République Algérienne Démocratique et Populaire
Ministère de L'enseignement Supérieur et de la Recherche Scientifique
Université A/Mira de Béjaia
Faculté des Sciences Exactes
Département Informatique

Mémoire de Fin de cycle

En vue de l'obtention du diplôme Master recherche en Informatique

Spécialité : Réseaux et Systèmes Distribués

Thème

Proposition et implémentation d'un protocole de la diffusion atomique dans les systèmes distribués

---

Réalisé par :

M^elle^ AIDOUN Naoual.
M^elle^ IFETICENE Ryma Asma.

Devant le jury composé de :

Présidente :	M^me^ Ouyahia Samira.
Examinatrice :	M^me^ Boulfekhar Samra.
Promotrice :	M^me^ REBOUH Nadjette.
Co-Promotrice :	M^me^ BOUALLOUCHE Louiza.

Promotion 2013

Dédicaces

Je dédie ce modeste travail
À mes chers parents, qui sont la cause de mon existence dans cette vie,
pour leur soutient, leur patience et leur amour qui m'ont donné la force pour continuer mes études
À mes frère Nourddine, Boualem, Chihad et Djamel
À mes soeurs Sabiha et Fatiha
À mes coupines de chambre de 17 octobre : khalasse, Zamil, Bila et Biba
À tous mes meilleurs amis, dont la liste est longue du groupe de choc.
À toute ma grande famille, à tous mes amis et à tous mes enseignants.
et à toutes les personnes que j'ai connus et qui m'ont aidées

Naoual

Je dédie ce modeste travail à tout d'abord à mes très chers parents
Que leur présence à mes cotés illumine ma vie de joie et de bonheur
qui m'ont tant soutenu tout au long de mon parcours des études.
j'espère qu'ils seront satisfait de mon travail.
Aussi à mes chères sœurs nihad, souhila et lyna.
Un grand merci pour Juba et sa famille pour leur soutien et pour l'aide qui m'ont donné durant les préparations de ce travail.
À mes copine sarah, samia, et leila.
et à ma famille et tout mes amis.

Ryma

Remerciements

En tout premier lieu, nous remercions Allah le tout puissant, à la sagesse et au savoir infinis, " *Gloire à Toi Nous n'avons de savoir que ce que Tu nous as appris. Certes c'est Toi l'Omniscient, le Sage, le tout miséricordieux le très miséricordieux* " (Sourate al-Baqarah, verset 32).

Nous tenons en premier lieu à exprimer notre profonde reconnaissance à Mme N.REBOUH enseignante à l'université A. Mira de Béjaia, pour son encadremente, ses qualités tant scientifiques qu'humaines. Ce travail doit beaucoup à sa disponibilité permanente, sa rigueur scientifique et sa patience.

Nous remercions aussi Mme S. OUYAHIA de nous faire l'honneur de présider le jury de notre soutenance. Nos remerciements s'adressent aussi à Mme S. BOULFEKHAR pour avoir accepté la tache de juger ce travail.

Que tous ceux et toutes celles qui ont contribué de près ou de loin à l'aboutissement de notre travail trouvent ici nos sincères remerciements et l'expression de notre reconnaissance. Un grand merci pour nos collègues du Master pour les bons moments que nous avons passé ensemble en particulier les éléments du groupe de choc.

NawNaw & Ryma

Résumé

La diffusion atomique est l'un des problèmes d'accord qui représente une brique de base dans la conception des applications tolérantes aux fautes. Par définition, un protocole de la diffusion atomique doit garantir que tous les processus non défaillants remettent la même séquence ordonnée de messages. Dans ce mémoire, nous avons étudié les algorithmes de la diffusion atomique dans un système asynchrone. Il se diffèrent suivant le mécanisme d'ordre de messages ou l'approche de tolérance de fautes. Nous avons proposé une nouvelle solution au problème de la diffusion atomique. Elle s'exécute sur un réseau complètement connecté. Elle nécessite l'utilisation du jeton circulant sur un anneau logique avec le coordinateur tournant pour l'ordre de messages et les détecteurs de défaillances non fiables pour la tolérance aux fautes. Nous avons évalué les performances de ce protocole par la simulation en utilisant le simulateur *neko*, et le comparé avec d'autres protocoles existants dans la littérature.

Mots clés **:Diffusion atomique, Problème d'accord, Système distribué, Détecteurs de défaillances, jeton.**

Abstract

The atomic broadcast is an important agreement problem and represent an essential component for the conception of fault-tolerant distributed systems. By definition, an atomic broadcast protocol must guarantee that all non-faulty processes call the same ordered sequence of messages. In this work, we studied the atomic broadcast algorithms in a distributed system. They differ from each other according to their principles to order messages or their approaches to detect failures of processes. We proposed a solution to the problem of atomic broadcast, that must be executed on a fully connected network. The solution require the use of a token ring, a quorum of correct processes organised in a ring and a coordinator to order broadcast messages and failure detectors to detect crashed processes. We evaluated the performance of this protocol through simulation using the simulator Neko, and compared this protocol with other existing protocols in the literature.

Keywords **: Atomic broadcast, Agreement problem, Distributed algorithms, Failure detectors, Token.**

Table des matières

Liste des figures

Liste des tableaux

Liste des algorithmes

Liste des Acronymes

ABS	**A**nti **B**lockier **S**ystem.
FLP	**F**isher **L**unch **P**aterson.
GST	**G**lobal **S**tabilization **T**ime.
MAC OS	**M**acintosh **O**perating **S**ystem.

Introduction générale

EN seulement deux générations, la révolution numérique a conquis le monde : aujourd'hui, quasiment tous les êtres humains interagissent, directement ou indirectement, à un moment de leur vie, avec un système informatique. Les ordinateurs sont présents sur nos bureaux ; les systèmes informatiques gèrent les freins ABS (*(Anti blockier system)*) et le contrôle de la trajectoire de nos voitures et collectent des statistiques d'utilisation dans les ascenseurs afin d'anticiper les opérations de maintenance et de réparation. L'informatique est également utilisée dans les systèmes critiques, tels que les centrales nucléaires, le contrôle aérien ou les fusées spatiales. De plus, les systèmes informatiques sont non seulement omniprésents, mais de plus en plus souvent reliés en réseau.

Avec l'augmentation considérable de l'utilisation de systèmes informatiques, les besoins et attentes associés à ces systèmes ont aussi augmenté. En particulier, un des points critiques d'un système est sa disponibilité (la fraction du temps durant laquelle le système fournit un service aux utilisateurs) : les coûts et la publicité négative d'une panne du système (que ce soit un serveur web commercial ou un marché financier) sont souvent considérables.

La tolérance aux pannes est une des approches pour concevoir un système à haute disponibilité : un système tolérant aux pannes est conçu de telle manière à ce que la défaillance d'un de ses composants ne compromette pas sa fonctionnalité dans son ensemble. Les problèmes, dits d'accord, sont au coeur de la conception des applications tolérantes aux pannes. La diffusion atomique représente une classe des problèmes d'accord et une abstraction importante des calculs distribués tolérants aux fautes. Elle assure que l'ensemble des messages, diffusés par les différents processus, sera délivré par tous les processus de destination dans le même ordre. Plusieurs algorithmes de la diffusion atomique ont été proposés dans la littérature. Ces algorithmes peuvent être classés selon leurs mécanismes utilisés pour ordonner les messages (la circulation du jeton), ou bien selon leurs manières de détecter les défaillances des processus (l'utilisation du service de gestion du groupe ou bien l'utilisation des détecteurs de défaillances non fiables).

L'objectif de ce travail est de pallier aux inconvénients des protocoles de la diffusion atomique existants dans la littérature. Entre autre, l'utilisation d'un mécanisme d'ordre de messages puissant qui améliore la latence et assure la sûreté de ce système, ainsi que l'utilisation d'un mécanisme plus fiable de tolérance aux fautes, représenté par les détecteur de défaillances non fiables.

Ce mémoire est organisé de la manière suivante. Dans le premier chapitre nous introduisons des généralités sur les systèmes distribués et les différentes notions utilisées dans ce domaine. Par la suite, nous présenterons quelques problèmes d'accord, en relation avec notre domaine de recherche, en mettant l'accent sur le problème de la diffusion atomique. Dans le troisième chapitre, nous exhibons quelques algorithmes de la diffusion atomique qui appartiennent à différentes classes (algorithme à coordinateur, algorithme à détecteurs de défaillances, algorithme à consensus), nous concluons ce chapitre avec une comparaison théorique de ces protocoles selon plusieurs critères. Dans le quatrième chapitre, nous présenterons notre proposition pour la diffusion atomique, qui représente une nouvelle solution pour ce problème. Elle nécessite l'utilisation du jeton circulant sur un anneau logique avec le coordinateur tournant pour l'ordre de messages et les détecteurs de défaillances non fiables pour la tolérance aux fautes. Une simulation, par un simulateur existant pour les systèmes distribués appelé neko, sera faite pour évaluer les performances du protocole proposé, suivie d'une comparaison pratique des protocoles présentés dans le chapitre précédent avec notre protocole. Nous clôturons par une conclusion et des perspectives.

Chapitre 1

Généralités sur les systèmes distribués

1.1 Introduction

Un système distribué est défini comme étant un ensemble de machines autonomes connectées par l'intermédiaire d'un réseau. Chaque machine exécute un ou plusieurs processus et peut communiquer avec les autres machines. Tous ces processus coordonnent leurs activités de telle manière que, de l'extérieur, le système entier peut-être perçu comme un tout capable d'accomplir une tâche bien déterminée.

Dans ce chapitre nous présenterons des généralités sur les systèmes distribués à savoir les différentes entités composant ce système,la manière dont elles interagissent, les types de défaillances que peuvent subir, et leurs manière de synchronisation (synchrone ou asynchrone).

1.2 Les entités

Un système distribué regroupe trois composantes essentielles définies de la manière suivante :

- Les processus : Ils décrivent les comportements des programmes s'exécutant à l'intérieur du système ;
- Le modèle de communication : Il décrit le mode d'interaction ou de communication entre les processus ;
- L'architecture d'interconnexion : Elle décrit la manière d'organisation des processus (la topologie).

1.2.1 Les processus

Les processus sont des programmes en cours d'exécution. Ils communiquent entre eux par l'échange de messages, à travers des canaux de communication, ou par l'accès à une variable commune dans une mémoire partagée. Par définition, un processus qui a un mauvais comportement ou s'écarte de ses spécifications initiales durant l'exécution d'une tâche est considéré défaillant, autrement, il est dit correct. Il existe plusieurs types de défaillances de processus [REB07] :

1.2.1.1 Défaillance par arrêt définitif

Le processus se comporte conformément à sa spécification jusqu'à ce qu'il subisse une défaillance franche, après laquelle, il arrête toutes ses activités (traitement, envoi et réception de messages). Son arrêt est définitif, il ne peut plus participer aux tâches du système [FLP85].

1.2.1.2 Défaillance par omission

Dans ce type de défaillance, un processus fautif peut omettre certaines tâches de sa sépicification. Ces tâches sont liées principalement à l'envoi et la réception de messages. Pratiquement, cela peut correspondre à une perte de messages [PT86].

1.2.1.3 Défaillance temporelle

Une défaillance temporelle, dite aussi de performance, est traduite par le fait que suite à une demande de service, le processus fournit la réponse soit trop tôt soit trop tard. De ce fait, le processus ne respecte pas les contraintes temporelles spécifiées à une tâche ou plusieurs tâches.

1.2.1.4 Défaillance arbitraire

Elle est appelée aussi défaillance byzantine où tout comportement s'écartant des spécifications est qualifié de comportement byzantin. Ce type de défaillance est le plus général et le plus sévère parmi ceux déjà cités (voir la figure 1.1) [PDC02]. L'étude de ce type de défaillance est très utile pour la construction des systèmes sécurisés [Mar04].

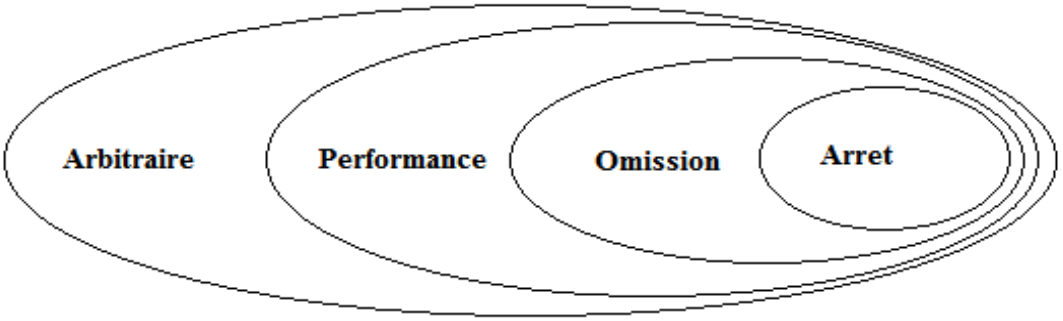

FIGURE 1.1 – Classification des défaillances de processus

1.2.2 Le mode de communication

La communication entre les processus d'un système distribué peut se faire soit par une mémoire partagée (shared memory) [Abr88], soit par l'échange de messages (message-passing) [CM84].

1.2.2.1 Communication par mémoire partagée

Dans ce mode de communication, les processus communiquent entre eux grâce à une mémoire commune à travers laquelle ils peuvent échanger des données. Une donnée déposée par un processus dans cette mémoire est ainsi aperçue et manipulée par tous les processus la partageant. L'accès à cette ressource commune est géré par deux primitives de lecture/écriture :

- **Lire ()** : Permet de lire une donnée stockée dans une zone de la mémoire partagée.
- **Ecrire ()** : Permet d'écrire une donnée dans une zone de la mémoire partagée.

1.2.2.2 Communication par échange de messages

La communication, dans ce mode, est éffectuée grâce à un mécanisme d'échange de messages. Ce modèle est caractérisé par l'absence totale d'horloge globale et de mémoire commune. L'échange de messages est effectué par les deux primitives de communication suivantes :

- **Envoyer (msg,** p_j**)** : Permet d'envoyer le message msg au processus p_j .
- **Recevoir (msg)** : Permet de recevoir un message msg déjà envoyé par un processus donné.

Ce mode dispose d'une autre primitive de communication destinée à la communication dans un groupe :

- **Diffuser (msg)** : Permet d'envoyer un message msg à tous les processus du système.

l'échange de messages se fait via des canaux de communication suivant une organisation bien définie des processus. Les canaux de communication, à leurs tours, sont sujets à plusieurs types de défaillances :

- La destruction du canal ;
- La perte de messages ;
- La duplication de messages ;
- La corruption de messages.

Suivant ces défaillances, les canaux de communication peuvent être classés en :

- **Canaux fiables** : Si un processus p_i envoie un message msg à un processus p_j, alors, eventuellement, msg sera reçu par p_j. Pratiquement, les canaux fiables sont implémentés en utilisant des techniques de retransmission de messages [1] [Tio08],[Mou05].

- **Canaux quasi-fiables** : Un processus correct p_i envoie un message msg à un processus correct p_j, alors p_j finissera par recevoir le message msg. Dans la pratique, ce type de canaux est le plus recommandé pour la conception des algorithmes distribués car il suppose que l'émetteur et le récepteur soient corrects.

- **Canaux fiables FIFO** : Les canaux sont supposés fiables et en plus l'ordre de livraison de messages est celui de l'émission de ces messages entre deux processus[Gre02].

- **Canaux fiables avec fautes de performances** : Ils supposent des canaux fiables avec la prise en compte des délais de transmission de messages. En effet, avec cette hypothèse, on n'assure nullement la réception des messages en un temps borné.

- **Canaux avec des pertes équitables** : Si un processus p_i envoie à un processus correct p_j un message msg une infinité de fois, alors p_j reçoit msg une infinité de fois. Dans ce type de canaux, on tolère les pertes de messages.

- **Canaux non fiables** : Des pertes de messages se produisent sans que l'on puisse faire la moindre hypothèse concernant leurs occurrences.

1. Le message est retransmis par l'émetteur jusqu'à ce que le récepteur, supposé correct, le reconnaisse

1.2.3 La topologie

La topologie d'un système distribué définit la manière d'organisation des processus. Elle fait référence à l'ensemble des processus qui la compose et leurs interconnexions. Il existe plusieurs types de topologies dans la littérature, qui sont classifiées en trois larges catégories : (1) les réseaux point- à- point (Unicast), (2) les réseaux de diffusion (Broadcast), représentant notre domaine d'étude,(3) et les réseaux mixtes qui permettent la diffusion à un nombre limité de processus (multicast)[Mou05]. L'arrangement physique des éléments d'un système est appelé topologie physique qui peut être :

- Une topologie complètement connecté (Figure 1.2).
- Une topologie en bus (Figure 1.3) ;
- Une topologie en étoile (Figure 1.4) ;
- Une topologie en anneau (Figure 1.5).

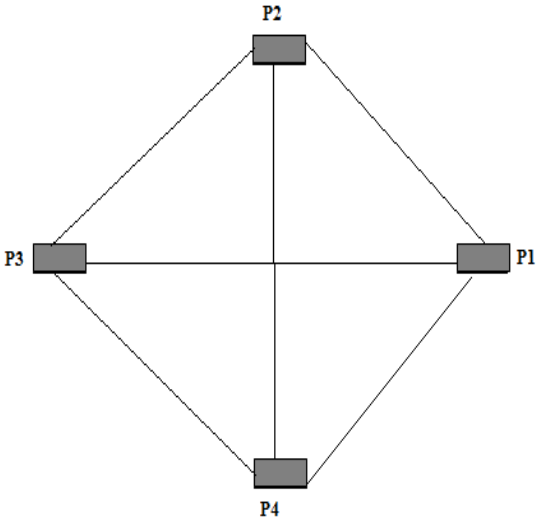

FIGURE 1.2 – Topologie complètement connecté

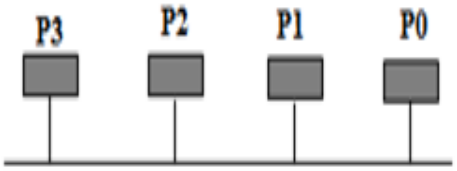

FIGURE 1.3 – Topologie en bus

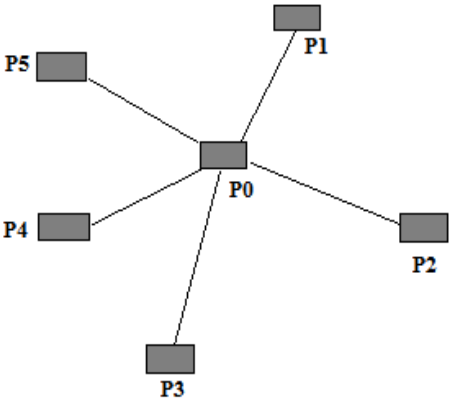

FIGURE 1.4 – Topologie en étoile

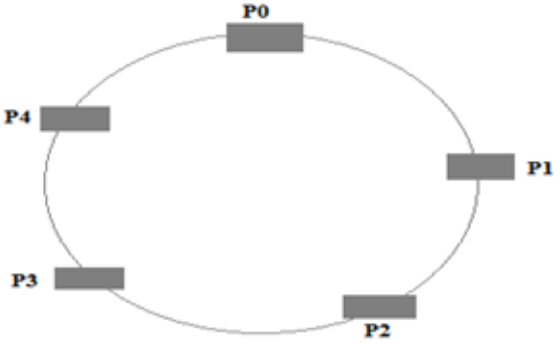

FIGURE 1.5 – Topologie en anneau

La topologie physique (câblage et organisation dimensionnelle) est complètement différente de la topologie logique. La topologie logique représente la façon suivant laquelle les

données transitent entre les processus via les supports physiques. Les topologies logiques les plus courantes sont Ethernet,Token Ring et FDDI [Pil06].

1.3 La synchronie

La synchronie d'un modèle de système dépend des hypothèses temporelles faites sur le comportement des processus et des canaux de communication. Plus précisément, on considère généralement trois paramètres principaux [Urb03] :

1. La vitesse relative des processus : elle traduit le temps nécessaire pour l'exécution d'une tâche par un processus ainsi les rapports de vitesses existant entre le plus rapide et le plus lent processus dans le système ;
2. Le délai de transmission des messages : Il quantifie le temps qui s'écoule entre l'émission et la réception des messages ;
3. Les dérives des horloges.

Le synchronisme d'un système est défini en termes de bornes faites sur ces trois paramètres [Urb03]. Suivant ces bornes, on distingue plusieurs modèles de temps :

- Le modèle synchrone ;
- Le modèle asynchrone ;
- Le modèle partiellement synchrone.

1.3.1 Le modèle synchrone

C'est le modèle temporel le plus simple pour définir et raisonner sur la correction d'algorithmes répartis, appelé aussi modèle à temps borné : le temps maximum nécessaire pour interroger et recevoir la réponse d'un processus non défaillant est strictement borné [ACT99]. Il s'agit d'un modèle très puissant car il permet d'utiliser des temporisations (timeout) pour détecter de façon non ambiguë l'arrêt ou le retard de réaction d'un processus distant [Mou05]. Ce modèle est approprié pour des applications critiques qui doivent garantir des propriétés temps réel de terminaison, même en présence de fautes. Par contre, il impose un ordonnancement temps réel strict de la communication et du traitement, et repose sur l'hypothèse que la borne sur le temps de communication est toujours respectée.

1.3.2 Le modèle asynchrone

À l'extrême opposé, le modèle asynchrone ou « sans temps » fait l'abstraction totale de la notion de temps et donc ne suppose aucune borne temporelle [REB07]. Dans ce modèle, on garantit qu'un message transmis sur un canal sera inéluctablement délivré au processus destinataire (la littérature parle de canal « fiable » ou « équitable »), mais à un instant non défini (puisque le modèle ne comporte aucune notion de temps). Les algorithmes conçus selon ce modèle sont attractifs car leur comportement logique est totalement indépendant des performances des réseaux et des systèmes mis en jeu. Malheureusement, il a été démontré que certains problèmes fondamentaux de tolérance aux fautes (par exemple, le consensus) n'admettent pas de solution déterministe avec ce modèle (résultat d'impossibilité de[FLP85]).

1.3.3 Le modèle partiellement synchrone

La simplicité de la conception et de l'implémentation des protocoles dans un système asynchrone rend ces systèmes plus général et plus attractifs par rapport au système synchrone. Cependant, la difficulté voir l'impossibilité de distinguer un processus correct lent d'un processus fautif conduit à une situation d'impossibilité [FLP85]. Les auteurs de [CT96], [DDS87], [DLS88] ont défini un autre modèle temporel, le modèle partiellement synchrone. Le modèle partiellement synchrone est un modèle intermédiaire entre les deux modèles d'extrémité synchrone et asynchrone et qui contourne l'impossibilité déjà introduite. Les travaux de Dolev et Al. [DDS87] ont permis de définir 32 modèles partiellement synchrone grâce à cinq critères principaux dont chacun peut prendre deux valeurs soit vrai "favorable" ou faux "défavorable". Parmi ces critères, le délai maximum de transmission d'un message, qui peut être borné et connu (favorable) ou non borné (défavorable).

En plus de ces modèles temporels, il existe d'autres modèles intermédiaires dans la litérature qui sont basés sur des hypothèses complètement différentes et qui reflètent certaines réalités (le modèle asynchrone temporisé)[CF99] .

Dans ces cas, il est possible de garantir le respect d'invariants de sûreté, mais toute garantie de progrès (par exemple, pour assurer un consensus) est conditionnée par le fait qu'au moins une majorité de processus se comporte de façon synchrone pendant « suffisamment » de temps [REB07].

1.4 Conclusion

Dans ce chapitre, nous avons présenté des généralités sur les systèmes distribués. Nous avons défini des entités composants ces systèmes à savoir les processus, les canaux de communication et la manière dont ces processus sont reliés entre eux. Nous avons détaillé les différents types de défaillances que les entités peuvent subir et la manière par laquelle ces dernières se synchronisent dans le temps. Nous nous intéressons dans ce mémoire au modèle purement asynchrone caractérisé par l'absence totale de bornes sur les vitesses relatives des processus, les délais de transmission des messages et les dérives des horloges et nous tolérons les défaillances franches des processus.

Chapitre 2

Les problèmes d'accord

2.1 Introduction

Dans de nombreuses applications, il est indispensable que les processus aient une vision unanime de la progression du calcul pour lequel ils coopèrent. Pour élaborer cette vision commune, chaque processus doit participer à l'exécution d'un protocole d'accord en fournissant au départ sa vision locale de l'état du système. Durant l'exécution du protocole d'accord, tous devront progressivement converger vers une valeur unique obtenue à partir de l'ensemble des valeurs initiales. Au sein de la classe des problèmes d'accord, le problème du consensus fait figure d'exemple. Dans ce cas particulier, la valeur unanimement décidée doit être une des valeurs proposées. Différents autres problèmes d'accord (validation atomique, diffusion atomique,etc.) peuvent être résolus en utilisant une solution au problème du consensus comme brique de base. Malheureusement, il a été démontré que le problème du consensus n'admet aucune solution déterministe si le système est asynchrone et non-fiable. Pour être résolu, le problème doit être affaibli (terminaison du protocole ou unicité de la décision non garantie) ou alors les hypothèses caractérisant l'environnement doivent être renforcées (rajout de propriétés de synchronie).

Dans ce chapitre, nous présenterons quelques problèmes d'accord liés à notre étude. Par la suite, nous exhiberons quelques résultats de recherche existants, dans la littérature sur la solvabilité du consensus, l'impossiblité de FLP, et les différentes approches pour la contourner.

2.2 Les problèmes d'accord

Dans cette partie, nous présenterons quelques problèmes d'accord : la diffusion fiable, la diffusion atomique, l'élection d'un leader et leur brique de base ; le consensus.

2.2.1 La diffusion fiable

La diffusion fiable est une primitive de communication. Elle garantit que tous les processus corrects délivrent un message diffusé [HT94]. C'est la politique de "tout ou rien".

l'implémentation de la diffusion fiable se fait grâce à deux primitives :

- **R-Broadcast (msg)** : Elle envoie un message msg à tous les processus.
- **R-Deliver (msg)** : Elle permet la consommation d'un message msg émis par l'invocation de R-Broadcast(msg).

La diffusion fiable est définie par les propriétés suivantes [HT94] :

- **La validité** : Si un processus correct diffuse un message msg, alors il finira par être délivré par tous les processus corrects (en un temps fini).
- **L'accord** : Si un processus délivre un message de diffusion msg alors tous les processus corrects délivrent msg ultimement.
- **L'intégrité** : Pour chaque message de diffusion msg, tout processus délivre msg au plus une fois, et seulement si msg a été antérieurement diffusé par l'émetteur de msg (pas de duplication ni de création de messages).

2.2.2 La diffusion atomique

La diffusion atomique, une primitive de communication, est une extension de la diffusion fiable. En plus de s'assurer que tous les processus reçoivent le même nombre de messages, il faut s'assurer également que les processus les reçoivent dans le même ordre. C'est la primitive clé pour la mise en oeuvre d'un serveur répliqué de manière active,car elle contribue à préserver le déterminisme des copies [Gre02]. La figure 2.1 illustre une telle abstraction.

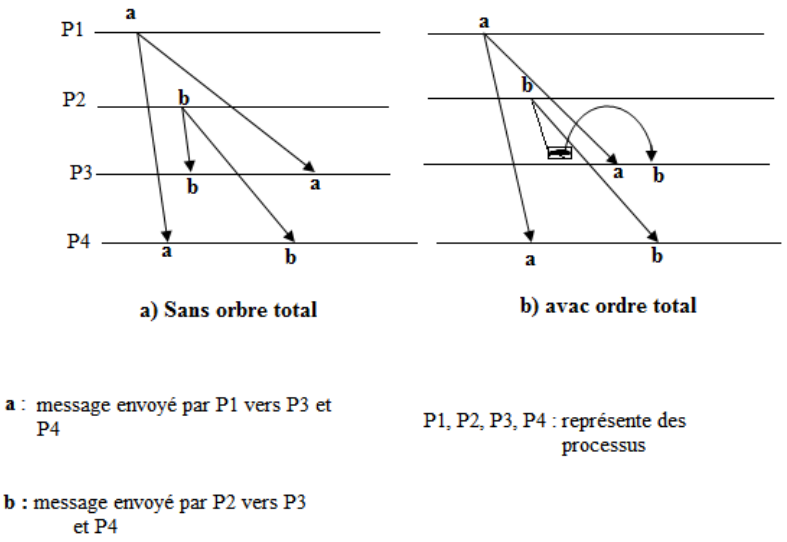

FIGURE 2.1 – L'abstraction de la diffusion atomique

Les processus P1 et P2 envoient les messages a et b vers les processus P3 et P4 , dans le cas ou sans ordre totale les messages {a,b} ont été délivré par les processus P3 et P4 dans un ordre différent (message b puis a par P3, tandis que pour P4 c'est a puis b).

Par contre pour le cas avec ordre totale les deux processus P3 et P4 délivrent les messages {a,b} dans le même ordre (a puis b).

l'implémentation de la diffusion atomique se fait grâce à deux primitives :

- **A-Broadcast (msg)** : Elle envoie un message msg à tous les processus.
- **A-Deliver (msg)** : Elle permet la consommation d'un message msg émis par l'invocation de A-Broadcast(msg).

La diffusion atomique est spécifiée par les propriétés de la diffusion fiable, en plus de la propriété de l'ordre total suivante [HT94] :

- **Ordre total** : Si deux processus corrects p_i et p_j délivrent deux messages msg et msg' alors p_i délivre msg avant msg', si et seulement si, p_j délivre msg avant msg'.

2.2.3 L'élection d'un leader

L'élection d'un leader est un problème fondamental dans les calculs distribués. Il a fait l'objet de plusieurs travaux [CHT96], [CF99], [CBS00]. Ce problème peut être utilisé à

titre d'exemple, quand la défaillance d'un noeud provoque la perte d'un jeton dans un algorithme d'exclusion mutuelle, les autres noeuds doivent élire un nouveau leader qui sera chargé de regénérer le jeton. L'élection d'un leader est aussi utile dans les protocoles de communication de groupes afin de choisir un nouveau coordinateur lorsque la composition du groupe change [REB07].

Le problème de l'élection d'un leader est défini par les deux propriétés suivantes :

- **Terminaison** : Il existera inéluctablement un leader.
- **Accord** : Il n'y a jamais plus d'un leader.

2.3 Le consensus

Le consensus constitue un problème fondamental de la tolérance aux fautes répartie, comme il est au centre des problèmes d'accord [GS01]. Dans sa forme de base, chaque processus *propose* une valeur et doit *décider* une des valeurs initiales de manière *unanime* et *irrévocable*. Plus formellement, tout algorithme de consensus doit vérifier les trois propriétés suivantes [CT96], [FLP85] :

- **Validité** : Si un processus décide une valeur v alors v a été proposée par au moins un processus.
- **Accord** : Au plus une valeur est décidée.
- **Intégrité** : Chaque processus décide au plus une fois.
- **Terminaison** : Tout processus correct décide ultimement.

Le consensus peut être implémenté à l'aide de deux primitives :*propose(v)* et *decide(v)*. Lorsqu'un processus p_i exécute *propose(v)*, on dit que p_i propose v. De même, chaque fois que p_i exécute *decide(v)*, on dit qu'il décide la valeur v [EKW07].

Chandra et Toueg [CT96] ont montré que le consensus et la diffusion atomique sont deux problèmes équivalents. En d'autres termes, une solution à l'un implique automatiquement une solution à l'autre.

Sabel et Marzullo [SM95] ont montré que le problème de l'élection d'un leader est réductible au problème du consensus. Informellement, une utilisation du consensus pour résoudre l'élection d'un leader consiste à effectuer une décision sur le leader à élire.

2.3.1 Les variantes du problème de consensus

La résolution du consensus dans un système distribué peut se faire de plusieurs méthodes, selon le modèle de système à considérer. De ce fait, il existe plusieurs variantes à ce problème qui se diffèrent les unes des autres par au moins une propriété, du problème initial, déjà définie.

2.3.1.1 Le consensus probabiliste

Le consensus probabiliste a été introduit par Ben Or [BO83]. Le caractère déterministe du consensus est supprimé, les processus ne décident pas avec une certitude mais avec une probabilité égale à 1. Donc, dans cette instance du consensus, la propriété de terminaison a été affaiblie et remplacée par :

- **R-terminaison (random-termination)** : Tous les processus corrects décident avec une probabilité égale à 1.

2.3.1.2 Le consensus uniforme

Dans la déffnition du consensus, les processus défaillants peuvent decider différemment des processus corrects. Cette définition ne convient pas à certaines applications qui demandent un niveau de sûreté très élevé. Pour interdire cette situation, le consensus uniforme [HT93] a été introduit et se diffère du consensus de base par la propriété d'accord suivante :

- **Accord uniforme** : Si un processus décide une valeur v, alors tous les processus décident v.

2.3.1.3 Le consensus ensembliste (k-consensus)

Le k-consensus (1<k< n) est plus connu sous sa dénomination anglaise " k-set agreement". Le k-consensus consiste en un accord sur au plus k valeurs différentes [Mou03]. Il a pour objectif l'affaiblissement de la propriété d'accord, qui est remplacée par :

- **k-Accord** : Il existe au maximum k valeurs différentes décidées par les processus.

Ce problème a été introduit par Chaudhuri [Cha90] pour montrer que plus la propriété d'accord est faible (plus k est grand), plus la tolérance aux défaillances est forte.

2.4 Le résultat d'impossibilité de FLP

Il a été démontré que le problème de consensus est impossible à résoudre de manière déterministe dans un système distribué asynchrone, pouvant être sujet à une seule défaillance de processus. Ce résultat est connu comme étant le résultat d'impossibilité de FLP (initiales des noms de ces auteurs Fisher, lynch et Paterson) [FLP85]. Intuitivement, ceci est dû à l'impossibilité de distinguer de manière déterministe un processus défaillant d'un processus lent ou d'un processus avec lequel les communications sont lentes. Cependant, différentes alternatives permettent de contourner cette impossibilité.

2.5 Contourner le résultat d'impossibilité

Pour contourner le résultat d'impossibilité de FLP, plusieurs approches ont été proposées :

- Une approche basée sur l'affaiblissement de certaines propriétés du consensus (par exemple : le consensus probabiliste,etc.,déjà présentés).
- Une approche basée sur la contrainte par condition, proposée par Mostéfaoui et Al. [AMR01],[MRR02], qui consiste à identifier des conditions, relatives aux valeurs proposables, autorisant une décision directe des processus.
- Une approche basée sur l'incorporation des hypothèses temporelles, soit à l'intérieur du système (les modèles partiellement synchrones [CT96], [DDS87], [DLS88], etc.) soit dans des boites noires introduites par Chandra et Toueg [CT96], et qui consistent à enrichir le système par un mécanisme de détection de défaillances, appelées détecteurs de défaillances non fiables.

2.6 Les détecteurs de défaillances

Les détecteurs de défaillances non fiables ont été introduits par Chandra et Toueg [CT96]. Ce sont des oracles qui donnent aux processus des indications sur les défaillances des autres processus dans le système, sous forme de listes. Ces indications peuvent être plus ou moins exactes, car les détecteurs de défaillances peuvent suspecter un processus correct comme ils peuvent ne pas suspecter un processus réellement défaillant, d'où leurs nom de

détecteurs de défaillances non fiables.

Comme le montre la figure 2.2, à chaque processus p_i est attaché un module de détection de défaillances , lui permettant de mettre à jour sa liste de processus suspectés d'être défaillants $suspected_i$.

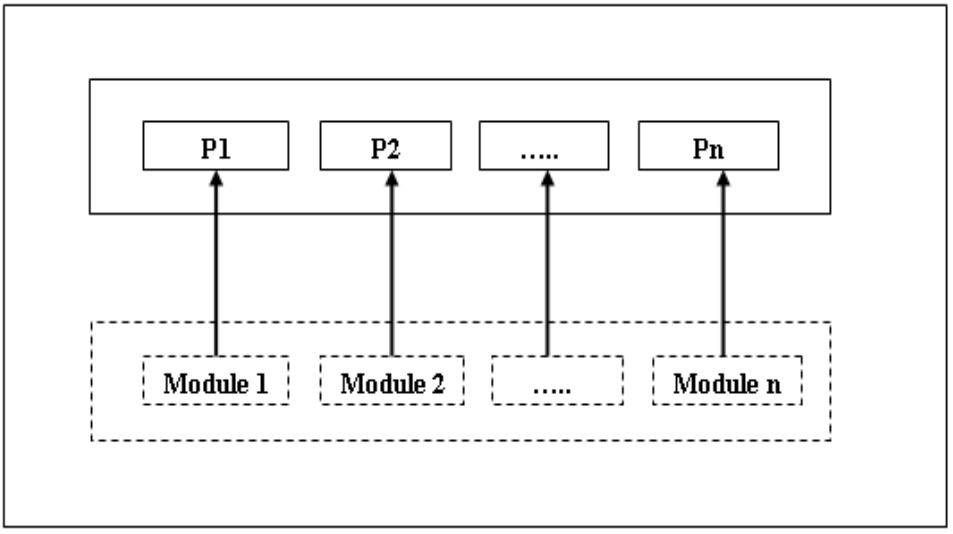

FIGURE 2.2 – Les détecteurs de défaillances

2.6.1 Propriétés d'un détecteur de défaillances

Les détecteurs de défaillances de Chandra et Toueg [CT96] sont caractérisés par deux propriétés abstraites : la complétude et la précision (exactitude).

- **La complétude** : C'est une propriété de vivacité [1] qui assure que les processus défaillants

1. Un événement désirable finira par arriver. Exemples : un message sera délivré à son destinataire, une ressource demandée sera rendue disponible, un algorithme se termine (indécidable dans le cas général, etc.)

finiront par être suspectés. Deux propriétés de complétude sont définies :

- **Complétude forte** : Tout processus incorrect finira par être suspecté de façon permanente par tout processus correct.
- **Complétude faible** : Tout processus incorrect finira par être suspecté par au moins un processus correct.

- **La précision** : C'est une propriété de sûreté [2] qui restreint les suspicion erronées sur des processus corrects. Il existe quatre propriétés de précision :
  - **Précision forte** : Aucun processus correct n'est suspecté avant de tomber en panne.
  - **Précision faible** :Il existe au moins un processus correct qui n'est jamais suspecté.
  - **Précision ultime forte** : Il existe un instant à partir duquel, aucun processus correct n'est plus jamais suspecté.
  - **Précision ultime faible** : Il existe un instant à partir duquel, il existe au moins un processus correct qui n'est plus jamais suspecté.

2.6.2 Classification

A partir des propriétés de complétude et de précision vues précédemment, Chandra et Toueg [CT96] ont défini huit différentes classes de détecteurs de défaillances. Ces classes sont représentées dans le tableau suivant :

	Précision forte	**Précision faible**	**Précision ultime forte**	**Précision ultime faible**
Complétude forte	Parfait P	fort S	Ultimement parfait ◊P	Ultimement fort ◊S
complétude faible	Q	Faible W	◊Q	Ultimement faible ◊W

TABLE 2.1 – Les classes de détecteurs de défaillances.

Chandra et Toueg [CT91] ont montré qu'il existe une équivalence entre certaines classes de détecteurs en exhibant une transformation permettant d'avoir la propriété de complétude forte à partir de la complétude faible. Il en résulte, donc, des équivalences entre les différentes classes de détecteurs de défaillances (P ≡ Q, S ≡ W, ◊P ≡ ◊Q, ◊S ≡ ◊W).

2. Un événement indésirable n'arrivera jamais. Exemples : violation de l'exclusion mutuelle, incohérence des données, etc.

2.7 Conclusion

Dans ce chapitre, nous avons présenté les problèmes d'accord en relation avec notre domaine d'étude. nous avons constaté que ces problèmes sont liés les uns aux autres (la diffusion atomique et le consensus sont deux problèmes équivalents). Cela signifie q'une solution à un problème peut être utilisée pour résoudre l'autre problème. Ainsi, nous avons remarqué qu'une solution à un problème d'accord peut être faite en collaborant plusieurs problèmes (une solution au problème du consensus peut se baser sur une combinaison du problème de l'élection d'un leader et la diffusion fiable, etc.). Cependant, ces solutions proposées doivent tenir compte des contraintes de conception (modèle de système, etc.) et des moyens fournis(détecteurs de défaillances, etc.). Dans le chapitre suivant, nous nous intéressons au problème de la diffusion atomique dans un système distribué asynchrone.

Chapitre 3

État de l'art sur les protocoles de la diffusion atomique

3.1 Introduction

La diffusion atomique est une importante abstraction dans les applications distribuées tolérantes aux fautes. Elle assure que les messages diffusés par différents processus seront délivrés par tous les processus de destination dans le même ordre. Plusieurs solutions ont été proposées pour résoudre ce problème. Ces solutions peuvent être classées suivant le mécanisme d'ordre de messages : l'exécution du consensus sur une séquence de messages, la circulation du jeton, etc. Cette dernière est la plus importante alternative. Dans les protocoles basés sur ce mécanisme, un jeton circule autour d'un ensemble de processus et le détenteur du jeton a le privilège d'ordonner les messages diffusés. De plus, dans d'autres cas, seulement le détenteur du jeton peut diffuser les messages.

Cependant, le mécanisme d'ordre n'est pas le seul mécanisme clé d'un protocole de la diffusion atomique. Le mécanisme utilisé pour tolérer les défaillances est une autre importante caractéristique pour ces algorithmes. Si on considère les systèmes asynchrones avec crash des processus, les deux principaux mécanismes de tolérance aux pannes dans le contexte des algorithmes de la diffusion atomique sont (1) les détecteurs de défaillances non fiables [CT96] et (2) le service de gestion de groupe [GCV01].

Dans ce chapitre, nous présenterons quelques algorithmes de référence pour la diffusion atomique, nous comparerons ces protocoles suivant plusieurs critères, et nous achèverons le chapitre par une conclusion.

3.2 Les algorithmes de la diffusion atomique

Les protocoles de la diffusion atomique sont au cœur de plusieurs problèmes d'accord dans les systèmes distribués, tels que la validation atomique [Gra78], la gestion des vues dans le service de gestion de groupe [FClD90], [RB91], etc. Dans ce qui suit, nous présenterons quelques protocoles de la de la diffusion atomique.

3.2.1 Un protocole basé sur le détecteur de défaillances $\diamondsuit S$

Dans ce travail, Chandra et Toueg [CT96] ont proposé un protocole qui résout le problème de la diffusion atomique dans un système asynchrone. Ils ont montré comment transformer n'importe quel algorithme du censensus en un algorithme de La diffusion atomique. l'algorithme de la diffusion atomique résultant tolère autant de défaillances que l'algorithme de consensus utilisé.

3.2.1.1 Principe de l'algorithme

Les auteurs dans [CT96] ont considéré un système asynchrone composé d'un ensemble de n processus $\{p_0,...,p_n\}$ reliés entre eux par des canaux de communication fiables, composants un réseau complètement connecté. Ces processus sont sujets à des défaillances par arrêt définitif.
Le système est augmenté d'un détecteur de défaillances non fiable $\diamondsuit S$ et tolère une majorité de processus corrects (les mêmes propriétés du protocole de consensus).

Le protocole de la diffusion atomique, présenté dans ce papier, utilise des exécutions consécutives du consensus. Intuitivement, la k^{ime} exécution du consensus est convoquée pour décider la k^{ime} sequence de messages qui doit être atomiquement délivrée. Les processus distinguent ces exécutions en affectant un compteur k aux messages appartenant à la k^{ime} exécution du consensus. La convocation de la k^{ime} exécution du consensus se fait grâce aux deux primitives ***propose(k,-)*** et ***decide(k,-)***.
En plus du consensus, ce protocole utilise la diffusion fiable. Il fait appel aux deux primitives de la diffusion fiable ***R-broadcast(m)*** et ***R-deliver(m)***. Pour soulever l'ambiguïté entre la diffusion fiable et la diffusion atomique, cette dernière sera convoquée grâce à ***A-broadcast()*** et ***A-deliver()***.
L'algorithme de la diffusion atomique, présenté dans l'algorithme 1, consiste en trois tâches, Tâche1, Tâche2 et Tâche3, tel que :

1. Tâche1 : Lorsqu'un processus veut diffuser un message m, il exécute R-broadcast(m).
2. Tâche2 : Lorsqu'un processus p exécute R-deliver(m), il ajoute m à l'ensemble $R\_delivered_p$.

3. Tâche3 : Lorsqu'un processus p exécute A-deliver(m), il ajoute m à l'ensemble $A\_delivered_p$.

Algorithm 1 Un protocole de la diffusion atomique basé sur le détecteur de défaillances $\diamondsuit S$

Chaque processus P exécute le code suivant :
Initialisation
$R - delivered \leftarrow \Phi$;
$A - delivered \leftarrow \Phi$;
$K \leftarrow 0$;

Pour exécuter A-broadcast(m) : (tâche 1)
$R - broadcast(m)$

Pour exécuter R-deliver(m) :
$Lorsque R - deliver(m)$ **(tâche2)**
$R - delivered \leftarrow R - delivered \cup \{m\}$
$Lorsque R - delivered - A - delivered \neq \Phi$ **(tâche3)**
$K \leftarrow K + 1$
$A - undelivered \leftarrow R - undelivered - A - undelivered$
$Proposer(K, A - undelivered)$
Attendre jusqu'à décider (K, msgSetk)
$A - deliverk \leftarrow msgSetk - A - delivered$
Délivrer atomiquement, tous les messages de msgSetk non encore délivrés atomiquement suivant un certain ordre déterministe décidé par tous les processus.
$A - delivered \leftarrow A - delivered \cup A - deliverk$

Donc, l'ensemble $R\_delivered_p$- $A\_delivered_p$, noté $A\_undelivered_p$, est l'ensemble de messages reçu non encore délivré, atomiquement, par le processus p. Intuitivement, ce sont les messages soumis à la diffusion atomique mais non encore délivré, d'une façon atomique, suivant le processus p.

Dans la tâche3, un processus p vérifie, périodiquement,le contenu de l'ensemble $A\_undelivered_p$. S'il contient des messages, il procède à une autre exécution du consensus, soit la k^{ime}, en proposant l'ensemble $A\_undelivered_p$ en tant que l'ensemble de messages à délivrer atomiquement. Le processus p attend la décision du k^{ime} consensus, notée $msgSet^k$. A la fin, le processus p délivre, atomiquement, tous les messages de $msgSet^k$ non encore délivrés atomiquement suivant un certain ordre déterministe décidé par tous les processus.

3.2.2 Un protocole basé sur le détecteur de défaillances R

Ce travail (ajouter la réference de Ekwall, Schiper) introduit pour la première fois et d'une manière explicite les détecteurs de défaillances non fiables dans une solution de la diffusion atomique. La solution classique de la tolérance aux pannes se base sur le service de gestion de groupe [GCV01]. Les auteurs dans ce papier ont proposé une solution au problème de la diffusion atomique, qui représente une réduction au problème du consensus [CT96]. Cette solution s'exécute sur un anneau logique, et se base sur le mécanisme de jeton pour l'ordre de messages diffusés. Cependant, pour la détection de défaillances, ils ont introduit un nouveau détecteur de défaillances convenable pour l'organisation en anneau logique et s'implémente facilement avec le mécanisme Heartbeat (ajouter une référence).

3.2.2.1 Principe de l'algorithme

Les auteurs dans [REU04] ont considéré un système asynchrone composé d'un ensemble de n processus $\{p_0,...,p_n\}$ reliés entre eux par des canaux de communication fiables et organisés en anneau logique, tel que le k^{me} successeur d'un processus p_i est $p_{(i+k)modn}$, dénoté p_{i+k} pour la clarté. Similairement, le k^{me} prédécesseur d'un processus p_i est dénoté $p_{(i-k)}$. Les processus sont sujets à des défaillances par arrêt définitif.

Ce protocole se base toujours sur les détecteurs de défaillances pour la tolérance aux fautes. Il représente le premier travail, dans la littérature, qui utilise explicitement le mécanisme de détecteur de défaillances non fiable dans un algorithme de la diffusion atomique au lieu d'un service de gestion classique. Cependant, cette famille de détecteurs n'appartient pas à la famille classique des détecteurs de défaillances non fiable de Chandra et Toueg [CT96], mais elle représente une nouvelle classe de détecteurs non fiables, introduite dans ce papier, nommée R (pour *Ring*) définie par les deux propriétés suivantes :

- **Complétude** : Si p_{i-1} tombe en panne et p_i est correct, alors p_{i-1} sera, éventuellement, suspecté d'une manière permanente par p_i.
- **Précision** : Si p_{i-1} et p_i sont corrects, alors il existe un certain temps t, à partir duquel p_{i-1} ne sera plus jamais suspecté par p_i.

Les auteurs [REU04] ont montré dans ce même papier que le détecteur de défaillances R est plus fort que le détecteur de défaillances $\diamondsuit S$ [1] et plus faible que $\diamondsuit P$.

L'implementation de ce protocole nécessite un nombre de processus, dans le système, quadratique par rapport au nombre de défaillances (un coût très élevé par rapport aux

1. Chandra et al. ont montré que $\diamondsuit S$ est la classe de détecteurs de défaillances minimale pour résoudre le problème de consensus.

protocoles existants).

l'idée de ce protocole est assez simple. Elle se base sur une transformation directe de la solution du consensus en une solution de la diffusion atomique (par réduction au consensus). Cette solution utilise le mécanisme du jeton circulant pour transporter un ensemble de messages et récolter un nombre suffisant de votes autour de l'anneau logique conçu par les processus. Une fois le nombre de votes dépasse un certain seuil, l'ensemble de messages sera, atomiquement, délivré. Plus précisément, le jeton transporte les informations suivantes : {numéro de ronde (*ronde*), messages à ordonner (*a_ordonne*), nombre de votes récolté (*vote*), messages déjà ordonnés (*ordonne*)}.
Chaque processus p_i gère les structures de données suivantes, voir l'algorithme 2) :

- $ronde_i$: Représente le numéro de la ronde en cours d'exécution,
- $non\_ordonne_i$: Représente l'ensemble de tous les messages diffusés atomiquement, non encore ordonnés,
- $ordonne_i$: Représente l'ensemble de tous les messages déjà ordonnés par p_i, sous forme d'un couple (numéro de l'exécution du consensus, ensemble de messages),
- $consensus\_suivant_i$: Représente le prochain numéro de séquence à attribuer.

Algorithm 2 Un protocole de la diffusion atomique basé sur le détecteur de défaillance R

1 : **initialisation**
2 : $abroadcast_i \leftarrow \Phi$; $adeliv_i \leftarrow \Phi$; $round_i \leftarrow \Phi$;
3 : $nextCons_i \leftarrow 1$;
4 : si $(P_i = P_0)$ **alors**
5 : $envoyer(0, abroadcast0, 1, \Phi)a\{P_i, .., P_(i + f + 1)\}$;
6 : $round_0 \leftarrow n$;
7 : sinon
8 : **si**$(P_i \in \{P_i,..,P_{(i+f+1)}\}$ **alors**
$envoyer(-1, \Phi, 0, \Phi)a\{P_i, .., P_(i + f + 1)\}$;
9 : **Pour exécuter abroadcast(m)**
10 : envoyer m a tous
11 : A la livraison de m **faire**
12 : si m $\in$ { msgs | (msgs,-) $\in$ $ordered_i$ } **alors**
13 : $broadcast_i \leftarrow broadcast_i \cup \{m\}$
14 : **Pour exécuter delivrer (m)**
15 : tanque $\exists$ ($nextCons_i$,msgs) $\in$ seq **faire**
16 : $ordered_i \leftarrow ordered_i \cup \{(nextCons_i, msgs)\}$
17 : $unordered_i \leftarrow unordered_i \backslash msgs$
18 : adeliver les messages dans msgs dans un ordre déterministe
19 : $nextCons_i \leftarrow nextCons_i + 1$
20 : A la réception de ($round_i$, ordering, votes, ordered)
(round = $round_i$ - 1)**faire**
21 : si (|ordered | < |$adeliv_i$ |) **alors**
22 : $ordering \leftarrow \Phi$
23 : sinon // jeton avec de nouvelles informations
24 : delivrer (ordered)
25 : **si**$(round = roundi - 1)$**ou**$(ordering = \Phi)$**alors**
26 : $votes \leftarrow 0$;
27 : $votes \leftarrow votes + 1$;
28 : **si**$(vote = f + 1)$**alors**
29 : delivrer ({($nextCons_i$, oredering)})
30 : $ordering \leftarrow \Phi$
31 : si $(ordering = \Phi)$ **alors** // La nouvelle proposition
32 : $ordering \leftarrow abroadcast_i$
33 : $votes = 1$
34 : $jeton \leftarrow (round_i, ordering, votes, ordered)$
35 : $envoyer\ le\ jeton\ a\ \{P_i, .., P_(i + f + 1)\}$
36 : $(round_i \leftarrow round_i + 1)$
37 : $A\ la\ rception\ de\ (round, -, -, ordered)$
(round < $round_i$ - n) **faire**
38 : si (|ordered | > |$ordered_i$ |) **alors**
39 : $delivrer(ordered)$

L'algorithme se déroule en trois tâches principales :

1. De la ligne 1 jusqu'à 19 : Elle représente la partie *initialisation* de l'algorithme, ainsi la diffusion et la livraison atomique des messages. La procédure de livraison (de la ligne 14 à 19) est convoquée par la deuxième partie.
2. De la ligne 20 jusqu'à 39 : Représente la partie *traitement* du jeton. Un processus donné doit vérifier l'émetteur du jeton, s'il a reçu ce dernier de son prédécesseur immédiat, ou bien de l'un de ses $(f+1)$ prédécesseurs. Dans les deux cas, il doit vérifier les deux ensembles *ordonne* et *a_ordonne* du jeton. S'il l'ensemble *ordonne* est inférieur à son ensemble, alors il initialise, automatiquement, *a_ordonne* à vide. Dans le cas contraire, il appelle la procédure *delivrer*(*ordonne*). A ce niveau, on peut envisager deux cas possible : soit l'émetteur du jeton n'est pas le prédécesseur immédiat ou l'ensemble *a_ordonne* est vide, dans les deux cas, le processus initialise la variable *vote* à zéro. Dans le cas échéant, *vote* est incrémentée. Dès qu'on récolte un nombre suffisant de votes, on délivre l'ensemble *a_ordonne* avec le numéro de séquence courant. Dans le cas où on a reçu le même ensemble *ordonne*, et l'ensemble *a_ordonne* est vide, alors on met à jours la liste *a_ordonne* par la liste $ordonne_i$, on initialise *vote* à 1, on met à jours les informations contenues dans le jeton par ($ronde_i$, $a\_ordonne_i$, *vote*, *ordonne*) et on envoie le jeton aux prochains $(f+1)$ successeurs. Finalement, on incrémente la ronde $ronde_i$ de n.
3. De la ligne 37 jusqu'à 39 : Cette partie traite la réception d'autres jetons (un jeton antérieur qui contient un ensemble *ordonne* plus à jours par rapport à l'ensemble de ce processus).

3.2.3 Un protocole basé sur le service de gestion de groupe

Les auteurs [PMP12] propose une solution au problème de la diffusion atomique basée sur le paxos [2] [Lam98] (il sera détaillé dans la section suivante). Elle est appelée Ring Paxos (un Paxos qui s'exécute sur une topologie en anneau logique). Dans ce qui suit, nous présenterons le principe du Paxos pour pouvoir présenter et expliquer le Ring Paxos.

3.2.3.1 Le protocole Paxos

Dans cette section, on s'intéresse au problème du consensus. Au cours d'une instance de consensus (identifiée par k), des valeurs initiales (potentiellement distinctes) peuvent être proposées par un ou plusieurs processus appelés des *auteurs* de propositions. Durant

2. Le Paxos est une solution proposée par Leslie Lamport pour résoudre le problème de consensus dans un système asynchrone.

l'instance k, le ou les auteurs communiquent leur proposition au même sous-ensemble de n processus du système. Ces n processus qu'on appellerai par la suite les *acteurs* sont directement impliqués dans l'exécution du protocole de consensus. Ils doivent collaborer pour assurer que celui-ci converge inéluctablement (terminaison) vers une valeur de décision unique (accord) qui doit nécessairement être l'une des valeurs proposées (validité). La valeur de décision est alors retournée vers tous les auteurs qui ont fourni (ou fourniront) une valeur initiale concernant cette instance de consensus particulière.
Ce protocole repose sur la notion de leader ultime : Il existe un instant à partir duquel un acteur correct est considéré par tous les processus comme étant le seul leader (et ceci durant un laps de temps suffisant pour qu'une décision puisse être prise). Les protocoles de la famille Paxos [3] s'appuient sur la notion de quorum majoritaire [GV10] : Si f est le nombre maximal de pannes pouvant affecter un système de n acteurs alors $f < n/2$. Depuis la présentation de la version initiale, deux optimisations majeures ont été proposées dans le cas du protocole Paxos.

Deux rôles distincts (Coordinateur et Accepteur) sont définis et chaque acteur joue un ou deux rôles (voir l'algoritme 3). Plus précisément, les n acteurs (et de fait, une majorité de processus corrects) jouent le rôle d'accepteurs tandis qu'au moins *(f+1)* acteurs (et de fait, au moins un processus correct) jouent le rôle de coordinateur. Un coordinateur n'est actif que lorsque le service d'élection de leader le désigne comme leader. En pensant alors agir comme un leader unique et incontesté (ce qui n'est pas forcément le cas), le coordinateur tente d'imposer une valeur de décision aux autres acteurs. Dans les protocoles de la famille Paxos, un numéro de ronde (un tour) r est associé à chaque tentative. Ce numéro est propre au processus leader N_i qui exécute la tentative. Dans la version originale du protocole Paxos, un leader tente d'imposer une valeur de décision en exécutant successivement deux phases au cours d'une ronde r. Au cours d'une première phase de Préparation, le leader s'assure que la valeur qu'il soumettra lors de la seconde phase n'est pas incompatible avec celles éventuellement soumises par d'autres coordinateurs ayant agi comme leader au cours du même consensus mais durant des rondes précédentes (i.e., dont les numéros sont inférieurs à r). La phase de préparation implique la diffusion d'un message du leader vers l'ensemble des accepteurs puis la collecte, par le leader, de réponses favorables en provenance d'une majorité d'accepteurs (soit deux étapes de communication). La seconde phase est une phase de Proposition. Elle débute une fois que le leader a identifié une valeur qu'il peut soumettre sans risquer de violer les propriétés de sûreté (Accord et Validité).La valeur est diffusée par le leader vers l'ensemble des accepteurs puis le leader attend de collecter une majorité de réponses favorables avant de pouvoir considérer que cette valeur soumise

3. Il y a plusieurs variantes du Paxos, proposées par Leslie Lamport et d'autres chercheurs, dans la littérature.

est la valeur de décision (soit à nouveau deux étapes de communication). Un accepteur est une entité passive dont l'état fait référence à la dernière phase de préparation acceptée (numéro de ronde r_1) et à la dernière phase de proposition acceptée (numéro de ronde r_2 et valeur adoptée). Cet état ne peut évoluer que lors de la réception d'une requête diffusée par un coordinateur et à condition que cette requête soit acceptable. Une requête peut être ignorée par un accepteur si la mise à jour qu'elle entrainerait ne garantit pas **i)** que la valeur de r_1 croit, **ii)** que la valeur de r_2 croit ou **iii)** que $r_1 \leq r_2$ au moment d'une mise à jour de r_2.

Dans sa version originale, le protocole requiert donc 4 étapes de communications entre les acteurs auxquelles viennent s'ajouter 2 étapes "externes" correspondant à la diffusion des valeurs initiales des auteurs vers les coordinateurs et à la diffusion de la valeur de décision du leader vers les auteurs. Le chemin de communication correspond à : auteur $\rightarrow$ coordinateurs (leader) $\rightarrow$ accepteurs $\rightarrow$ leader $\rightarrow$ accepteurs $\rightarrow$ leader $\rightarrow$ auteurs.

Algorithm 3 Un protocole de Paxos basé sur le service de gestion de groupe

1 : Algorithme de Paxos

2 : Tâche 1 : (coordinateur)
3 : A la réception de la valeur v de l'auteur de proposition
4 : Augmentez c-rnd à une valeur unique arbitraire
5 : **Pour** tout P $\in N_a$ **faire** envoyer(P, (PHASE 1A, c-rnd))

6 : Tâche 2 : (accepteur)
7 : A la réception (PHASE 1A, c-rnd) du coordinateur
8 : **si** c-rnd $>$ rnd **alors**
9 : $rnd \leftarrow c - rnd$
10 : envoyer (coordinateur, (PHASE 1B, rnd, v-rnd, v-val))

11 : Tâche 3 : (coordinateur)
12 : A la réception (PHASE 1B, rnd, v-rnd, v-val) de Q_a tel que c-rnd = rnd
13 : K= Max(v-rnd) valeur reçue
14 : V= l'ensemble (v-rnd, v-val) reçu avec v-rnd =k
15 : **si** $K = 0$ **alors** $v - val \leftarrow v$
16 : **sinon** $c - val \leftarrow v - val$
17 : **Pour** tout P $\in N_a$ **faire** envoyer(P, (PHASE 2A, c-rnd, c-val))

18 : Tâche 4 : (accepteur)
19 : A la réception (PHASE 2A, c-rnd, c-val) du coordinateur
20 : **si** c-rnd $\geq$ rnd **alors**
21 : $v - rnd \leftarrow c - rnd$
22 : $v - val \leftarrow c - val$
23 : envoyer (coordinateur, (PHASE 2B, v-rnd, v-val))

24 : Tâche 5 : (coordinateur)
25 : A la réception (PHASE 2B, v-rnd, v-val) de Q_a
26 : **si** pour tous les messages reçus : v-rnd = c-rnd **alors**
27 : **Pour** tout P $\in N_l$ **faire** envoyer(P, (DECISION, v-val))

3.2.3.2 Le protocole Ring Paxos

Ring Paxos [PMP12] est une variation du Paxos, optimisé pour les réseaux clustérisés et résout le problème de la diffusion atomique. Il suppose un coordinateur fixe , aucune défaillance des processus n'est tolérée et aucune perte de messages.

3.2.3.3 Modèle de système

Les auteurs supposent un modèle distribué crash-recovery (chercher la traduction) dans lequel les processus communiquent entre eux par l'échange de messages. Les processus

peuvent subir des défaillances par arrêt définitif. Ainsi, les processus ont l'accès à une mémoire stable nécessaire pour le recouvrement après une défaillance.
La communication peut être point à point, via les primitives *envoyer(p,m)* et *recevoir(m)*, et un à plusieurs, via les primitives *ip-multicast(g,m)*et *ip-delivrer(m)*, où :

- m : Un message,
- p : Un processus,
- g : Un groupe de processus.

Les messages peuvent être perdus mais ne peuvent jamais être interrompus.
Pour assurer la correction du protocole et surmonter l'impossibilité de FLP [FLP85], les auteurs suppose un système partiellement synchrone [DLS88], qui est, initialement, asynchrone et devient éventuellement synchrone après un Temps de Stabilisation Global [4], non connu par tous les processus.

3.2.3.4 Principe de l'algorithme

la figure 3.1 présente le Ring Paxos ; les parties encadrées sont communes entre le Paxos et le Ring Paxos. Similairement au Paxos, l'exécution se déroule en deux phases. Ainsi, Ring Paxos implémente le même mécanisme utilisé par le Paxos pour assurer qu'une seule valeur puisse être décidée dans une instance de consensus.

Ring paxos dispose d'un quorum majoritaire d'accepteurs dans un anneau logique dirigé [5] (voir l'algorithme 4 [PMP12]). Le coordinateur joue,aussi, le rôle d'un accepteur dans Ring Paxos et il est placé en dernier dans l'anneau logique des accepteurs.

Quand le coordinateur commence la première phase (Phase1, Task1), il propose l'anneau pour être utilisée dans la deuxième phase (Phase2). Le numéro de l'anneau proposée est gardé par le coordinateur dans une variable *c-ring*.

Les accepteurs envoient une réponse au coordinateur et acquittent implicitement l'acceptation de l'anneau proposée dans (Phase1, Task2).

A cet effet, le coordinateur affecte un identificateur unique à la valeur proposée utilisée dans (Phase2, Task3). Ring Paxos exécute un consensus sur les identificateurs des valeurs. Les valeurs proposées sont transmises aux accepteurs et aux auteurs dans les messages *Phase2A* en utilisant *ip-multicast*.

4. Global Stabilization Time (GST)
5. Logical Directed Ring.

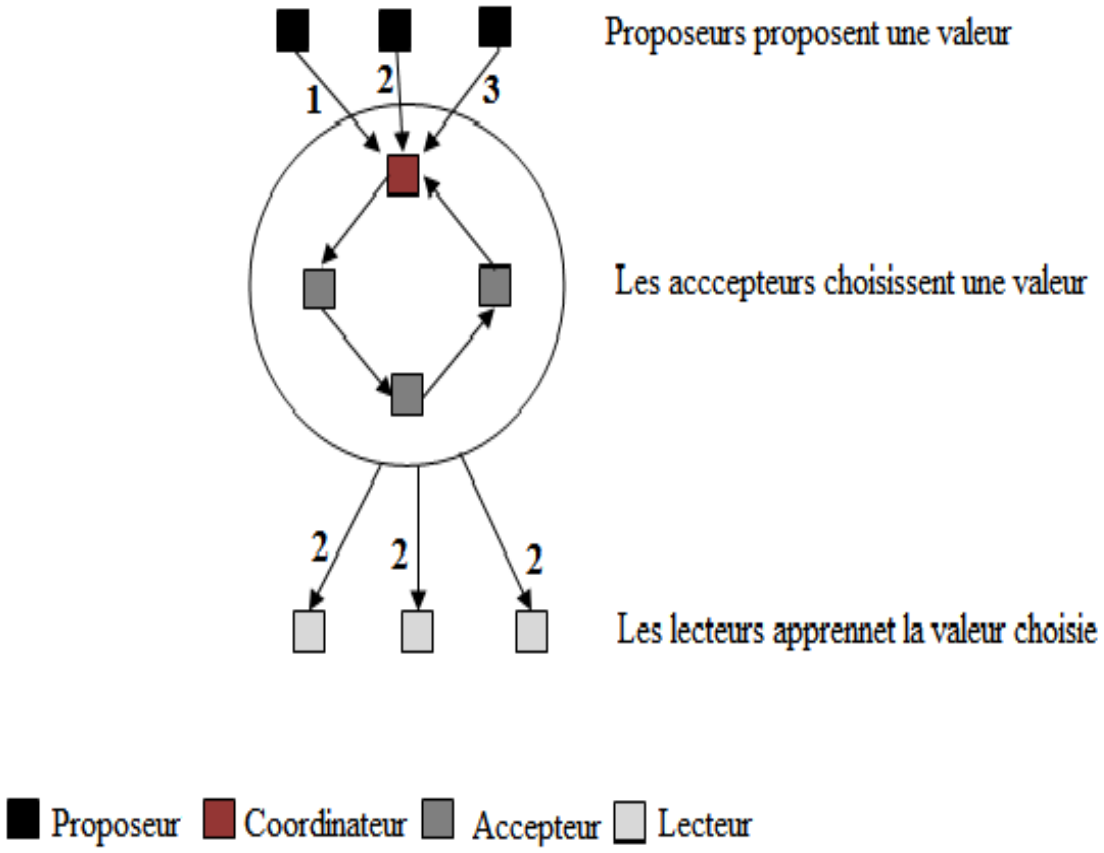

FIGURE 3.1 – le Protoco Ring Paxos

Quand un accepteur délivre un message *Phase2A* dans (*Task4*), il vérifie s'il peut voter sur la valeur proposée. Si oui, il met à jour ses variables *v-rnd*, *v-val* et *v-vid* (elle contient l'identificateur unique de la valeur proposée). Le premier accepteur dans l'anneau envoie un message *Phase2B* à son successeur.

Le prochain accepteur, dans l'anneau, en recevant le message *Phase2B*, vérifie s'il a déjà délivré la valeur proposée par le coordinateur dans le message *Phase2A*. Cela peut être effectué en comparant les deux variables *v-vid* de l'accepteur et l'identificateur de la valeur calculé par le coordinateur. Deux cas peuvent être envisagés, si la comparaison est positive :

- L'accepteur n'est pas le dernier dans l'anneau : Il envoie un message *Phase2B* à son successeur dans l'anneau ;
- L'accepteur est le coordinateur : Il diffuse un message de décision contenant l'identificateur de la valeur choisie.

De ce fait, les auteurs délivrent le message de décision reçu du coordinateur contenant la valeur déjà transmise, toujours par le coordinateur, dans le message *Phase2A*.

Algorithm 4 Un protocole de Ring Paxos basé sur le service de gestion de groupe

1 : Algorithme de Ring Paxos

2 : Tâche 1 : (coordinateur)
3 : A la réception de la valeur v de l'auteur de proposition
4 : Augmentez c-rnd à une valeur unique arbitraire
5 : $c - ring \leftarrow ring$
6 : **Pour** tout P $\in Q_a$ **faire**
envoyer(P, (PHASE 1A, c-rnd, c-ring))

7 : Tâche 2 : (accepteur)
8 : A la réception (PHASE 1A, c-rnd, c-ring) du coordinateur
9 : **si** c-rnd $>$ rnd **alors**
10 : $rnd \leftarrow c - rnd$
11 : $ring \leftarrow c - ring$
12 : envoyer (coordinateur, (PHASE 1B, rnd, v-rnd, v-val))

13 : Tâche 3 : (coordinateur)
14 : A la réception (PHASE 1B, rnd, v-rnd, v-val) de Q_a tel que c-rnd = rnd
15 : K= Max(v-rnd) valeur reçue
16 : V= l'ensemble (v-rnd, v-val) reçu avec v-rnd =k
17 : **si** $K = 0$ **alors** $v - val \leftarrow v$
18 : **sinon**
$c - val \leftarrow v - val$
19 : c-vid = ID(c-val)
20 : **ip-multicast** $(Q_a \cup N_l, (PHASE\ 2A, c - rnd, c - vid, c - val))$

21 : Tâche 4 : (accepteur)
22 : A la réception (PHASE 2A, c-rnd, c-vid, c-val) du coordinateur
23 : **si** c-rnd $\geq$ rnd **alors**
24 : $v - rnd \leftarrow c - rnd$
25 : $v - val \leftarrow c - val$
26 : $v - vid \leftarrow c - vid$
27 : **si** le premier dans le ring **alors**
28 : envoyer (successeur, (PHASE 2B, c-rnd, c-vid))

29 : Tâche 5 : (coordinateur et accepteur)
30 : A la réception (PHASE 2B, c-rnd, c-vid) de prédécesseur
31 : **si** (v-vid = c-vid) **alors**
32 : **si** n'est pas le dernier dans le ring**alors**
33 : envoyer (successeur, (PHASE 2B, c-rnd, c-vid))
34 : **sinon**
35 : **ip-multicast** $(Q_a \cup N_l, (\mathbf{DECISION}, c - vid))$

3.3 Tableau comparatif des protocoles de la diffusion atomique

Le tableau 3.1 présente une comparaison et une synthèse des protocoles de la diffusion atomique étudiés au cours de ce chapitre. La comparaison est faite suivant les critères suivants :

- **Nombre de messages échangés** : C'est le nombre de messages nécessaire pour délivrer une séquence de messages.
- **Type de défaillance de processus toléré** : C'est le mode de défaillance qu'un processus puisse subir.
- **Type de communication** : C'est le type de communication (point à point, multipoints, etc.) utilisé entre les processus.
- **Nombre de processus nécessaire** : C'est le nombre de processus n nécessaire dans le système pour la correction du protocole par rapport au nombre de défaillants f toléré.
- **Mécanisme de tolérance aux fautes** : C'est l'approche utilisée pour détecter les défaillances des processus.
- **Topologie** : C'est la manière d'organisation des processus.
- **Mécanisme d'ordre de messages** : C'est le mécanisme utilisé pour ordonner les messages.

	Le protocole de Chandra et Toueg	**Le protocole de Ekwall et al.**	**Le protocole Ring Paxos**
Nombre de messages échangés	Diffusion+ exécution du consensus K fois	Vote de (f+1) successives	proposition + exécution du consensus + diffusion de la décision
Type de défaillance de processus toléré	Arrêt définitif	Arrêt définitif	Arrêt définitif
Type de communication	Mutlipoints (n)	Mutlipoints (($f+1$))	point à point
Nombre de processus défaillants toléré	$n \geqslant (2f+1)$	$n \geqslant f(f+1)+1$	$Qa \geqslant f/2$
Mécanisme de tolérance aux fautes	Détecteur de défaillance ◊S	Détecteur de défaillance R	Service de gestion de groupe
Topologie	Complètement connecté	Anneau logique	Complètement connecté + anneau majoritaire
Mécanisme d'ordre de messages	Consensus	vote majoritaire de ($f+1$ processus	Consensus

TABLE 3.1 – Synthèse des protocoles de la diffusion atomique

D'aprés le tableau comparatif, nous avous pu extraire les inconvénients de chaque protocole :

Le protocole Chandra et Toueg : la communication total(all- to- all) qui surcharge le réseau par les messages de traitement et de contrôle. Le protocole de Ekwall et al : le nombre de processus global par rapport au nombre de processus défaillants, ce protocole utilise un détecteur de défaillance R qui est très puissants par rapport aux autres détecteurs de défaillances. Le protocole Ring Paxos utilise un service de gestion de groupe qui génère plus de temps, ainsi que l'utilisation d'un coordinateur fixe.

3.4 Conclusion

Dans ce chapitre, nous avons présenté trois principaux protocoles de la diffusion atomique. Chaque protocole est caractérisé par ses propres mécanismes : d'ordre de messages diffusés, de tolérance aux fautes, et la manière d'organisation des processus. L'étude de ces

protocoles nous a amené à réaliser une comparaison suivant des critères bien définis. Suite à cette étude et cette comparaison, nous avons pu extraire les avantages ainsi les défis et les insuffisances de ces solutions et nous avons pu donner naissance à une nouvelle solution, dans le chapitre suivant, et évaluer les performances de cette solution.

Chapitre 4

Proposition et évaluation des performances d'un nouveau protocole de la diffusion atomique

4.1 Introduction

Plusieurs protocoles de la diffusion atomique ont été proposés dans la littérature. Ces protocoles peuvent être classés selon leurs mécanismes utilisés pour ordonner les messages ou bien selon leurs manières de détecter les défaillances des processus.

Dans ce chapitre, nous présenterons un nouveau protocole qui résout le problème de la diffusion atomique. Ce protocole se base sur un détecteur de défaillances pour la tolérance aux pannes, et une combinaison entre la circulation du jeton et le principe du coordinateur pour l'ordre de messages diffusés.
Nous commencerons ce chapitre par la présentation du modèle de système. Nous donnerons par la suite le principe du protocole, sa preuve de correction ainsi le protocole. Nous définirons, ensuite, l'environnement de simulation, suivis de la présentation de quelques résultats de simulation et leurs analyses. Nous conclurons le chapitre par une conclusion.

4.2 Modèle de système

Nous considérons un système distribué asynchrone, composé d'un ensemble $II = (p_0,...,p_n)$ de $\boldsymbol{n}$ processus. Ces derniers sont reliés entre eux par des canaux de communication fiables de telle sorte qu'ils forment un réseau complètement connecté, ce qui permet à ces processus de se synchroniser et de communiquer entre eux par l'échange de messages.

Nous supposons un mécanisme de tolérance aux fautes qui utilise les détecteurs de défaillances non fiables de la classe ◊S, tel que $f < n/2$ sachant que f représente le nombre maximum de défaillances tolérées. Un processus ne peut subir une défaillance que par crash (ou arrêt définitif).

4.3 Structures de données

Chaque processus correct, participant au protocole de la diffusion atomique, manipule les structures de données suivantes :
N_a : Ensemble contenant tout les processus.
Q_a : Un quorum d'accepteurs.
N_l : C'est un sous ensemble de N_a, qui désigne l'ensemble des processus lecteurs.
rnd : C'est le numéro de ronde (tour), il permet de distinguer les valeurs proposées par les différents coordinateurs.
c-rnd : Représente le numéro de ronde du coordinateur, initialisé à 0.
v-rnd : La valeur décidée dans une ronde (une séquence de messages).
ring : C'est la liste des identificateurs des processus disposant sur l'anneau virtuel proposé par le coordinateur courant.
c-ring : Représente le numéro de l'anneau correspondant au coordinateur courant.
c-val : Contient le numéro de ronde où un accepteur a effectué une vote.
v-val : Contient la séquence de messages décidée par un accepteur.
c-vid : L'identifiant unique affecté par le coordinateur à la valeur proposée.
v-vid : L'identifiant unique affecté par un accepteur à la valeur proposée par le coordinateur.
Decision : Ce message est diffusé par le coordinateur pour informer les autres processus corrects de la séquence de messages à délivrer selon l'ordre imposé.
Abort : Lorsque le coordinateur est suspecté par la majorité des accepteurs, il diffuse ce message pour les informer que le consensus est interrompu, s'il est correct.
nack : Quand un processus accepteur ne reçoit pas le message attendu du coordinateur, il lui envoie ce message *(nack)*.

4.4 Principe du protocole proposé

Dans ce protocole un processus pourra jouer (exécuter) plusieurs rôles : *Coordinateur*, *Accepteur*, *Lecteur* et *Proposeur*. Le coordinateur courant cherche à joindre une majorité d'accepteurs pour atteindre un consensus unanime sur la séquence de messages à délivrer et l'ordre de livraison de ces messages.

Le protocole s'éxecute en cinq tâches principales. Le nombre de phases nécessaire pour valider une séquence de messages ainsi que son ordre de livraison est de deux phases : *A* et *B*. Le protocole se déroule comme suit :

- **Tâche 1** : Cette tâche est exécutée seulement par le *coordinateur*. Après qu'il ait reçu des messages de la part des *proposeurs*, le *coordinateur* courant cherche à joindre un quorum *d'accepteurs* (ensemble Q_a) en disposant les processus de cet ensemble en un anneau pour les préparer à participer à un consensus sur l'ensemble et l'ordre de livraison des messages proposés de la part des *proposeurs*, cela en lançant la phase *1A* (ligne 4).

- **Tâche 2** : Elle est exécutée par les processus *accepteurs*, chaque processus p de Q_a se met en attente de la réception d'un message *(Phase 1A)* de la part du *coordinateur*. S'il le reçoit, donc il va chercher à rejoindre le numéro de ronde le plus récent (ligne5) et ignore les plus anciens. Puis, il envoie un message *(Phase 1b, rnd, v-rnd, v-val)* au *coordinateur* pour l'informer qu'il accepte l'anneau proposé et lui fournir la dernière valeur votée *v-val* ainsi que le numéro de ronde correspondant à ce vote *v-rnd* (ligne 10). Dans le cas contraire (l'accepteur n'a pas reçu le message du coordinateur), il le considère comme défaillant (grâce aux informations fournies par le détecteur de défaillances), et dans ce cas il envoie un message *nack* au *coordinateur*.

- **Tâche 3** : Lors de l'exécution de cette tâche, et suite à la collection des résultats des rondes précédentes de la part de la majorité des processus Q_a, le coordinateur attend la réception d'un message de la Phase *1B* contenant la dernière valeur *v-val* sur laquelle a votée au moins un processus dans Q_a et qui n'est pas encors décidée. S'il reçoit ce message de la part de l'ensemble Q_a mais ne contient aucune valeur, alors il propose une nouvelle valeur et essaie d'obtenir une majorité de voix pour l'approuver, sinon il choisit une valeur envoyée par l'un des accepteurs. Pour pouvoir distinguer laquelle des valeurs pourra être proposée dans la phase *2B*, le processus *coordinateur* affecte une identité unique *c-vid* à la valeur dont il va proposer aux processus du Q_a (ligne 13 jusqu'à 20). En cas de réception du message Phase *1B* contenant un *nack*, le coordinateur diffuse un message Abort de Phase *2A* en appelant la primitive *R-Broadcast* (ligne 21). Pour les informer que le consensus a été interrompu.

- **Tâche 4** : Au niveau de cette tâche, nous envisageons deux cas. Dans le premier, si le processus *accepteur* reçoit le message *(Phase 2A, c-rnd, c-vid, c-val)*, il vérifie s'il peut voter pour cette valeur (ligne 24), si c'est le cas, ses variables *v-rnd*, *v-val*, et

v-vid seront mises à jour. Après un message *(Phase 2B, c-rnd, c-vid)* sera envoyé à son successeur dans l'anneau jusqu'à son arrivée au coordinateur (le dernier processus dans l'anneau). dans le deuxième cas, l'accepteur reçoit le message *(Phase 2A, Abort)*, donc le *coordinateur* a été suspecté d'être défaillant par la majorité, alors il émet au *coordinateur* un message *nack* (ligne 29).

- **Tâche 5** : C'est à ce niveau d'exécution du protocole que sera diffusé la séquence de messages à délivrer atomiquement. Après que le *coordinateur* ait reçu l'accord de la part du quorum Q_a (Ligne 33) sur *v-val*, il diffusera un message de décision *(décision, c-vid)* en faisant appel à la primitive *R-broadcast*. Dans le cas contraire (ligne 35), lorsque le message *(Phase 2B, nack)* sera reçu par le *coordinator*, dans ce cas il diffusera un message d'interruption *(Phase 2B, Abort)*.

L'algorithme 5 montre les instructions détaillées de l'algorithme.

Algorithm 5 Un nouveau algorithme Ring Paxos Failure detector based

Tâche 1 : (Coordinateur)
1 : A la réception de la valeur v de l'auteur de proposition
2 : Augmentez c-rnd à une valeur unique arbitraire
3 : $c - ring \leftarrow ring$
4 : **Pour** tout P $\in Q_a$ **faire** envoyer(P, (PHASE 1A, c-rnd, c-ring))
Tâche 2 : (Accepteur)
5 : Attendre jusqu'à la réception [(PHASE 1A, c-rnd, c-ring)] Ou coordinateur $\in D_a$ le coordinateur suspecté d'être défaillant
6 : **Si** la reception de [(PHASE 1A, c-rnd, c-ring)] **alors**
7 : **si** c-rnd $>$ rnd **alors**
8 : $rnd \leftarrow c - rnd$;
9 : $ring \leftarrow c - ring$;
10 : envoyer (coordinateur, (PHASE 1B, rnd, v-rnd, v-val))
11 : **Sinon** envoyer (coordinateur, (PHASE 1B, rnd, nack)) ;
Tâche 3 : (Coordinateur)
12 : Attendre jusqu'à la réception [(PHASE 1B, rnd, v-rnd, v-val)] Ou la réception [(PHASE 1B, rnd, nack)] de Q_a
13 : **Si** la reception de [(PHASE 1B, rnd, v-rnd, v-val)] **alors**
14 : K= Max(v-rnd) valeur reçue
15 : V= l'ensemble (v-rnd, v-val) reçu avec v-rnd =k
16 : **si** (K=0) **alors** $c - val \leftarrow v$;
17 : **sinon** c-val la seul valeur dans V ;
18 : c-val la seul valeur dans V ;
19 : $c - vid = ID(c - val)$;
20 : **ip-multicast** $(Q_a \cup N_l, (PHASE\ 2A, c - rnd, c - vid, c - val))$;
21 : **Sinon** envoyer(Q_a, (PHASE 2A, Abort)) ;
Tâche 4 : (Coordinateur et Accepteur)
22 : Attendre jusqu'à la réception [(PHASE 2A, c-rnd, c-val, c-vid)] de coodinateur Ou coordinateur $\in D_a$ le coordinateur suspecté d'être défaillant
23 : Si la reception de [(PHASE 2A, c-rnd, c-val, c-vid)] **alors**
24 : **si** c-rnd = rnd **alors**
25 : $v - rnd \leftarrow c - rnd$;
26 : $v - val \leftarrow c - val$;
27 : $v - vid \leftarrow c - vid$;
28 : **si** le premier dans ring **alors** envoyer (successeur, (PHASE 2B, c-rnd, c-vid))
29 : sinon envoyer (coordinateur, (PHASE 2B, c-rnd, nack))
Tâche 5 : (Coordinateur et Accepteur)
30 : Attendre jusqu'à la réception [(PHASE 2B, c-rnd, c-vil)] de predeccesseur Ou la réception [(PHASE 2B, nack)] de Q_a
31 : **Si** la reception de [(PHASE 2B, c-rnd, c-vil)] **alors**
32 : **si** v-vid = c-vid **alors**
33 : **si** n'est pas le dernier dans le ring **alors** envoyer (successeur, (PHASE 2B, c-rnd, c-vid))
34 : **sinon** $R - broadcast\ (Q_a \cup N_l, (\mathbf{DECISION}, c - vid))$;
35 : **Sinon** $R - broadcast\ (PHASE2B, nack)$;

4.5 Les preuves de correction du protocole proposé

Dans un système distribué, tout problème doit satisfaire les deux propriétés suivantes : La vivacité et la sûreté [Lam77]. De manière informelle, la sûreté assure que rien de mauvais n'arrivera au système durant l'exécution du protocole, la vivacité garantit que quelque chose finira par se produire durant cette exécution [REB07].

Lemme 1 : Si un processus correct délivre un message m, alors inéluctablement tous les processus corrects délivrent m (propriété d'accord).

Preuve : Ce lemme est garanti par l'utilisation de la primitive de diffusion fiable (R-broadcast) qui assure, que tous les processus délivrent le même ensemble de messages diffusé par le *coordinateur*, qui de sa part, garantira la diffusion de la séquence de messages à délivrer à tous les processus du système.

Lemme 2 : Si un processus délivre un message m alors m a été diffusé par au moins un processus correct (propriété de validité).

Preuve : Chaque processus correct ne délivre un message m sauf s'il a été accepté par une majorité des *accepteurs*. Les *accepteurs* votent sur la séquence de messages proposée par le *coordinateur*.

Lemme 3 : Pour tout message m , chaque processus correct délivre le message m au plus une fois, et seulement si m a été précédemment diffusé par un certain processus correct (propriété d' intégrité).

Preuve : A chaque message m est affecté un identificateur unique qui permettra aux processus de vérifier si ce message a été déjà délivré ou non.

Lemme 4 : Si deux processus corrects p et q délivrent deux messages m et m_0, alors p délivre m avant m_0 si, et seulement si, q délivre m avant m_0 (propriété d'ordre total).

Preuve : Cette propriété est garantie par la diffusion de la séquence de messages (ou la décision) par le *coordinateur*. l'utilisation des détecteurs de défaillance $\Diamond\ \boldsymbol{S}$ assure que ultimement un processus correct ne sera plus suspecté par aucun processus correct, ce processus sera le coordinateur, qui imposera sa liste de messages pour qu'elle sera délivrée par tous les processus corrects dans le même ordre.

Théorème : l'algorithme 5 implémente les primitives de la diffusion atomique.

4.6 Evaluation de performances du protocole proposé

Le développement actuel des systèmes distribué, permet d'élaborer de nouvelles applications plus performantes en distribuant le travail sur plusieurs machines inter-connectées. Des problèmes physiques et logiques, qui n'apparaissent pas dans une application centralisée, doivent être résolus. Des algorithmes, comme celui de la détection de défaillances, permettent aux machines de mieux gérer ces problèmes. Lors de l'élaboration d'application distribuée, on voudrait alors prévoir le comportement du système, lors d'une faute de machine par exemple. La simulation est une solution simple et peu coûteuse à mettre en œuvre pour résoudre ce problème. Elle permet, par exemple, de prédire le temps d'exécution de l'algorithme distribué, etc.

Dans cette section, nous évaluons les performances de notre protocole. Pour ce fait, nous utilisons un simulateur connu dans les systèmes distribués, neko.

4.6.1 Le simulateur NEKO

Neko est une plateforme de communication des algorithmes distribués qui intègre différentes couches, chacune permet de réaliser plusieurs concepts fondamentaux dans le contexte des systèmes distribués tels que la diffusion fiable, la détection de panne, le consensus, la diffusion atomique,etc.[Ber99].

Neko est édité en langage Java qui est réputé pour sa mobilité, de cette optique, neko est extrêmement mobile (on peut l'exécuter sous différents environnements tel que Linux, Windows ou MAC OS [Ber99].

L'architecture générale du simulateur neko est schématisée comme suit(voir la figure 4.1) :

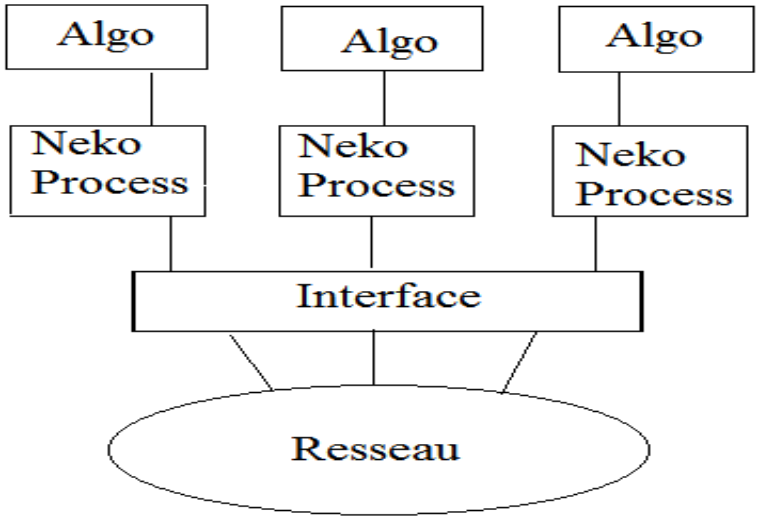

FIGURE 4.1 – Architecture générale de Neko

- Au niveau le plus bas, nous avons le réseau qui est considéré comme une "boîte noire" qui reçoit un message et le redirige vers le bon destinataire, dans un ordre quelconque (déterminé selon le type du réseau : Ethernet, FDDI, etc.).
- Une interface entre les processus et le réseau, elle permet d'envoyer et de recevoir des messages depuis le réseau.
- Les processus, avec leurs espaces d'adressage, qui envoient des messages au réseau et en reçoivent.

4.6.2 Les étapes d'exécution du simulateur neko

Dans notre cas, nous avons choisi d'utiliser le simulateur Neko sous ubuntu (environnement Linux). Neko s'exécute sous linux comme suit :

- Entrer dans le dossier neko depuis le terminal ;
- Exécuter la commande **./configure** ;
- Procéder la compilation du simulateur par l'outil **ant** ;
- Accéder au chemin où se trouve le fichier de configuration, puis taper la commande suivante **java lse.neko.Main <fichier de configuration>**.

4.6.3 Paramètres de simulation

Dans cette simulation, nous avons utilisé un seul paramètre $\lambda(0 < \lambda)$ qui indique la vitesse relative du traitement d'un message sur une processus par rapport à sa transmission

sur le réseau. Nous avons opté pour λ=1, qui indique que le traitement du processus et la transmission sur le réseau ont le même coût.

4.6.4 Résultats de la simulation

Dans cette section, nous allons présenter ce que nous avons obtenu comme résultats de simulation en comparant le protocole proposé avec les deux protocoles déjà présentés, de *Chandra et Toueg* [CT96] et celui du *Ring Paxos* [PMP12], en terme de la variation de la latence et le débit selon l'alternance du nombre de processus. Comme notre protocole utilise les détecteurs de défaillances, nous allons évaluer notre protocole suivant les deux cas d'exécution suivants :

- Exécution sans crash des processus.
- Exécution avec crash des processus.

La latence est définie en le temps qui s'ecoule entre l'émission du message et sa livraison atomique, elle est calculée comme suit :

$$L = 1/n(\sum_{i=0}^{n-1} t_i) - t_a \tag{4.1}$$

avec :

- L : Représente la valeur de la latence.
- n : C'est le nombre de processus dans le système.
- i : Représente le numéro du processus dans le système tel que i $\in$ {0, 1, 2,.., n-1}.
- t_a : Le temps d'émission *R-Broadcast(m)*.
- t_i : Le temps de consommation *adeliver(m)*. (validation d'un message par un processus p_i).

Le débit c'est le nombre de messages délivrés atomiquement.

4.6.4.1 Exécution sans crash des processus

Dans ce cas d'exécution, l'évaluation de performances est faite dans un environnement (ou système) sans occurrence de défaillance de processus. Les résultats de la figure illustrent les variations de la latence en terme du nombre de processus (Latence/nombre de processus). Tandis que les résultats de la figure illustrent les variations du débit suivant le nombre global de processus.

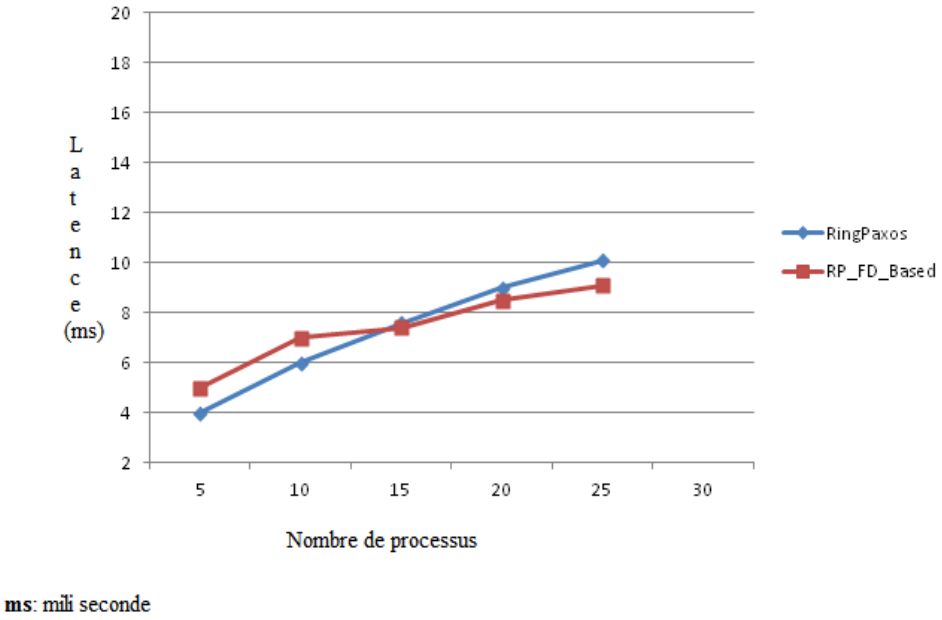

FIGURE 4.2 – Latence vs nombre de processus : Execution sans crash (λ=1)

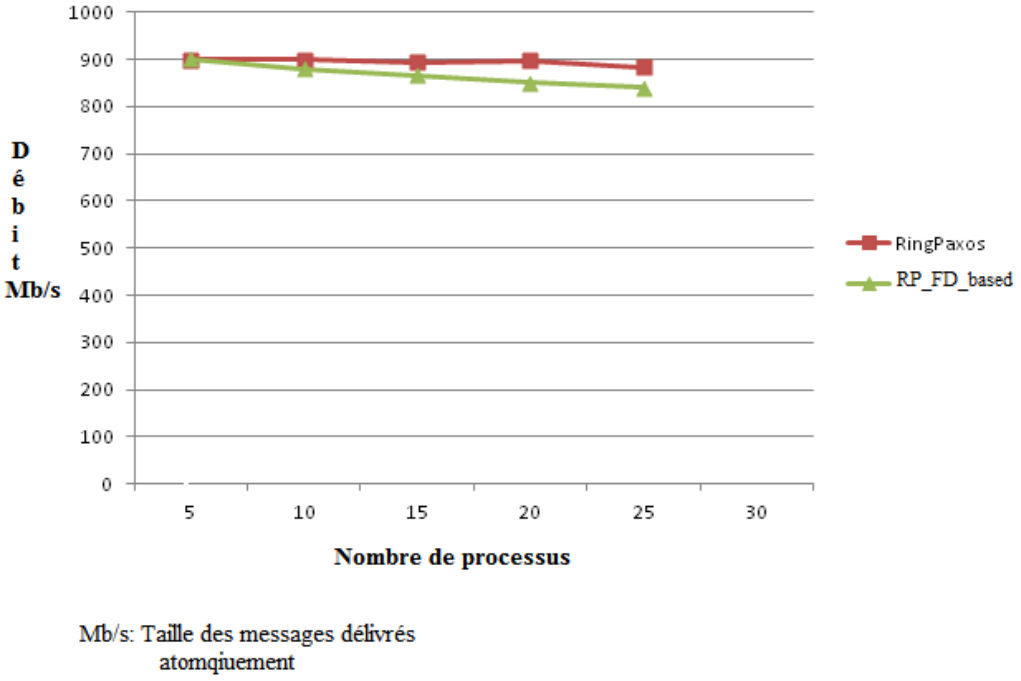

FIGURE 4.3 – Débit vs nombre de processus : Execution sans crash (λ=1)

Comme le montrent les deux graphes précédents, notre protocole offre des résultats bien

meilleur en ce qui concerne la latence par rapport à celui de *Chandra et Toueg*, mais par rapport au *Ring Paxos* notre protocole offre des résultats meilleur avec l'augmentation du nombre de processus. L'augmentation (dans notre protocole et celui de *Ring Paxos*) par rapport à celui de *Chandra et Toueg* est justifiée par l'utilisation du principe de jeton circulant dans un anneau logique. Tandis que la différence entre notre protocole et celui de *Ring Paxos* est remarquée suivant la variation de la taille du système et aussi l'utilisation du détecteur de défaillances augmente le nombre de messages de contrôle, ce qui influe directement le débit et la latence.

4.6.4.2 Execution avec crash des processus

L'exécution avec crash se passe dans un environnement ou il y'a possibilité d'occurrence de défaillances des processus. De ce fait, le nombre de processus défaillant toléré dans ce cas est *(f< n/2)* tel que n représente le nombre des processus du système. En effet, le tableau 4.1 récapitule les différentes valeurs de f dont nous avons considéré suivant les différentes valeurs du nombre de processus.

n	5	10	15	20	25
f	2	3	6	7	10

TABLE 4.1 – Les différentes valeurs de f selon le nombre de processus n

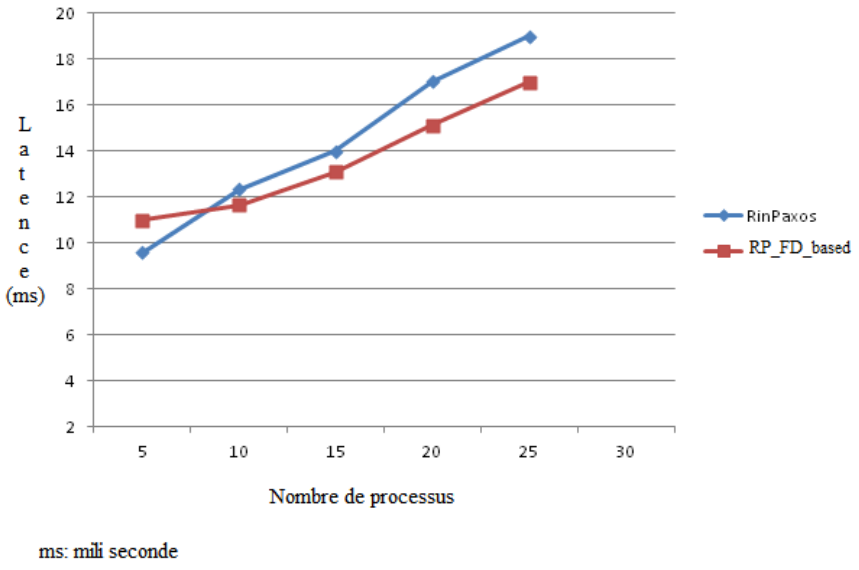

FIGURE 4.4 – Latence vs nombre de processus :Execution avec crash

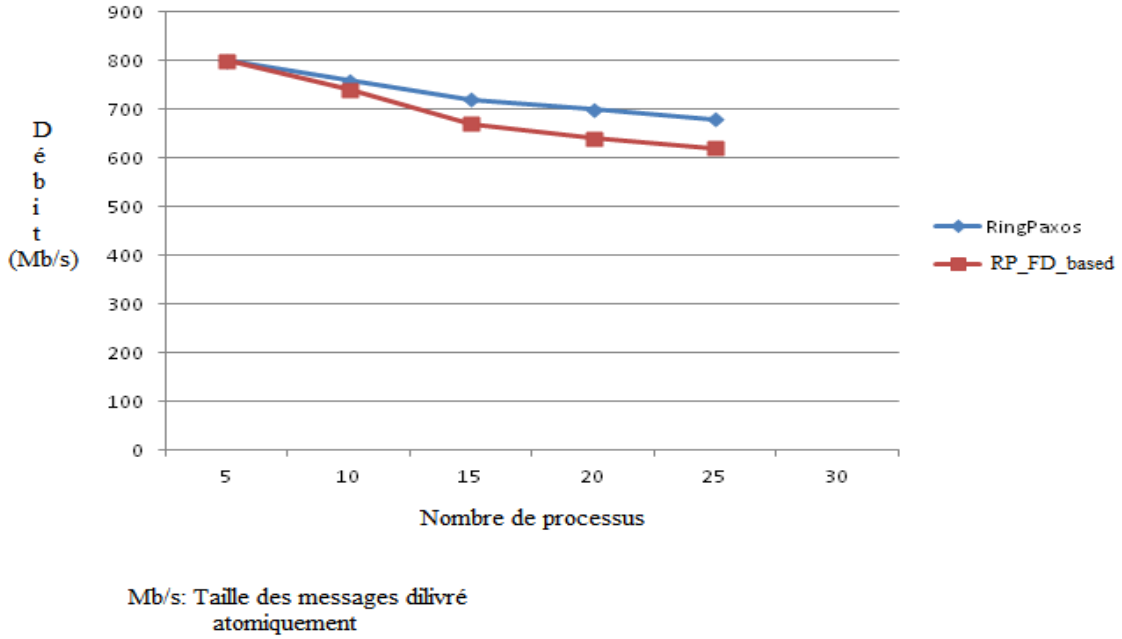

FIGURE 4.5 – Latence vs nombre de processus :Execution avec crash

Suivant les résultats illustrés par la figure (4.4), nous apercevons que notre protocole devient plus optimal avec l'augmentation du nombre de processus que celui du *Ring Paxos* en terme de latence. Par contre, dans la figure 4.5 en terme de débit, le protocole de *Ring Paxos* reste toujours un protocole de haut débit même en occurrence de crash de processus, qui est justifié par l'utilisation d'un coordinateur unique dans le protocole de *Ring Paxos*.

4.7 Conclusion

Dans ce chapitre, nous avons proposé un nouveau protocole résolvant le problème de la diffusion atomique dans un système distribué asynchrone. Ce protocole est une amélioration d'un autre protocole existant. Le protocole proposé est basé sur le détecteur de défaillances non fiable $\Diamond S$, le principe du jeton circulant et le coordinateur. Les résultats de simulation, obtenus en utilisant le simulateur *neko*, montrent que les protocoles basés sur la circulation du jeton sur un anneau logique donnent de meilleurs résultats en termes de débit et de latence par rapport aux autres protocole. Ainsi, l'utilisation dans cette même classe (à jeton), l'utilisation d'un seul coordinateur est meilleure par rapport à un coordinateur jeton mais seulement dans un réseau de petite taille.

Conclusion générale et Perspectives

LES problèmes d'accord ont une extrême importance dans la conception et la vérification des applications tolérantes aux fautes dans les systèmes distribués, ce qui a suscité des recherches intenses pour trouver des solutions efficaces à ces problèmes. La diffusion atomique est l'un de ces problèmes d'accord et qui représente une brique de base dans la conception des applications tolérantes aux fautes. Elle est définie par le fait que tous les processus, participants à une tâche dans le système, doivent délivrer le même ensemble de messages dans le même ordre. Plusieurs solutions ont été proposées pour le résoudre.

Dans ce travail, nous avons fait une étude exhaustive sur le problème de la diffusion atomique. Nous avons présenté des généralités sur les systèmes distribués et les problèmes d'accord, qui ont une relation directe ou indirecte avec le problème étudié. Par la suite, nous avons exhibé le problème de la diffusion atomique, nous avons donné une définition formelle et informelle de ce problème, nous avons présenté, aussi, quelques solutions de base, pour résoudre ce problème, qui diffèrent selon deux critères : le mécanisme d'ordre de messages utilisé et le mécanisme de détection de défaillances implémenté. Ces solutions présentent des inconvénients remarquables, ou bien l'utilisation d'un mécanisme de tolérance aux fautes très fort, ou bien l'utilisation d'un mécanisme d'ordre de messages qui augmente le trafic dans le système, ce qui a une influence directe sur la latence de ce système.

Pour remédier à ces inconvénients, nous avons proposé une solution au problème de la diffusion atomique. Cette solution se base sur le principe du coordinateur tournant qui essaie, après son élection par la liste des processus corrects générée par le plus faible détecteur de défaillances $\Diamond S$, de regrouper un ensemble de processus corrects dans un anneau logique et d'imposer la liste des messages à délivrer par cette même liste en collaboration avec le coordinateur, en faisant circuler un jeton qui transporte les informations nécessaires à la prise de décision.

Nous avons évalué les performances de notre proposition par la simulation. L'utilisation d'un simulateur des systèmes distribués existant, nommé neko, a été très bénéfique, du fait

qu'elle nous a permis de comparer les résultats obtenus de notre protocole avec d'autres résultats des autres solutions existants dans la littérature, celui de *Chandra et Toueg* [CT96] et celui de *Ring Paxos* [PMP12].

En guise de perspectives, nous souhaitons implémenter notre proposition sur un réseau réel, ce qui nous permettra de renforcer les résultats obtenus par la simulation. Nous souhaitons, aussi, proposer une autre solution au problème de la diffusion atomique, qui remédie à l'inconvénient de la surcharge du réseau par les messages de contrôle (votes), ainsi que l'utilisation d'un détecteur de défaillances équivalent à $\Diamond S$, nommé Ω , qui a un double travail, la détection de défaillances et la génération automatique d'un leader, ce qui évite d'élire un nouveau coordinateur à la demande des processus du système.

Annexe

1. Installation et configuration

Comme système d'exploitation on doit utiliser un environnement «Unix-Like» dont les dérivants de GNU/Linux, pour notre cas plus précisément on utilise la distribution Ubuntu 12.04 LTS Desktop.

L'installation et la configuration de Neko Nécessite l'installation des prérequis suivant :

- Java, JDK 1.5 au moins. (Pour installer Java sous Ubuntu, on doit installer openjdk-6 ou 7).
- Compilateur C.
- Perl.

1. Copier le fichier « neko-1-0-beta-1.tgz » vers le bureau.
2. Décompresser le en utilisant la commande (tar xzvf neko-1-0-beta-1.tgz), un dossier va s'apparaître sous le nom **neko**
3. Ouvrir le Terminal, puis entrer dans ce dossier par la commande :cd (Bureau|Desktop)/neko
4. tapper la commande suivante : ./configure
5. Puis ajouter le chemin de fichiers binaires (les commandes) apportés par le paquet d'installation **Neko** sur le variable d'environnement **PATH** par la commande suivante (changer ‹Bureau› par ‹Desktop› dans les version anglaises d'OS) : **export PATH=/home/nom-pc/Burea/dest-neko/bin : SPATH**
6. et a la fin, on doit procéder la compilation de simulateur par l'outil ant, il suffit de taper la commande : **ant**
7. Pour lancer une configuration on ne doit qu'accéder au chemin où se trouve, puis taper la commande suivante : java lse.neko.Main <fichier de config> .

Notes :

1- Après chaque réouverture du Terminale, on doit ajouter le chemin de fichiers binaire au variable d'environnement PATH à nouveau (étape 5).
2- tout les aspects pour l'exécution d'une application Neko sont définies dans un seul fichier de configuration (Nom-fichier.config) , Neko assure que chaque processus dans l'application

aura les information contenues dans le fichier de configuration (Nom-fichier.config).

2. Exemple d'exécution Après avoir installé Neko et ses prés requis voila un exemple d'exécution :

- Ouvrir le terminal.
- Taper dans le terminal : cd Bureau/neko
- ./configure
- ant
- cd lse/neko/consensus/tests
- java lse.neko.Main LatencyTest.config

Notes :

- Les résultats de simulation se trouvent dans un fichier "log " se trouvant dans le même dossier que le fichier de configuration.
- LatencyTest.config : c'est le fichier de configuration, il contient les propriétés pour l'exécution.

Bibliographie

[Abr88] K. Abrahamson. On achieving consensus using shared memory. In *7th ACM Symposium on Principles of Distributed Computing*, pages 291–302, 1988.

[ACT99] M.K. Aguilera, W. Chen, and S. Toueg. Using the heartbeat failure detector for quiescent reliable communication and consensus in partitionable networks. *Theoretical Computer Science*, pages 3–30, 1999.

[AMR01] M. Raynal A. Mostefaoui, S. Rajsbaum and M. Roy. Efficient condition-based consensus. *In the 8th international Colloquium on Structural Information and Communication Complexity*, pages 275–291, 2001.

[Ber99] D. Berard. Modèle de réseau pour la simulation d'algorithme distribués. *Technical report,LSE-EPFL*, 1999.

[BO83] M. Ben-Or. Another advantage of free choice :completely asynchronous agreement protocols. In *2nd ACM Symp. on Principles of Distributed Computing (PODC'83)*, pages 27–30, 1983.

[CBS00] B. Charron-Bost and A. Schiper. Uniform consensus is harder than consensus. *Technical Report DSC/200/028, EPLF*, 2000.

[CF99] F. Cristian and C. Fetzer. The timed asynchronous distributed system model. *IEEE,Transactions on Parallel and Distributed Systems*, pages 642–657, 1999.

[Cha90] S. Chaudhuri. Agreement is harder than consensus : Set consensus problems in totally asynchronous systems. In *Proceedings of the 9th annual ACM Symposium on Principles of Distributed Computing (PODC'90)*, pages 311–324, 1990.

[CHT96] T. Chandra, V. Hadzilacos, and S. Toueg. The weakest failure detectors for solving consensus. *Journal of the ACM*, pages 685–722, 1996.

[CM84] J. Chang and N. Maxemchuck. Reliable broadcast protocols. *In ACM Transactions on Computer systems (TOCS)*, pages 251–273, 1984.

[CT91] T. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *In the 10th Annual ACM Symposium on Principles of Distributed Computing*, pages 325–340, 1991.

[CT96] T. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *ACM*, pages 225–267, 1996.

[DDS87] D. Dolev, C. Dwork, and L. Stockmeyer. On the minimal synchronism needed for distributed consensus. *ACM*, pages 77–97, 1987.

[DLS88] C. Dwork, N. Lynch, and L. Stockmeyer. Consensus in the presence of partial synchrony. *ACM*, pages 288–323, 1988.

[EKW07] N.R. EKWALL. Atomic broadcast : a fault-tolerant token based algorithm and performance evaluations. *Phd thesis. Lausanne, EPFL*, 2007.

[FClD90] B. Dancey F. Cristian and l. Dehn. "fault-tolerance in the advanced automation system", ftcs 20th, newcastle, uk. pages 6–20, 1990.

[FLP85] F. Fisher, N. Lynch, and M. Paterson. Impossibility of distributed consensus with one faulty process. *ACM*, pages 374–382, 1985.

[GCV01] I. Keidar G. Chockler and R. Vitenberg. Group communication specifications : A comprehensive study,acm computing surveys. pages 427–469, 2001.

[Gra78] J. Gray. "notes on database operating systems" in "operating systems : an advanced course". *Lecture Notes in Computer Science, Springer,Verlag pub, No.60*, pages 393–481, 1978.

[Gre02] F. Greve. Réponses efficaces au besoin d'accord dans un groupe. *Thèse de doctorat ,Université de Renne 1*, 2002.

[GS01] R. Gerraoui and A. Schiper. The generic consensus service. *IEEE Transaction on Software Engineering*, pages 29–41, 2001.

[GV10] R. Guerraoui and M. Vukolic. Refined quorum systems. *Distributed Computing*, pages 1–42, 2010.

[HT93] V. Hadzilacos and S. Toueg. Reliable broadcast and related problems. *In Distributed Systems, ACM Press*, pages 97–145, 1993.

[HT94] V. Hadzilacos and S. Toueg. A modular approach to fault-tolerant broadcasts and related problems. Technical Report 94-1425, Computer Science Department, Cornell University, 1994.

[Lam77] L. Lamport. Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering*, page 125–143, 1977.

[Lam98] L. Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, pages 133–169, 1998.

[Mar04] C. Marchand. *Mise au point d'algorithmes répartis dans un environnement fortement variable, et expérimentation dans le contexte des pico-réseaux.* PhD thesis, Institut National Polytechnique de Grenoble, 2004.

[Mou03] E. Mourgaya. Les problèmes d'accord : Une approche comportementale. *Thèse de doctorat. Université de Rennes 1*, 2003.

[Mou05] H. Moumen. Evaluation de mécanisme de détection de défaillances dans un système répartie asynchrone. *Mémoire de magistère, université de Bejaia*, 2005.

[MRR02] A. Mostefaoui, S. Rajsbaum, and M. Raynal. A versatile and modular consensus protocol. In *Proceedings of the 2002 International Conference on Dependable Systems and Networks (DSN-2002)*, pages 364–373. IEEE, 2002.

[PDC02] E.P. Cortés P.Q. Duong and C. Collet. La tolérance aux fautes adaptable pour les systèmes à composants : application à un gestionnaire de données. *Laboratoire LSR-IMAG Martin Hères,FRANCE*, 2002.

[Pil06] J.F Pillou. Tout sur les réseaux et internet. *IEEE transaction on software Energineering*, 2006.

[PMP12] N. Schiper P.J. Marandi, M. Primi and F. Pedone. Ring paxos : A high-throughput atomic broadcast protocol. pages 1–12, 2012.

[PT86] K.J. Perry and S. Toueg. Distributed agreement in the presence of processor and communication faults. *IEEE Transaction on software Engineering*, pages 477–482, 1986.

[RB91] A. Ricciardi and K. Birman. Using process groups to implement failure detection in asynchronous environments. *In ACM Operating Systems Review*, pages 341–351, 1991.

[REB07] N. REBOUH. Evaluation par simulation d'un mécanisme de détection de défaillances dans un système distribué asynchrone. *Mémoire de magistère, université de Bejaia*, 2007.

[REU04] A. Schiper R. Ekwall and P. Urban. Token-based atomic broadcast :using unreliable failure detectors. In *In Proceedings of the International Symposium on Reliable Distributed Systems (SRDS)*, pages 6–20, 2004.

[SM95] L. Sabel and K. Marzullo. Election vs consensus in asynchronous systems. *Technical Report*, 1995.

[Tio08] A. Tioura. Implémentation de mécanismes de détection de défaillances dans un environnement byzantin. *Mémoire magistère, Université de Bejaia*, 2008.

[Urb03] P. Urban. Evaluating the performance of distributed agreement algorithms : Tools, methodology and case studies. *Phd thesis, lausanne, EPFL*, 2003.