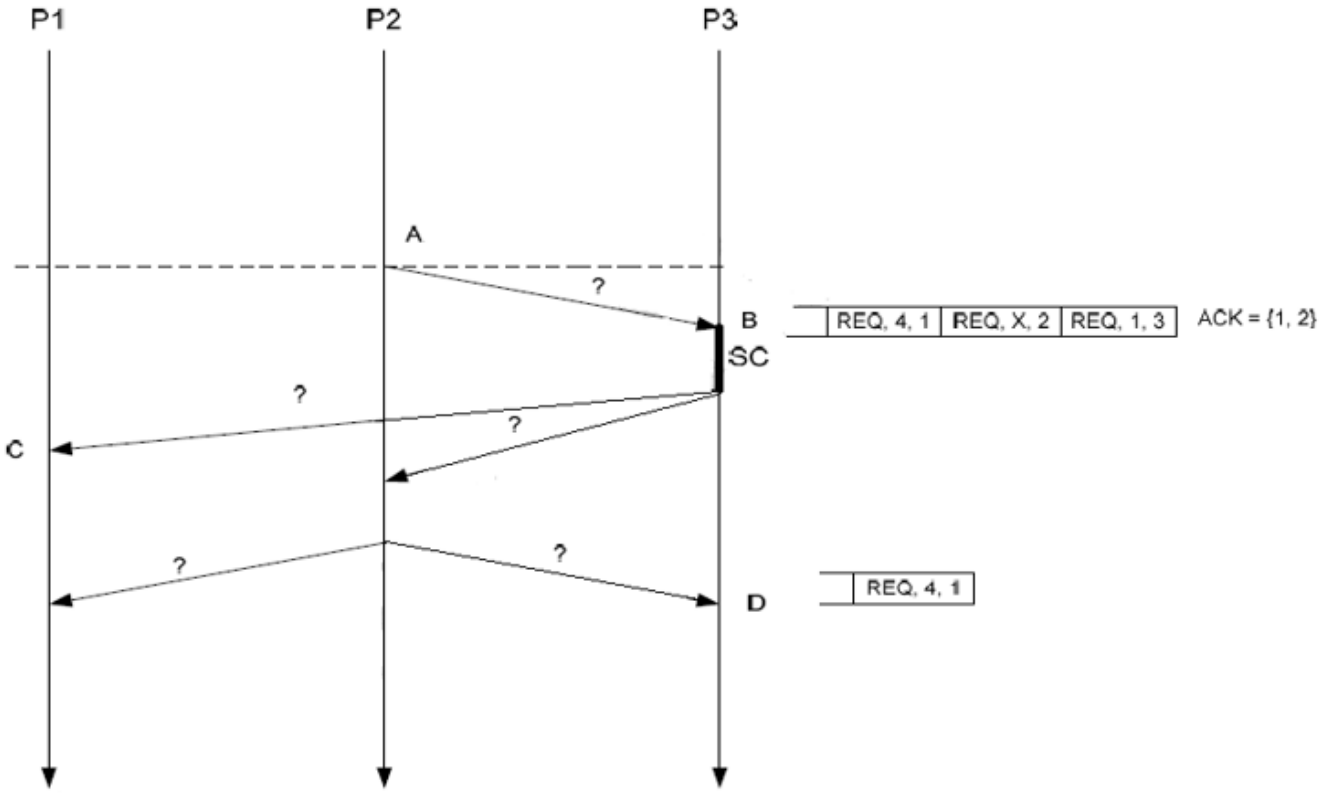
3.5.1. Exercice 01 :

On considère un système réparti à N sites (numérotés de 1 à N). On utilise l'algorithme de synchronisation de Lamport. Le site 1 est le seul à solliciter l'entrée à la SC. Il émet des requêtes REQ de façon itérative : après chaque sortie de la SC , il émet une nouvelle requête.

1. Faire un diagramme général montrant l'entrée et la sortie de la SC, ainsi que les ?messages échangés. ?
2. Pensez-vous qu'il y'a des messages qui peuvent être considérés comme inutiles ? Lesquels ? Que proposez-vous ?

3.5.2. Exercice 02 :

La figure suivante montre une partie des messages échangés entre 3 processus (P1, P2, P3) d'un système réparti pour utiliser une section critique (SC). Nous savons que P3 accède à la SC au point B ; le contenu de sa file de requêtes et son ensemble des réponses ACK sont précisés sur la figure suivante :



1. D'après-vous l'algorithme utilisé est celui de Lamport ou de Ricart-Agrawala ? Justifiez. ?
2. En supposant que la date de A=10, identifiez le type de chaque message marqué par "?" et datez le. ?
3. A partir des informations de la figure, peut-on connaitre à quel moment P2 peut-il entrer en SC ? Justifiez. Quelle est la valeur de X ?
4. On suppose que l'ensemble des réponses ACK de P1, au point C, est ACK=2, 3. A quel ?moment P1 entre-t-il en SC ?
5. Compléter la figure en ajoutant tous les messages manquants, à partir du point A. Donnez l'évolution du contenu des files de requêtes et d'ACK de chaque site. ?

Chapitre 4

Algorithmes d'élection de leader dans un système réparti

4.1. Introduction

Une élection permet de particulariser un processus dans un réseau. Cette particularisation est nécessaire dans certains algorithmes : panne du processus coordinateur, tâches spécifiques, régénérer le jeton perdue, diffusion d'information à l'ensemble du groupe et MAJ des sites dans un groupe...etc.

4.2. Définition

Le problème de l'élection est de partir d'une configuration dans laquelle tous les processus sont dans le même état pour arriver dans une configuration dans laquelle un seul processus est dans l'état (gagnant) et tous les autres dans l'état (perdant).
Choisir parmi un ensemble de processus , un et un seul processus et le faire connaitre de tous les autres processus.
Étant donné un ensemble de processus choisir un unique chef (sûreté) en un temps fini (vivacité).

4.3. Propriétés

Un algorithme d'élection est un algorithme qui satisfait les propriétés suivantes :

- Chaque processus exécute le même algorithme : symétrie complète ;
- L'algorithme est décentralisé : une exécution peut être commencée par un nombre quelconque de pro-cessus ;
- L'algorithme atteint une configuration terminale dans laquelle il existe exactement un processus « ga-gnant » et tous les autres processus sont (perdants).
- Les algorithmes d'élection sont basés sur l'identité des processus : choix du plus grand ou plus petit identifiant (numéro).

4.4. Algorithme de BULLY [14]

4.4.1. Hypothèses

- La topologie du réseau est quelconque (graphe)

- Les canaux de communication sont fiables et synchrones
- Chaque processus possède un ID unique(ex :adresse réseau)
- Les processus peuvent tomber en panne (panne franche)

4.4.2. Principe

Quand un processus P s'aperçoit que le coordinateur ne répond plus à ses requêtes(time-out sur TEMPO),il lance l'algorithme d'élection.
Envoi d'un message ELECTION à tous les autres processus dont le numéro est plus grand que le sien.
Le processus Q envoie un message ACK à P lui signifiant qu'il est actif. A son tour Q,lance une élection si ce n'est pas déjà fait.
Si aucun processus ne lui répond avant TEMPO, P gagnel'élection et devient le coordinateur. Si un processus de numéro plus élevé répond, c'est lui qui prend le pouvoir. Le rôle de P est terminé.
Le nouveau coordinateur envoie un message à tous les participants pour les informer de son rôle. L'application peut alors continuer à s'exécuter.

4.4.3. Formalisation

Algorithme 1 : Initialisation

```
1 Description :
2 for all i do
3 |   demande-electi = False ;
4 |   Leader = MaxEntier ;
5 |   elected = False ;
```

Algorithme 2 : Lancement d'une élection par Pi

```
1 Description :
2 demande-electi = True ;
3 for all j > i do
4 |   send( Pi, ELECT) ;
5 |   Armer délai de garde T (exécuter l'Agorithme 5) ;
6 FIN
```

Algorithme 3 : Reception d'un message (Pj, elect) par Pi($i > j$)

```
1 Description :
2 if non demande-electi then
3 |   demande-electi = True;
4 |   Send (Pi, ACK) ;
  for all k > i do
  |   send( Pi, ELECT) ; // Pi lance l'élection;
  |   Armer délai de garde T (exécuter l'Agorithme 5) ;;
  FIN
```

Algorithme 4 : Reception d'un message (elected ,Pj) par Pi

```
1 Description :
2 demande-electi = False ;
3 Leader =Pj ;
4 FIN
```

Algorithme 5 : Declenchement du délai T(ms) par Pi

```
1 Description :
2 if aucun msg ACK reçu then
3 |   Elected = True ;
4 |   Send (elected, Pi) to all ; //Pi est leader
5 else
6 |   Armer délai de garde T1 (Algoritme 6)
  FIN
```

Algorithme 6 : Declenchement du délai T1(ms) par Pi

```
1 Description :
2 if aucun msg Elected reçu then
3 |   Lancer une election
  FIN
```

4.4.4. Complexité

4.4.4.1. Complexité au pire des cas

Le processus de plus petit id détecte la panne et lance l'élection.
P1=n-1 msg
P2=n-2 msg
.....
Pn-1= 1 msg
= n(n-1)/2 msg
De plus Le leader doit envoyer un msg de notification à tous les processus => (n-1) msg supplémentaires.
D'où une complexité de 0(n^2)

4.5. Algorithme de Chang & Roberts [10]

L'élection se fait sur un anneau unidirectionnel, chaque site«i» dispose d'un pointeur vers son successeur succ[i].

4.5.1. Hypothèses

1. La topologie du réseau est un anneau unidirectionnel
2. Les canaux de communication sont fiables et synchrones
3. Chaque processus possède un ID unique
4. Les processus sont corrects

Resultat : Le processus ayant la plus grande/la plus petite Identité est élu Leader

Idée : chaque candidat diffuse autour de l'anneau sa candidature ; le processus ayant l'identifiant max gagne

4.5.2. Principe

Soit un ensemble de processus organisés autour d'un anneau unidirectionnel, chaque candidat diffuse autour de l'anneau sa candidature. Le processus ayant l'identité maximale est élu « leader ».

Si un processus décide d'être candidat à l'élection, il transmet son identité à son voisin. Lors de la réception de l'identifiant :

- Si id reçu > id local, le processus transmet l'id reçu.
- Si id reçu < id local, le processus transmet son id.
- Si id reçu = id local, le processus devient « leader » et transmet cette information à tous les processus

Remarque : un processus qui reçoit son propre Id conclut qu'il est « leader » car cet Id a fait un tour complet et il est supérieur à tous les autres Ids des membres de l'anneau : c'est le principe de l'extinction sélective.

4.5.3. Algorithme

Algorithme 7 : Candidature

```
1 Description :
2 Candidatute-i=true ;
3 envoyer (election, i) à succ[i] ;
```

Algorithme 8 : Reception d'un message (election, j) par Pi

```
1 Description :
2 Selon (j) ;
3 if j > i then
4 |  envoyer (election,j) à succ[i]
  if j < i then
  |  candidat-i=true ;envoyer (election,i) à succ[i]
  if j = i then
  |  envoyer (elu,i) à succ[i]
```

Algorithme 9 : Reception d'un message (elu,j) par Pi

```
1 Description :
2 Candidatute-i=false ;
3 Leader = j ;
4 if i ≠ j then
5 |  envoyer (elu, i) à succ[i]
6 else
7 |  Fin election
```

On peut remarquer qu'il y a deux types de messages :
election : indique que la valeur associée est candidate.
elu : indique que la valeur associée a gagné l'élection.

4.5.4. Complexité

4.5.4.1. Complexité au pire des cas

Les IDs sont ordonnées dans l'ordre décroissant autour de l'anneau. (1->n->n-1->n-2->2->1).
Tous les processus sont initiateurs.
Un processus pi visite i noeuds avant de décider de son statut.

- Le processus 1 envoie 01 msg et il s'eteint
- Le processus 2 envoie 02 msg et il s'eteint

  .....
- Le processus n envoie n messages.
- D'où n+(n-1)+(n-2) ..+1 = n(n+1)/2

De plus il faut n messages de notification : Total : n+ n(n+1)/2 = 1/2 n2+ 3/2 n = $O(n^2)$

4.5.4.2. Complexité au meilleur cas

Les IDs sont ordonnées dans l'ordre croissant autour de l'anneau (1->2>3->4->n-1->1)
Le processus Pn fait vehiculé n messages
le processus pi visite 01 noeuds avant l'extinction .

- Le processus 1 envoie 01 msg et il s'eteint
- Le processus 2 envoie 01 msg et il s'eteint

.....

- Le processus (n-1) envoie 01 msg et il s'eteint
- Le processus n envoie n messages
- D'où n+(n-1) messages.

De plus il faut n messages de notification : Total : n+ n+(n-1) = 3n-1 messages = $O(n)$.

4.6. Exercices

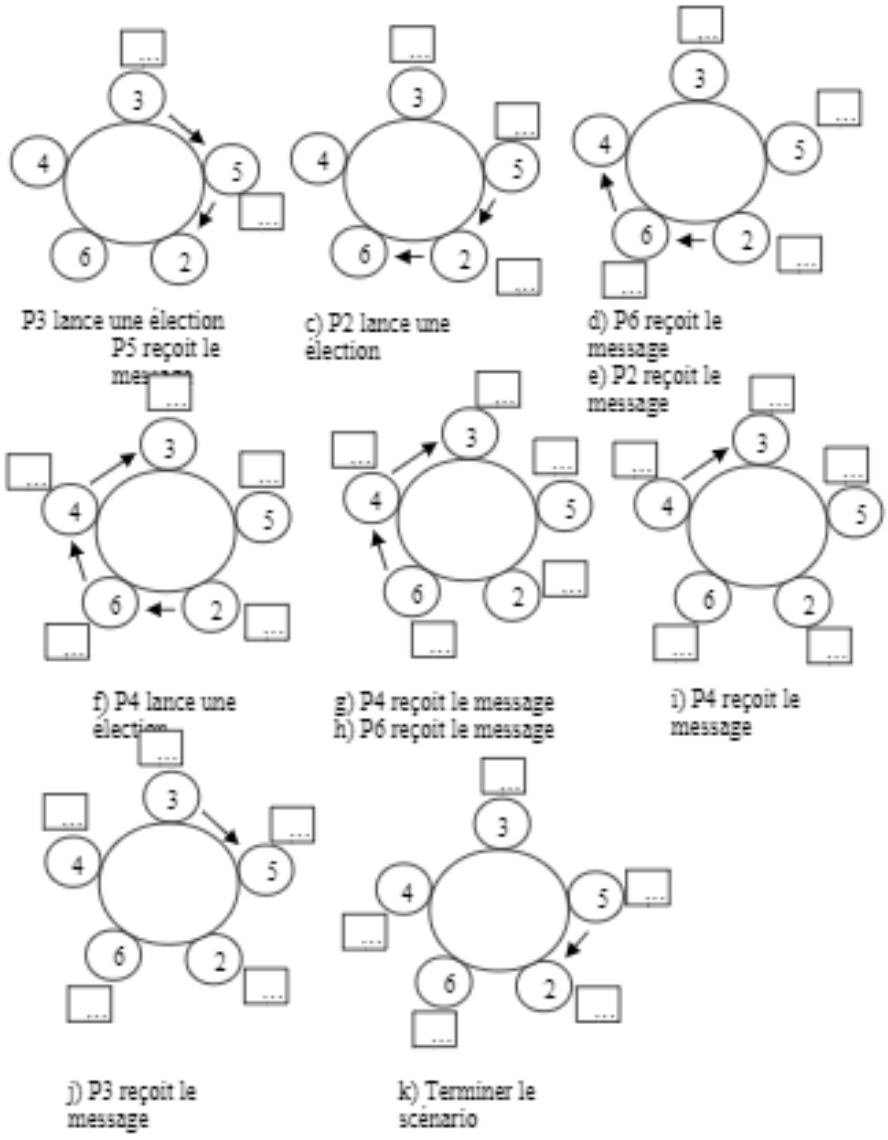
4.6.1. Exercice 01 :

Un groupe est constitué de dix processus, avec des identificateurs numérotés de 0 à 9. Auparavant, le processus P9 était le coordinateur, mais il vient de tomber en panne. Le processus P6 est le premier à le remarquer.

- Appliquer l'algorithme de BULLY pour élire un nouveau leader.

4.6.2. Exercice 02 :

Tracer l'évolution du scénario suivant en montrant les numéros véhiculés dans lemessage (élection,i)et l'évolution de la variable chef dans les différentsprocessus.


P3 lance une élection
P5 reçoit le
c) P2 lance une
élection
d) P6 reçoit le
message
e) P2 reçoit le
message
f) P4 lance une
g) P4 reçoit le message
h) P6 reçoit le message
i) P4 reçoit le
message
j) P3 reçoit le
message
k) Terminer le
scénario

Chapitre 5

Algorithmes de terminaison réparti

5.1. Introduction

Dans un système réparti, le problème de terminaison revient à détecter la fin d'un algorithme réparti. La détection de la terminaison n'est pas un problème trivial, un message émis par un processus non encore observé, et qui deviendra ensuite passif, peut réactiver un processus qui a été observé passif. Pour ce faire, il faut vérifier deux conditions sur l'état global du système :
1. Tous les processus sont au repos.
2. Aucun message n'est en transit dans les canaux.
L'objectif est de concevoir un mécanisme de contrôle distribué qui rende compte d'une propriété de stabilité globale (la terminaison).

5.2. Terminologies

- Un processus est dit **actif** lorsqu'il exécute son texte de programme et **passif** dans les autres cas.
- Un processus **passif** peut être soit terminé, soit en attente de message en provenance d'autres processus.
- l'algorithme distribué est dit terminé lorsque tous les processus sont **passifs** et qu'il n'y a pas de message en transit.

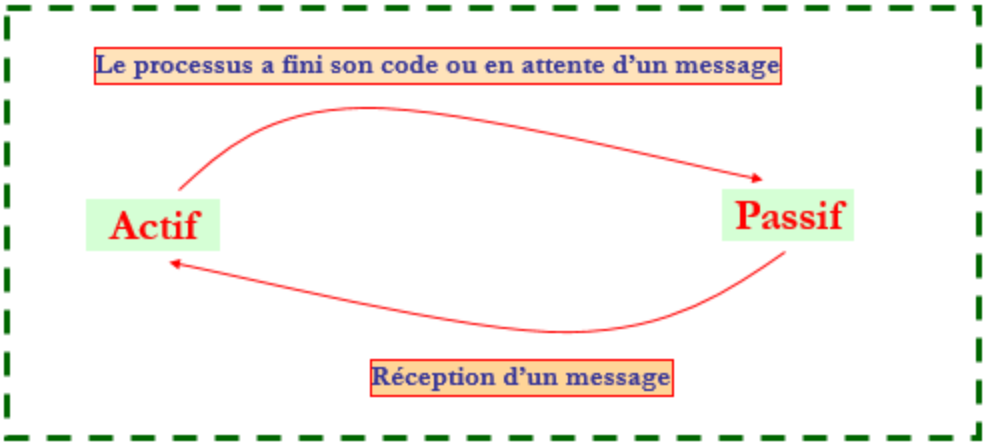


Figure 5.1 – *Etat d'un processus*

5.3. Méthodes pour détecter la terminaison

Il existe deux types de méthodes :
Méthode générale : analyse de l'état global.La terminaison est une propriété stable. On peut donc la détecter par examen d'un état global enregistré (Algorithme de Chandy et Lamport 1985 [6]).
Méthodes spécifiques : applicables à un schéma particulier de communication. Sur un anneau(Algorithme de MISRA [18]) et sur un arbre (algorithme Dijkstra-Scholten [19]).

5.4. Algorithme de Misra en 1983 [18]

L'algorithme de Misra consiste à faire visiter les processus par un jeton, la constatation par le jeton que tous les processus sont passifs lorsqu'ils ont été visités ne permet pas de conclure à la terminaison : des processus ont pu être réactivés et des messages peuvent être en transit.
La difficulté se situe si un message émis après le passage du visiteur (et non visible par celui-ci) peut venir réactiver (derrière lui) un processus trouvé passif.

5.4.1. Hypothèses

- Topologie en anneau unidirectionnel
- L'anneau est le seul moyen de communication
- Canaux de communication fiables ,synchrones et fifo
- Pas de perte de messages.

5.4.2. Principes de fonctionnement

La terminaison est réalisée lorsque tous les processus sont passifs et qu'il n'y a plus de messages en transit. Comme la topologie de communication entre processus est un anneau, le jeton peut affirmer que le calcul est terminé si, après avoir effectué un tour sur l'anneau, il constate que chaque processus est resté passif depuis sa dernière visite à ce processus. En effet, les messages ne pouvant pas se doubler, entre deux visites du jeton un processus a nécessairement reçu les messages qui étaient en transit lors de la première visite du jeton.

- Chaque processus Pi est doté des variables suivantes :
    1. état : (actif, passif) initialisé à actif.

2. couleur : (blanc, noir) initialisé à noir.
   noir = Pi a été actif depuis le dernier passage du jeton.
   blanc = Pi a été passif depuis le dernier passage du jeton.

3. jeton_présent : booléen initialisé à faux.
   Le jeton véhicule une valeur j.
   Le jeton porte un compteur des processus trouvés passifs.

4. nb : entier initialisé à 0.
   Sert à mémoriser la valeur associée au jeton entre sa réception et sa réémission.

- Les messages sont de deux types : messages, jeton.
- Réception d'un message ou initialisation :
  état = actif
  couleur = noir
- Attente d'un message ou fin :
  état = passif

5.4.3. Algorithme

Algorithme 10 : Reception de Jeton (Valeur=j)

```
1 Description :;
2 Nb=j ; //Nb :Nombre de processus Jeton-Present=Vrai ;
3 if Nb=N and couleur=lan then
4 |  Terminaison détectée ;
  FIN
```

Algorithme 11 : Emission de Jeton (Valeur=j)

```
1 Description :;
2 if Jeton-Present and etat=passif then
3 |  if couleur=blan then
4 |  |  Nb++ ;
5 |  else
6 |  |  Nb=0 ;
  |  Envoyer(jeton,Nb) à successeur Pi ; couleur=blan ; jeton-present =
  |  faux ;
  FIN
```

5.5. Algorithme de Dijikstra & Sholten 1980 [19]

5.5.1. Concept de calcul diffusant

Le concept de calcul diffusant définit un mode de communicationentre processus :

- Les voies de communication entre processus permettent de définir un graphe dirigé Gdont les processus sont les sommets ;
- on définit un noeud (processus) particulier qui n'a pas de prédécesseur que l'on appelle environnement ou racine ;
- initialement tous les processus sont passifs et le calcul diffusant débute lorsquel'environnement envoie un ou plusieurs messagesà un ou plusieurs de ses successeurs ;
- ce n'est qu'après avoir reçu un premier message qu'un processus devient actifet peut à son tour émettre des messages vers d'autres processus (ses successeurs dans G).

5.5.2. Hypothèses

- Un processus ne peut émettre qu'un nombre fini de messages.
- Les délais de transmission des messages sont arbitraires mais finis (pas de pertes).

5.5.3. Terminologies

1. On associe à tout message, émis d'un processus Pi vers un processus Pj, un message particulier ou signal, émis de Pj vers Pi.
2. Ces messages constituent une sorte d'acquittement et leurs émissions sont telles que :
3. Lorsque le processus environnement a reçu tous les signauxrelatifs aux messages qu'il a émis, il en conclut que le calcul diffusant est terminé.
4. On introduit la notion de déficit associée à chaque connexion entre deux processus (arc dans le graphe)comme étant la différence entre le nombre de messages reçus et le nombre de signaux renvoyés sur cette voie de communication.

Pour chaque processus Pi on définit les variables :

- defin et defout : comptent respectivement la somme des déficits des arcs entrantvers le processus et la somme des déficits des arcs sortant.
- initialement ces deux compteurs sont nuls

Gestion des compteurs definet defoutpar Pi :

- envoi d'un message :defout=defout + 1 ;
- réception d'un message : defin=defin + 1 ;
- envoi d'un signal : defin=defin 1 ;
- réception d'un signal :defout=defout 1 ;

A l'exception du processus environnement(racine), tous les autres processus ne peuvent émettre des messages que s'ils en ont reçu un et s'ils ne sont pas revenus dans leur état initial (caractérisé par defin = defout = 0).

5.5.4. Principe de l'algorithme

Tout processus détient 04 variables : Defin, Defout , père et appelants.
Père : père du processus
Appelants :Ensemble de processus autre que le père qui ont envoyé un msg au processus .
La racine détecte la terminaison par defin(racine)= defout(racine) = 0 après calcul

Algorithme 12 : Emission de (message, racine) vers j : Activation de Pj par la racine

```
1 Description : defout=defout+1 ; Envoyer (message, racine) vers j ; FIN
```

Algorithme 13 : Emission de (message, i) vers j : activation de Pj par Pi

```
1 Description :
2 if defin ≠ 0 then
3 |  defout=defout+1 ; envoyer(message,i) à j ;
  FIN
```

Algorithme 14 : Reception de (message, Exp) : activation de Pi par Expéditeur

```
1 Description :
2 if defin = 0 then
3 |  père = Exp ; // le père est le premier expéditeur;
4 else
5 |  Appelant = Appelant + Exp ;
  defin= defin + 1;
  FIN
```

Algorithme 15 : Reception de (signal, Exp)

```
1 Description :
2 /* une réponse est reçu par Pi de Exp */
3 defout = defout - 1;
```

Algorithme 16 : Emission (signal, i) : Pi envoi un signal

```
1 Description :
2 if defin = 1 and defout = 0 then
3 |  Envoyer(signal,i) à père ; // envoyer réponse au père
  if defin > 1 then
  |  x = un elément de Appelant ; // envoyer réponse à un autre activateur
  |  Appelant = Appelant - x ;
  |  Envoyer (signal,i) à x ;
  defin= defin - 1 ;
```

5.6. Exercices

5.6.1. Exercice 01 :

La détection de la terminaison d'un calcul est l'un des problèmes étudiés dans les systèmes répartis.

1. Expliquez en quelques lignes pourquoi ce problème est réputé difficile ?
2. Pour modéliser ce problème, on considère que chaque processus est soit dans l'état actif, soit dans l'état passif, à un instant donné. Expliquez ce que signifie chaque état ?
3. Quand, un processus peut-il passer de l'état actif à l'état passif ?
4. Quand, un processus peut-il passer de l'état passif à l'état actif ?
5. A un instant donné, on constate que tous les processus du système sont dans un état passif. Peut-on dire qu'il y'a terminaison ? Justifiez.
   Pour résoudre ce problème, un algorithme (vu en cours) organise les processus en un anneau virtuel unidirectionnel, sur lequel circule un jeton. On suppose que chaque processus est sur un site différent.
6. Que contient ce jeton ?
7. Le jeton est coloré en blanc ou en noir . Que signifie chaque couleur ?
8. Quelle est le traitement à faire lorsque le jeton arrive sur un site ?
9. Avec cet algorithme, quand pouvons-nous dire que nous avons une terminaison de calcul ?

5.6.2. Exercice 02 :

On considère une exécution répartie de 5 processus (P0, P1, P2, P3, P4) , Nous supposant le scénario suivant et P0 représente la racine (voir figure) :

1. P0 active p1
2. P1 active p3
3. p0 active p2
4. p2 active p4

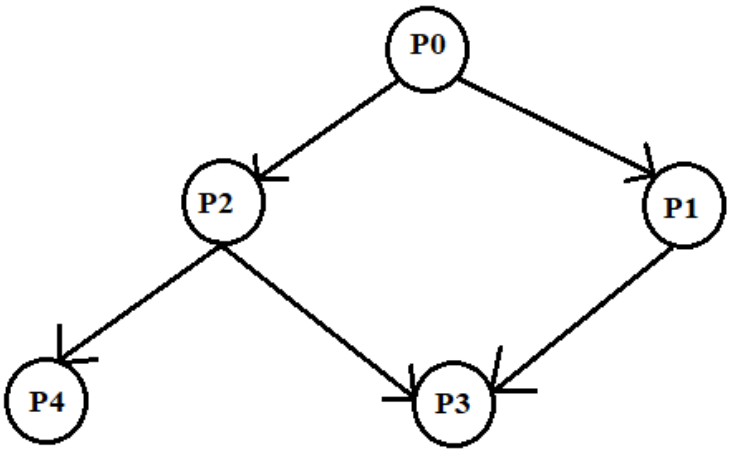


FIGURE 5.2 – *Représentation des processus*

5. p2 active p3

- Dérouler l'algorithme de DIJKSTRA & SHOLTEN et détecter la terminaison.

Chapitre 6

Le problème du consensus dans un système distribué

6.1. Introduction

Le problème du consensus est un problème fondamental en théorie du calcul distribué. Le principe étant de parvenir à obtenir une certaine fiabilité d'un système dans des problèmes de décision distribuée, en présence de pannes. On peut citer quelques exemples concrets d'applications de ce problème dans les bases de données distribuées, l'élection d'un leader ....etc.
Le consensus demande à ce qu'un certain nombre de processus s'accordent sur une valeur unique. Les processeurs peuvent tomber en panne, ou être non fiables, donc les protocoles de consensus doivent être tolérants aux défaillances.
Certaines approches à ce problème ne demandent qu'une majorité de processus qui s'accordent, et la valeur sera tirée par votes [16].

6.2. Définition du consensus

Les processus s'accordent sur une valeur après qu'un ou plusieurs processus ont proposé ce que cette valeur devrait être [16].

6.3. Type de consensus

6.3.1. Consensus uniforme

- Accord uniforme : la valeur décidée est la même pour tous les processus (corrects ou ultérieurement fautifs) qui décident
- Un processus peut décider puis devenir incorrect.

6.3.2. k-consensus

- k Accord : au plus k valeurs distinctes sont décidées pour l'ensemble des processus corrects
- Consensus basique : k = 1

6.4. Modélisation d'un consensus [16]

Soit un ensemble de (n) processus (P1,..,Pn) reliés par des canaux de communication et s'envoyant des messages.

- Le consensus doit être atteint même en présence de fautes : Nous supposons, que les communications sont fiables mais les processus peuvent crasher.
- Initialement : chaque processus Pi est indécis et propose une valeur vi.
- A la terminaison de l'algorithme : chaque Pi décide d'une valeur di.
- Les exigences d'un algorithme de consensus :
    1. Accord : la valeur décidée est la même pour tous les processus corrects.
    2. Intégrité : tout processus décide au plus une fois (sa décision est définitive).
    3. Validité : la valeur décidée est l'une des valeurs proposées.
    4. Terminaison : tout processus correct décide au bout d'un temps fini.

Par exemple, nous pouvons collecter les processus dans un groupe et faire en sorte que chaque processus diffuse de manière fiable la valeur proposée aux membres du groupe.
Chaque processus attend d'avoir collecté toutes les N valeurs (y compris les siennes). Il évalue ensuite la fonction majority(v1,v2 ,...vN), qui renvoie la valeur qui apparaît le plus souvent parmi ses arguments. Chaque processus reçoit le même ensemble de valeurs proposées et chaque processus évalue la même fonction de ces valeurs. Donc, ils doivent tous être d'accord, et si chaque processus propose la même valeur, ils décident tous de cette valeur.
majority() n'est qu'une fonction possible que les processus pourraient utiliser pour convenir d'une valeur à partir des valeurs candidates. Par exemple, si les valeurs sont ordonnées, les fonctions minimum() et maximum() peuvent être appropriées.

6.4.1. Défaillance d'un processus

- Arrêt (crash failure ou panne franche) : le processus fonctionne correctement jusqu'a un point ou il cesse définitivement d'agir.
- Omission : Fautes Temporaires
    1. omission en émission : le processus omet certaines émissions qu'il aurait dû faire, ou cesse définitivement.
    2. omission en réception : le processus ignore certains messages en réception, ou cesse définitivement.
- Arbitraire (byzantine failure) : le processus ment (par omission ou par contenu arbitraire des messages envoyés, fautes malicieuses volontaires)

6.4.2. Consensus dans un contexte avec fautes

6.4.2.1. Système synchrone et pannes franches

Hypothèse

- Les communications sont fiables (pas de perte ni duplication de messages).
- On peut déterminer si un processus est en panne.
- Utilisation de la diffusion fiable.

Résultat tous les processus recevront les mêmes valeurs et pourront appliquer la même fonction qui conduira forcément au même choix.

6.4.2.2. Système asynchrone et pannes franches

En théorie : Résultat d'impossibilité.
En 1985, Fischer, lynch et Paterson (FLP) ont montré que « dans un système asynchrone, même avec un seul processus fautif, il est impossible d'assurer que l'on atteindra le consensus ».
Intuitivement, il est impossible de distinguer un processus lent d'un processus arrêté.

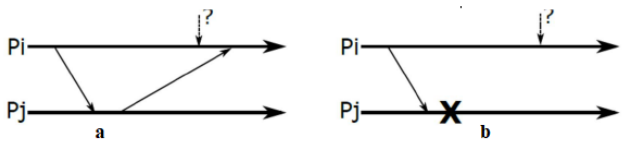


Figure 6.1 – *(a) processus lent ; (b) processus arrêté*

En pratique : il est possible d'atteindre des résultats satisfaisants :
- Se baser sur les systèmes partiellement asynchrones.
- Utiliser des détecteurs de fautes.

6.4.2.3. Système synchrone et pannes byzantines

En théorie : il n'est pas possible d'assurer que le consensus soit toujours atteint.
Resultat : En 1982, Lamport, Shostak et Pease ont montré que « si f est le nombre de processus fautifs, il faut au minimum 3f+1 processus pour que le consensus soit assuré. En dessous il n'est pas assuré. »
Le consensus ne peut être résolu que s'il y a plus de deux tiers de processus corrects. Solutions lourdes en nombre de messages

6.4.2.4. Système asynchrone et pannes byzantines

En théorie : Le consensus n'est pas toujours atteint (très difficile à atteindre) En pratique : Quelques algorithmes spécifiques existent mais ils ne sont pas parfaits.

Le consensus est atteint facilement dans un contexte synchrone sans fautes ou avec pannes franches Mais Il est difficile à résoudre dans un milieu asynchrone avec pannes franches ou byzantine.

Chapitre 7

Annexe

7.1. Solutions des exercices

7.1.1. Exercices du chapitre 02

7.1.1.1. Exercice 01 : Horloge

Application de l'horloge de Lamport (scalaire) sur le Chronogramme suivant :

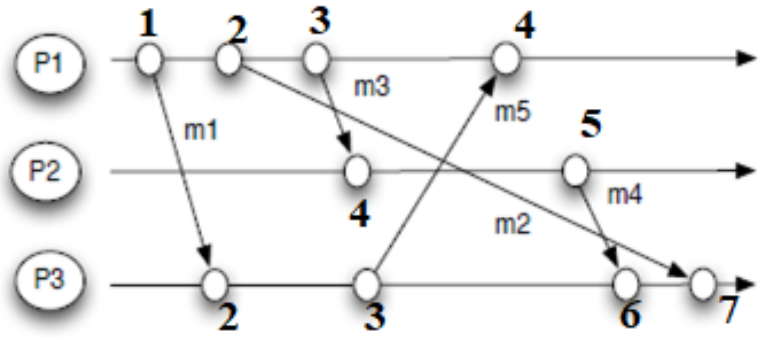


Figure 7.1 – *Horloge de Lamport.*

Application de l'horloge de Mattern (vectorielle) sur le Chronogramme suivant :

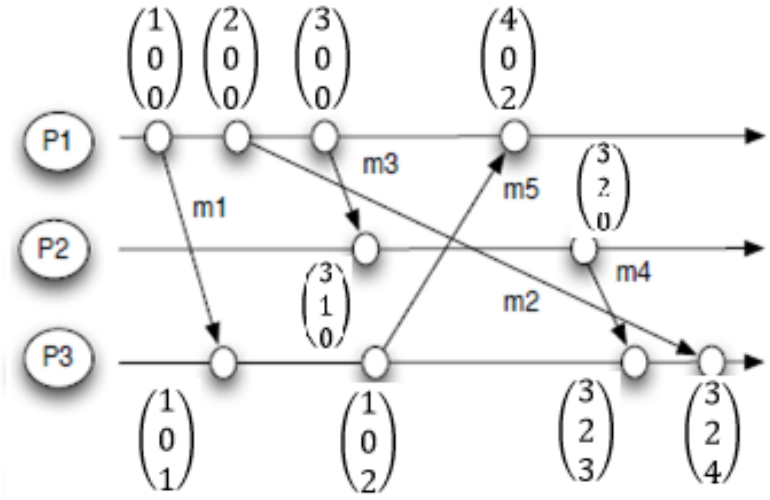


Figure 7.2 – *Horloge de Mattern.*

7.1.1.2. Exercice 02 : Coupure

La coupure C1 : est une coupure cohérente car le message traverse la coupure du passé vers le passé.
La coupure C2 : est une coupure incohérente car il y a un message qui traverse la coupure du futur vers le passé.

7.1.1.3. Exercice 03 : Horloge Matricielle

1. Au total, combien de messages ont été échangés ?
   - Il y a 4 messages.

2. Au total, combien d'événements ont eu lieu ?
   - Il y a 11 évènements : 4 emissions, 4 receptions et 3 évènements locaux.

3. Au total, combien d'événements locaux ont eu lieu ?
   - Il 3 évènement locaux. Un pour chaque site.

7.1.2. Exercices du chapitre 03

7.1.2.1. Exercice 01 : Exclusion Mutuelle

Réponse 01 :

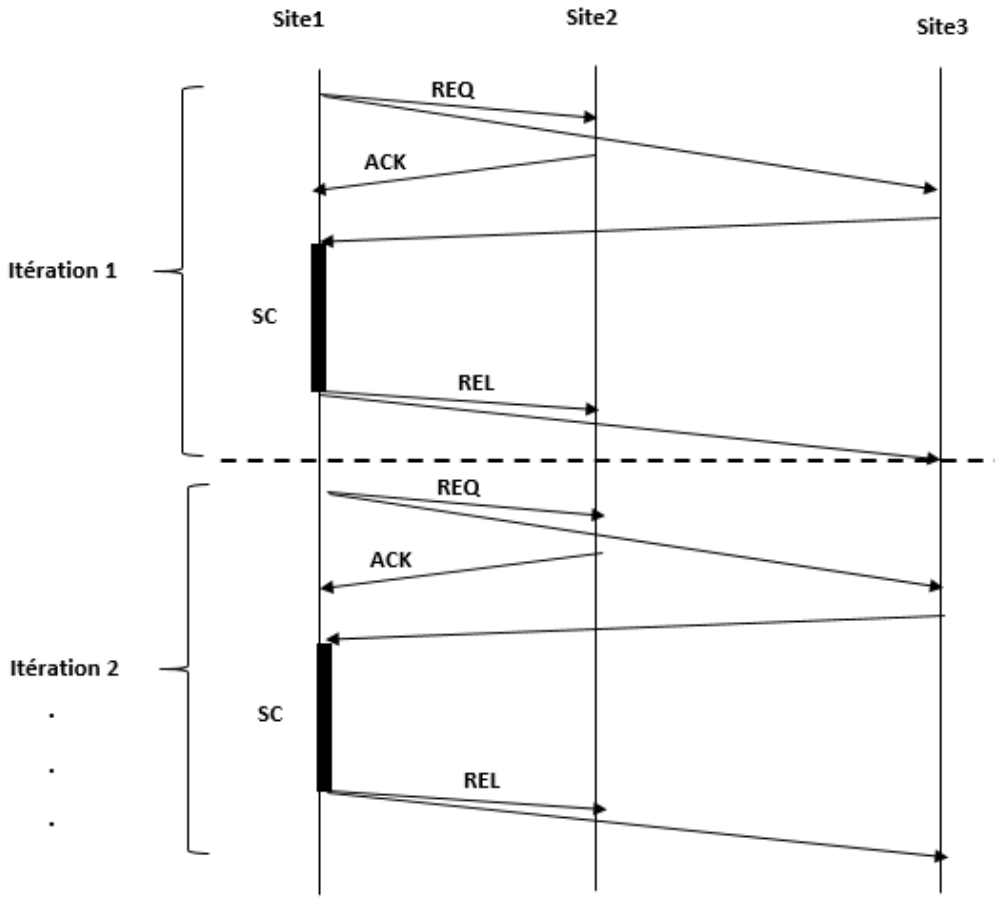


Figure 7.3 – *Demande itérative d'entrée à la section critique.*

Réponse 02 : pour chaque passage à la SC , le site 1 doit réitérer le message de Requête aux autres sites et doit attendre les Accusés de réception de tous ces sites. Ces messages (REQ et ACK) peuvent paraitre inutiles.
L'idée de l'Algorithme de Cavalho et Roucairol est donc de permettre au site 1 d'entrer directement à la SC s'il a déjà obtenu un Ack des autres sites tant qu'il n'ya aucun de ces sites qui n'a émis une requête pour entrer en SC. On revient à l'algorithme original lorsque l'un des sites souhaite de nouveau l'utilisation de la SC.

7.1.2.2. Exercice 02

Réponse 01 : L'algorithme utilisé est celui de Lamport. Justification : dans la file des requêtes au point B, on voit que P3 a sa propre liste d'ACK. L'algorithme Ricart-Agrawala ne fonctionne pas de cette manière.

Réponse 02 :

1. Le message envoyé de P2 vers P3 est : (ACK, 10, 2)
2. Le message envoyé de P3 vers P1 et P2 est : (REL, 12, 3)
3. Le message envoyé de P2 vers P1 et P3 est : (REL, 14, 2)

Réponse 03 : P2 entre en SC lorsque sa valeur d'horloge H2 est égale à 13. Justification : Le message envoyé à H2=14 (qui ne peut être qu'un REL) a fait supprimer la requête (REQ, X, 2) qui était dans la file de P3.

P2 entre en SC après P3 et avant P1. La date de sa requête X doit donc être postérieure à la date de la requête de P3 ("1") et antérieure à la date de la requête de P1("4"). X est donc égal à 2 ou 3.

Réponse 04 :

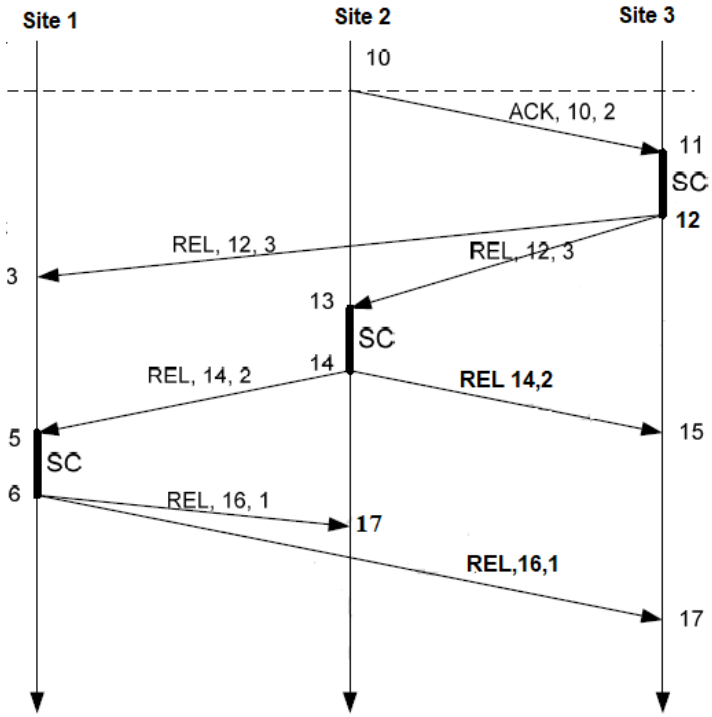


FIGURE 7.4 – *Evolution des sites.*

7.1.3. Exercices du chapitre 04

7.1.3.1. Exercice 01

Réponse 01 :

Un groupe est constitué de dix processus, avec des identificateurs numérotés de 0 à 9. Auparavant, le processus P9 était le coordinateur, mais il vient de tomber en panne. Le processus P6 est le premier à le remarquer.
- Application de l'algorithme de BULLY :

- Le processus P6 est le premier qui a detecter que le coordinateur est tombé en panne, il envoie donc des messages **ELECTION** à tous les processus supérieurs, à savoir P7, P8 et P9.
- Les processus P7 et P8 répondent tous deux par **OK**.
- Dès qu'il obtient la première de ces réponses, P6 sait que son travail est terminé, sachant que l'un des P7 ou P8 prendra la relève et deviendra le coordinateur.

- P7 et P8 envoient des messages uniquement aux processus dont l'identificateur est supérieur à lui même. P8 indique à P7 qu'il prendra le relais.
- À ce stade, P8 sait que P9 est mort et qu'il est le vainqueur.
- Lorsqu'il est prêt (P8) à prendre le relais, il annonce la prise de contrôle en envoyant un message **COORDINATOR** à tous les processus en cours d'exécution.

7.1.3.2. Exercice 02

Réponse 01 : voir (Fig.7.5)

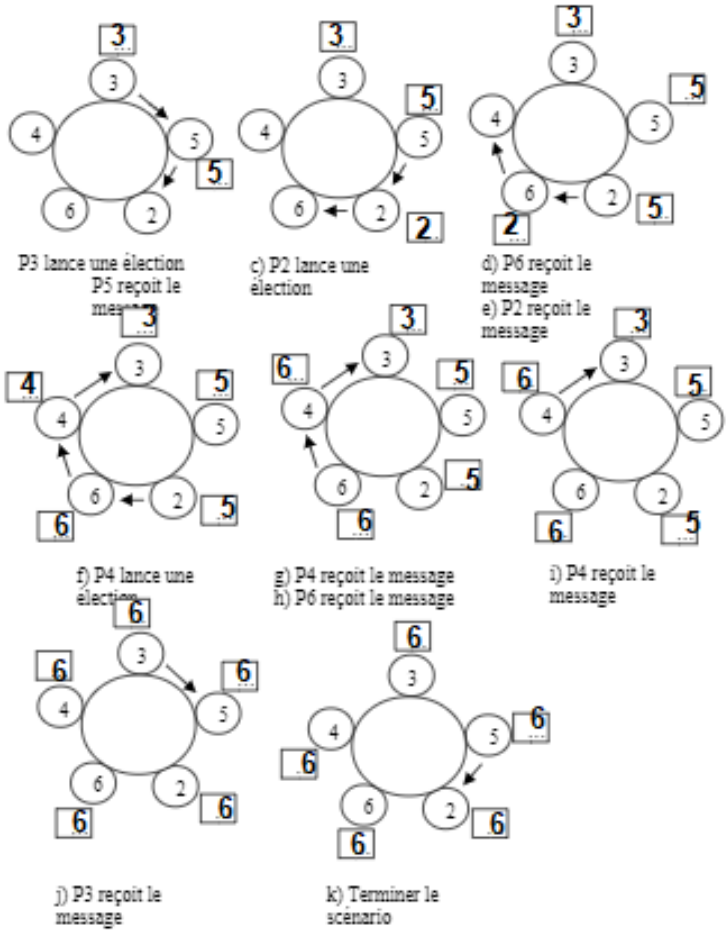


Figure 7.5 – *Scenario de Chang & Robert.*

7.1.4. Exercices du chapitre 05

7.1.4.1. Exercice 01

Réponse 01 : Dans un système réparti, un calcul peut être fait par plusieurs processus situés sur des sites différents. Pour savoir s'il y'a terminaison du calcul , il faut examiner l'état de tous les processus concernés et les éventuels messages en transit à travers le réseau.

Réponse 02 :
Etat actif : le processus est en exécution.
Etat passif : le processus est terminé , ou il est en attente d'un message.

Réponse 03 : Un processus passe de l'état actif à l'état passif : s'il a terminé son exécution ou s'il se met en attente d'un message.

Réponse 04 : Un processus passe de l'état passif à l'état actif : s'il reçoit un message.

Réponse 05 : Non. Si tous les processus sont dans un état passif, cela n'implique pas qu'il y'a terminaison de processus , car il se peut qu'il y'ait un message en transit dans le réseau qui n'a pas été reçu par le destinataire. En le recevant, le destinataire redeviendra actif, et il va falloir tout recommencer. Pour qu'il y'ait terminaison il faut que deux conditions soient réunies :
Tous les processus doivent être dans un état passif + Il n'ya aucun message en transit dans le réseau.

Réponse 06 : Le jeton contient le nombre de processus passifs comptés pendant sa circulation dans l'anneau.

Réponse 07 :
Blanc : Tous les processus visités sont passifs, et leur état n'a pas changé depuis le dernier passage du jeton.
Noir : Il y'a au moins un processus qui est passé de Passif à Actif , depuis le dernier passage du jeton.

Réponse 08 : Lorsque le processus (présent sur le site) reçoit le jeton, il attend dêtre passif avant d'envoyer le jeton. Il transmet le jeton en fonction du traitement qu'il vient de faire (a-t-il envoyé un message ou pas).
Si le processus n'a pas envoyé un message à un autre site
Alors Il incrémente la valeur du jeton ,

Il transmet le jeton sans modifier la couleur (blanche) de ce dernier
Sinon //Le processus a envoyé un message
Il réinitialise la valeur du jeton
Il transmettra le jeton en noir.

Réponse 09 : Il y'a terminaison si le jeton revient blanc avec le compteur égale à N (Tous les sites ont été visités)

7.1.4.2. Exercice 02

Gestion des messages :

P0 active P1 : defout=1 (P0), defin=1 (P1), Père= P0 (P1).

P1 active P3 : defout=1(P1), defin=1 (P3), Père=P1 (P3).

P0 active P2 : defout=2 (P0), defin=1 (P2), Père=P0 (P2).

P2 active P4 : defout=1 (P2), defin=1(P4), Père=P2 (P4).

P2 active P3 : defout=2 (P2), defin = 2(P3), Père= P1(P3)(occupé), Appelant={P2}.

Gestion des signaux :

P3 envoi un signal à P2 : defin=1(P3), defout=1 (P2), Appelant={ }.

P3 envoi un signal à P1 : defin=0 (P3), defout=0 (P1).

P4 envoi un signal à P2 : defin=0 (P4), defout=0 (P2).

P1 envoi un signal à P0 : defin=0 (P1),defout=1 (P0).

P2 envoi un signal à P1 : defin=0 (P2), defout=0 (P0).

conclusion : defin(P0)=defout(P0)=0 alors terminaison détectée.

Bibliographie

[1] George Coulouris,Jean Dollimore ,Tim Kindberg,Gordon Blair .(2014). "DISTRIBUTED SYSTEMS Concepts and Design". Livre.

[2] Ajay D. Kshemkalyani and Mukesh Singhal "Distributed Computing , Principles, Algorithms, and Systems". Livre.

[3] Flynn, M., Some Computer Organizations and Their Effectiveness, IEEE Trans. Comput., Vol. C 21, p. 948, 1972

[4] M. Singhal and N. Shivaratri (1994). Advanced Concepts In Operating Systems. Livre.

[5] Andrew S. Tanenbaum, Maarten Van Steen(2001). Distributed Systems : Principles and Paradigms.

[6] Chandy, K. and Lamport, L. (1985). Distributed snapshots : Determining global states of distributed systems. ACM Transactions on Computer Systems, Vol. 3, No. 1, pp. 63 ;75

[7] Lamport, L. (1978). Time, clocks and the ordering of events in a distributed system. Comms. ACM, Vol. 21, No. 7, pp. 558,65.

[8] Mattern, F. (1989). Virtual time and global states in distributed systems. In Proceedings of the Workshop on Parallel and Distributed Algorithms, Amsterdam, North-Holland, pp. 215,26.

[9] M.Raynal , M.Singhal (1996)." Logical Time : Capturing Causality in Distributed System". Livre

[10] Chang, F., Dean, J., Ghemawat, S., Hsieh, W., Wallach, D., Burrows, M., Chandra, T., Fikes, A. and Gruber, R. (2008). Bigtable : A distributed storage system for structured data. ACM Trans. on Computer Systems Vol. 26, No. 2, pp. 1,26

[11] L. Lamport. The mutual exclusion problem : Part I - A theory of interprocess communication. Journal of the ACM, 33(2) :313 ;326, 1986.

[12] L. Lamport. The mutual exclusion problem : Part II - Statement and solutions. Journal of the ACM, 33(2) :327,348 : 1986.

[13] Ricart, G. and Agrawala, A.K. (1981). An optimal algorithm for mutual exclusion in computer networks. Comms. ACM, Vol. 24, No. 1, pp.9,17.

[14] Garcia-Molina, H. (1982). Elections in distributed computer systems. IEEE Transactions on Computers, Vol. C-31, No. 1, pp. 48,59.

[15] Pease, M., Shostak, R. and Lamport, L. (1980). Reaching agreement in the presence of faults. Journal of the ACM, Vol. 27, No. 2, pp. 228 ;234

[16] O. Carvalho , G.Roucairol. "On Mutual Exclusion in Computer Networks". Communications of the ACM 26(2) :146,147.1983

[17] LE LANN G. Distributed systems : towards of a formal approach. IFIP Congress, North-Holland, (1977), pp. 155,160

[18] Misra, Jayadev, "Detecting Termination of Distributed Computations using Markers", PODC 1983

[19] E.W. Dijkstra and C.S. Sholten. Termination detection for diffusing computations.Information Processing Letters, 11(1) :1 ,4, 1980.

Table des figures