République Algérienne Démocratique et Populaire

Ministère de l'Enseignement Supérieur et de la Recherche Scientifique.

Université A/Mira de Bejaia

Faculté des Sciences et des Sciences de l'Ingénieur

Département d'Informatique

Ecole Doctorale d'Informatique

Mémoire de Magister En Informatique

Option : **Réseaux et Systèmes Distribués**

Présenté par : TIOURA Abdel hamid

Thème

IMPLEMENTATION DE MECANISMES DE DETECTION DE DEFAILLANCES DANS UN ENVIRONNEMENT BYZANTIN

Soutenu le 28 – 02 – 2008

Devant le Jury :

Président	**:**	**KERKAR Moussa**	**Professeur, Université de Bejaia**
Rapporteur	**:**	**BELMEHDI Ali**	**Professeur, Université de Bejaia**
Examinateur	**:**	**AHMED NACER Mouhamed**	**Professeur, USTHB, Alger**
Examinateur	**:**	**BOUKERRAM Abdallah**	**M C, Université de Sétif**
Invité	**:**	**MOUMEN Hamouma**	**M A Université de Bejaia**

بسم الله الرحمن الرحيم

All Praise is due to Allaah. We praise Him, and seek His help and forgiveness. We seek refuge in Allaah, Most High, from the evils of our own selves and from our wicked deeds. Whomever Allaah guides cannot misguide, and whomever He leads astray cannot be guided. I testify that there is no true god worthy of being worshipped except Allaah, alone, without partner and associate. I further testify that Mohamed is his slave and messenger. My Allaah's salaah and salaam also be granted to the prophet's pure family and to all his noble companions.

Remerciements

Je tiens à remercier en premier lieu Mr A. Belmehdi Professeur à l'université de Bejaia d'avoir accepté d'être mon encadreur durant cette année de magistère, et pour la confiance qu'il ma donné et ses précieux conseils.

Je tiens à remercier vivement Mr Moumen Hamouma, maitre assistant à l'université de Bejaia, pour accepter d'être mon coencadreur, pour les longues discutions, les conseils judicieux ainsi que les heures passes sur ce document en regrettant peut-être les bons moments passés ensemble avant ce mémoire. J'espère bien lui être dune quelconque aide à mon tour

Mes remerciements vont également aux membres de jury d'avoir accepté de juger mon travail. J'adresse mes très sincères remerciements à : Mr Moussa Kerkar professeur à l'université de Béjaïa, Mr Ahmed Nacer Mouhamed, professeur à l'université de bejaia et Mr Boukarram Abdallah, maître de conférences à l'université de Sétif. je les remercie d'avance pour leurs critiques et suggestions.

Je remercie tous nos enseignants de l'école doctorale. Merci pour Mr A. Tari, chef de département d'informatique de l'université de Béjaïa, pour tout ce qu'il a fait pour l'école doctorale.

Merci pour la personne qui m'a encouragé dès ma naissance, qui m'a toujours aidé et conseillé, et qui a été toujours près de moi, merci à toi mon père Mohamed.

Je remercie Khergag Mohamed, Siam Abd-Errahim qu'ont apporté le sérieux de leurs commentaires et de leurs critiques, qu'ils en soient ici remerciés

Je remercie mon grand père, mes frères, Adel, farok et abdel hak, qui m'ont soutenu durant la préparation de ce mémoire. Je pense à mes sœurs et à mon petit frère Omar.

Je remercie tous les personnes qui m'ont soutenu durant la préparation de ce mémoire de prés ou de loin.

Dédicaces

Á mon grand père El hadj Amar

Á mon père Mohamed

Á mes frères Abdel hak, Adel et Farok

Á mes sœurs : Abella, Farida, Soria, Lamia et Soumia

Amon petit frère Omar

Á toute ma famille

Á mes frères Mohamed, Mazinio, Hamouma, Boudhiaf et Samir

Á tous les algériens

Á tous les musulmans

Á tous le monde

Résumé

La conception d'algorithmes distribués est un problème difficile en raison de la possibilité d'existence de défaillances partielles. Pour résoudre ce problème, Chandra et Toueg ont suggéré une approche modulaire où la détection est encapsulée à l'intérieur d'un oracle spécifique appelé détecteur de défaillances. Cette approche modulaire simplifie le raisonnement au sujet de la correction des algorithmes d'accord, mais elle a été proposée dans un contexte de défaillances crashs. Une question naturelle vient à l'esprit : Est-il raisonnable de suivre une approche similaire dans le contexte de défaillances byzantines ? En d'autres termes, y a-t-il une notion du détecteur de défaillances byzantines qui permet également une bonne modulation ?

Dans ce travail, nous nous intéressons à la conception des protocoles implémentant les détecteurs de défaillances. Nous présentons d'abord une synthèse sur les détecteurs de défaillances utilisés pour les systèmes asynchrones, puis nous donnons les deux protocoles implémentant les détecteurs de défaillances byzantines.

Ensuite, nous proposons deux protocoles nouveaux implémentant le détecteur de défaillances Byzantine *Omega*. Le premier est basé sur des hypothèses de synchronie faible, et le deuxième est basé sur des hypothèses qui n'utilisent pas le temps physique.

Mots clé : *Systèmes distribués asynchrones, algorithme distribué, tolérance aux fautes, processus byzantin, Omega, détecteurs de défaillances, consensus, synchronie.*

Abstract

Devising distributed algorithms is a difficult problem because of the existence possibility of partial failures. To resoling this problem, Chandra and Toueg suggested a modular approach where failure detection is encapsulated inside a specific oracle called failure detector. This modular approach simplifies the reasoning about the correctness of agreement algorithms, but it has been proposed in a context of crash-failures. A natural question comes to mind: does it make sense to follow a similar approach in the context of Byzantine failures? In other words, is there a notion of Byzantine failure detector that also enables a nice modularization?

In this work, we are interesting in the design of protocols implementing failure detectors. We first present a synthesis about failure detectors used for asynchronous systems, then we present the tow protocols implementing Byzantine failure detectors. Then, we propose tow new protocols implementing a Byzantine failure detectors Omega. The first is based on weak synchrony assumption, and the second is based on weak time-free assumptions.

Keywords: *Asynchronous distributed systems, Distributed algorithm, Fault tolerance, Byzantine process, Omega, failure detectors, consensus.*

TABLE DES MATIERES

Liste des Figures

Liste des Tableaux

Liste des Algorithmes

INTRODUCTION GENERALE

Les systèmes distribués tiennent une place de plus en plus importante dans le monde actuel, en particulier avec l'avènement de l'Internet. Informellement, ils représentent une abstraction dans laquelle un ensemble d'entités coopèrent afin d'effectuer une tâche ou d'offrir un service donné. Un système distribué typique peut être structuré comme un ensemble de processus, s'exécutant dans des postes différents, fonctionnant selon des conditions fonctionnelles spécifiques.

Pendant que les systèmes deviennent plus distribués, ils deviennent également plus complexes et doivent traiter de nouveaux genres de problèmes tels que la défaillance des processus et des canaux de communication. Ainsi, gérer les systèmes distribués est une tâche difficile puisqu'on doit traiter la transmission distante et les divers types de défaillances qui peuvent résulter de la distribution. L'avantage principal d'un système distribué est sa tolérance aux défaillances. Dans un système distribué, les services peuvent être répliqués sur plusieurs ordinateurs, ainsi la panne d'un ordinateur n'affecte pas le fonctionnement du système. Cet avantage a une importance fondamentale pour développer des systèmes qui fournissent des services fiables.

Dans un système distribué, la tolérance aux fautes est basée sur la réplication et les protocoles d'accord [32]. En répliquant les composants critiques du système, nous rendons la totalité du système plus fiable que ses parties. La réplication des composants soulève une question sur la manière de coordonner entre ces répliquas. Ceci est réalisé grâce aux protocoles d'accord. Un tel accord est nécessaire pour garantir la cohérence du système, par exemple, un seul serveur logique peut se comporter comme un groupe de serveurs répliqués. Beaucoup de problèmes d'accord sont liés au problème de consensus [61, 16].

Le consensus est un paradigme fondamental pour les calculs distribués tolérants aux fautes. Informellement, le consensus permet aux processus de prendre une décision commune, qui dépend de leurs valeurs initialement proposées, en dépit des défaillances. Le consensus peut être utilisé comme une brique de base pour résoudre plusieurs autres problèmes d'accord, tels que la diffusion atomique [35], la validation atomique [37], la gestion de groupes [7], l'élection d'un leader [61].

Le problème du consensus est sujet à des recherches intensives. Fisher, Lynch, et Paterson ont montré qu'il est impossible de résoudre le problème de consensus d'une manière

déterministe, dans un système distribué asynchrone avec des canaux de communication fiables, même s'il y a un seul processus défaillant [25].

Il est très difficile de raisonner correctement sur un problème dès lors que des éléments non maîtrisables entrent en jeu. L'idée est alors souvent d'isoler la zone d'incompréhension. Cela permet de réfléchir efficacement au problème. Ensuite nous pouvons traiter de manière indépendante la zone d'incompréhension que nous avons isolée.

Les oracles proposés par Chandra et Toueg [16] remplissent cette fonction d'isolement. Ils vont encapsuler une partie de problème que nous ne savons pas résoudre, ou dont la solution est très complexe. Les oracles ou détecteurs de défaillances de Chandra et Toueg ont été proposés pour contourner le résultat d'impossibilité de FLP.

En englobant les propriétés nécessaires à la résolution d'un problème donné, les oracles permettent de clairement exhiber ces propriétés. Cela permet de classifier les problèmes par les propriétés que ces problèmes nécessitent, autrement dit par la puissance des oracles qui permettent de les résoudre.

Dans le contexte des systèmes asynchrones soumis aux défaillances, des oracles devinent les défaillances. Ils permettent ainsi de résoudre de multiples problèmes, dont les problèmes d'accord. Parmi ces problèmes d'accord se trouve le consensus que nous allons utiliser pour introduire les oracles dans le contexte des systèmes asynchrones.

Les détecteurs de défaillances sont des modules non fiables qui fournissent aux processus une liste des processus suspectés d'être défaillants. Un module de détecteurs de défaillance peut commettre des erreurs par la non suspicion d'un processus crashé ou par la suspicion d'un processus correct. Formellement les détecteurs de défaillances sont définis par deux propriétés : la complétude (concernent la détection de processus crashé), et la précision (qui restreint les erreurs sur les suspicions erronées).

Plusieurs implémentations de détecteurs de défaillances crashs ont été proposées dans la littérature. Ces implémentations se basent sur des hypothèses de synchronie ou sur des hypothèses sur le modèle d'échange de messages.

Implémenter les détecteurs de défaillances avec la présence de l'asynchronisme et les défaillances byzantines est une tâche très ardue. Ce modèle est le plus sévère dans les systèmes distribués. A cause de difficulté dans tel type de système, seulement deux implémentations qui ont été proposées : celle de Doudou et al [23] et celle de Kihlstrom et al

[36]. Ces deux implémentations se basent sur un système où tous les canaux sont inéluctablement synchrones

Rechercher des hypothèses de synchronie plus faibles, minimales, ou plus réelles reste une piste très importante dans le domaine de systèmes distribués. Pour cela, Moumen, Mostéfaoui et Trédan [44] ont défini une hypothèse très faible que celles existantes. Ils ont utilisé celle-ci pour résoudre d'une façon déterministe le consensus byzantin. Cette hypothèse exige seulement la présence de *4t* canaux de communication inéluctablement synchrone (*t* est le nombre maximum de processus qui peuvent être défaillants).

Dans ce mémoire, nous reprenons cette hypothèse pour implémenter le détecteur Omega Byzantin et nous allons définir une autre hypothèse sur le comportement du système pour implémenter le même détecteur.

Le présent mémoire est divisé en cinq chapitres. Dans le premier chapitre, nous présentons des concepts généraux sur les systèmes répartis et les problèmes d'accord et en particulier le problème du consensus. Dans le deuxième chapitre, nous présentons une des solutions au problème du consensus en utilisons les détecteurs de défaillances. Nous présentons d'abord la notion de détecteur de défaillances, ses caractéristiques, et un protocole l'implémentant. Dans le troisième chapitre, nous présentons les deux implémentations existantes de détecteurs de défaillances byzantines et nous comparons entre celles-ci. Dans le quatrième, nous proposons un protocole implémentant un détecteur de défaillances Omega Byzantin avec des hypothèses de synchronie très faibles. Dans le dernier chapitre, nous proposons un autre protocole implémentant Omega Byzantin, mais avec des hypothèses qui n'utilisent pas le temps physique. Enfin, ce mémoire se termine par une conclusion et des perspectives.

CHAPITRE I

MODELES DE SYSTEMES DISTRIBUES ET PROBLEMES D'ACCORD

Introduction

La mise en œuvre et la gestion des systèmes distribués sont des tâches difficiles à réaliser, puisqu'on doit traiter la transmission distante et les divers types de défaillances qui peuvent résulter de la distribution. Pour vaincre ces difficultés, il est nécessaire de prévoir des mécanismes qui permettront au système de continuer à fonctionner malgré les défaillances d'un nombre limité de ses composants. Ces mécanismes sont aussi nécessaires pour résoudre tous les problèmes des systèmes distribués, qui nécessitent la coordination des processus du système. L'un des mécanismes les plus importants est la réalisation des détecteurs de défaillances. Un détecteur de défaillances est un ensemble de modules locaux, chacun lié à un processus, qui fournit des informations sur les processus défaillants du système.

L'objectif de ce premier chapitre est de présenter la problématique qui est à l'origine des détecteurs de défaillances, c'est-à-dire les contraintes de synchronismes et les problèmes qui nécessitent la présence d'un détecteur de défaillances pour qu'ils soient résolus.

Dans un premier temps, nous allons présenter un certain nombre de notions et de caractéristiques des systèmes distribués. Ensuite, nous présenterons certains problèmes d'accord et nous nous concentrons sur le problème du consensus qui représente une forme abstraite de tous ces problèmes, c'est-à-dire tous les autres problèmes sont réductibles au consensus.

1.2 Processus et canaux de communication

Typiquement, un système distribué se compose d'un ensemble de processus échangeant des messages via des canaux de communication. Un processus peut être correct ou incorrect, s'il fonctionne selon sa spécification il est correct sinon il est incorrect (défaillant), les canaux de communication aussi peuvent subir des défaillances. Nous décrivons dans la partie suivante quelques types de défaillances des processus, puis les défaillances des canaux de communication.

1.2.1 Modèles de fautes des processus

Les processus d'un système réparti sont susceptibles d'être défaillants, ils peuvent subir plusieurs types de fautes. Ces fautes peuvent être classées selon trois familles principales :

***Fautes par arrêt définitif ou crash (fail-stop)* :**

Le processus se comporte conformément à sa spécification jusqu'à ce qu'il subisse une défaillance franche, après laquelle il arrête toute ces activités (tous ses calculs sont stoppés et il ne reçoit et n'envoie aucun message). Son arrêt est définitif, il ne peut participer au fonctionnement de système. Ce type de faute a été introduit dans [25].

***Fautes par omission (omission failure)* :**

Le processus cesse momentanément son activité puis reprend son activité normale. Ces activités concernent **l'envoi** et la réception des messages. Typiquement, cela peut correspondre à une perte de message [54].

Fautes de performance :

Ce type de défaillance se caractérise par un non respect des délais d'exécution des tâches de la part du processus. Les fautes par omission sont des cas particuliers de fautes de performance.

***Fautes arbitraires ou byzantines (byzantine failure)* :**

Un processus présentant ce type de défaillance agit d'une manière complètement incontrôlable et imprévisible. Ce type de faute est le plus général, aucune hypothèse ne peut être faite sur le comportement d'un processus byzantin. L'étude de ce type de défaillance est très utile pour la construction des systèmes sécurisés : un système tolérant des processus byzantins tolère n'importe quels autres types de fautes commises par les processus. Comme

nous pouvons le voir sur la Figure 1, chaque type de faute est imbriqué dans un autre. Les fautes par arrêt et par omission sont des cas particuliers de fautes arbitraires (byzantines).

Dans ce mémoire, notre but est de tolérer les fautes arbitraires (byzantines) commises par les processus. Des définitions plus détaillées et les techniques bien précises sur les fautes arbitraires serons données dans les parties qui suivent. Les fautes byzantines sont plus générales que les autres et englobent tous genres de fautes, ce qui rend très délicat la résolution de leurs problèmes ainsi que la preuve des solutions proposées.

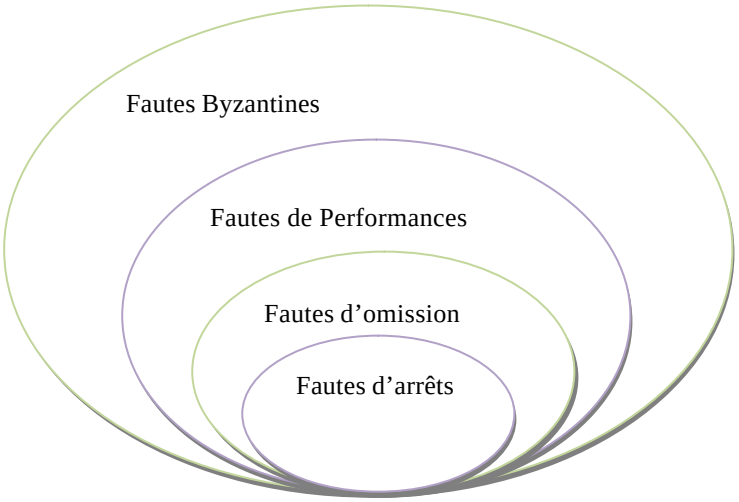

Figure 1 : Imbrication des types de fautes

1.2.2 Les défaillances des canaux de communication :

Les canaux de communication peuvent subir plusieurs types de fautes qui sont :

- la destruction du canal ;
- la perte de message ;
- la duplication de message ;
- la corruption de message.

Selon ces fautes, les canaux de communication peuvent être classés en :

- **Canaux fiables :** si un processus *p* envoie un message *m* à un processus correct *q*, alors ce dernier reçoit le message *m*. Habituellement, les canaux fiables sont implémentés en utilisant des techniques de retransmission (le message est retransmis par l'émetteur jusqu'à ce que le récepteur supposé correct le reconnaisse). Le crash potentiel de l'émetteur rend difficile l'implémentation d'un tel modèle de canal. Dans la pratique, la conception des algorithmes distribués est basée sur un type plus faible de canal, appelé canal quasi-fiable.

- **Canaux quasi-fiables :** si un processus correct *p* envoie un message *m* à un processus correct *q,* alors ce dernier reçoit le message *m.*

- **Les canaux fiables avec fautes de performance :** dans ce type de canaux, les délais de transmission peuvent être mis en défaut. En effet, avec cette hypothèse, nous n'assurons nullement la réception des messages en un temps borné

1.3 Les modèles de réseaux

Dans les systèmes répartis, les processus et leurs interconnexions forment une structure logique, qui peut être modélisée formellement par un graphe *G(S, A),* où *G* dénote le réseau, *S* l'ensemble des sommets égal à *?* = {P_1…..…P_n}, et *A* l'ensemble des arrêtes identifiant les interconnexions inter-processus. Cette représentation des réseaux par les graphes ne permet pas de modéliser leur évolution temporelle ; c'est pour cela qu'on distingue deux types de réseaux :

Les réseaux statiques sont des réseaux dont l'ensemble des processus *?* est statique et tous les canaux de communication sont connus initialement. Dans ce mémoire nous étudions les réseaux statiques finis représentables par un graphe complet, ce qui veut dire que l'ensemble des processus ? est fini et pour toute pair de processus *(P_i, P_j)*, le canal de communication <*ij*> existe.

Les réseaux dynamiques par opposition aux réseaux statiques sont des réseaux où les processus et les canaux de communication peuvent être ajoutés ou supprimés à tout instant. Ces réseaux peuvent être représentés par un graphe à un instant donné *t*. Nous pouvons citer comme exemples de réseaux dynamiques Internet, *P2P* ou les réseaux de capteurs.

1.4 Les modèles temporels de communication par message

Les modèles de communication par message sont caractérisés par des hypothèses sur les délais de transmission des messages, les vitesses relatives des processeurs et sur la dérive des horloges. Selon ces métriques, nous pouvons décomposer les systèmes répartis en trois modèles : le modèle synchrone, asynchrone et le modèle partiellement synchrone.

1.4.1 Le modèle synchrone

Ce type de modèles possède les plus fortes hypothèses de synchronie, ce qui veut dire qu'il existe une borne supérieure *?* sur les délais de transmission des messages et *F* pour la vitesse relative des processus (quand un processus exécute une étape, le processus le plus rapide exécute au maximum *F* étapes), ainsi les processus ont une horloge synchrone [35]. En présence de ces hypothèses fortes, les processus peuvent synchroniser toutes leurs actions, ce qui permet la résolution des problèmes d'accord, tel que le consensus [27, 62], parce que les défaillances des processus sont plus faciles à détecter en utilisant le mécanisme de Timeouts.

Ce modèle présente un inconvénient major, les hypothèses de synchronie (bornes *?* et *F*), peuvent être violées par les algorithmes, s'elles n'ont pas été définies avec prudence, ce qui peut mettre la propriété de sûreté des algorithmes en défaut.

1.4.2 Le modèle asynchrone

Ce modèle est plus général que le synchrone (un modèle synchrone est aussi asynchrone), ce type de système ne connaît aucune borne sur les délais de transmission des messages ni sur les vitesses relatives des processus, ce qui rend ce type de système plus faible.

Ceci implique qui donne que plusieurs problèmes n'ont pas de solutions déterministes dans ce modèle, comme il a été présenté par Fisher, Lynch et Paterson : un accord entre les processus de système ne peut pas avoir lieu lors de la présence d'au moins un processus crashé [26], ceci est causé par la difficulté de distinguer entre un processus lent et un processus en panne dans le cas du modèle asynchrone.

1.4.3 Le modèle partiellement asynchrone

Pour contourner ce résultat d'impossibilité, une approche a été proposé par Dolev et Al [21], dans laquelle ils ont défini 32 modèles partiellement asynchrones par l'introduction de cinq critères, chaque critère pouvant prendre une des deux valeurs binaires (vrai ou faux), parmi ces critères, il ya un qui dénote que si le délai de transfère d'un message est borné, alors le critère est vrai, sinon il est faut.

Quatre modèles minimaux parmi les 32 ont été identifiés par Dolev et Al [21], sur lesquels le problème de consensus est solvable :

- Dans le premier modèle M_1 (qui a été utilisé par Dwork et al [21] pour résoudre des problèmes d'accord), il existe un moment appelé temps global de stabilisation (GST),

après lequel dans chaque exécution de système, il existe aussi des bornes connues sur les délais de transmission des messages et les vitesses relatives des processus.

- Dans le deuxième modèle M_2, à chaque exécution du système, il existe une borne sur les délais de transmission des messages et sur les vitesses relatives des processus, mais ces bornes ne sont pas connues.

- Dans le troisième modèle M_3, il existe un moment non connu appelé temps global de stabilisation (GST), des bornes non connues sur les délais de transmission des messages et sur les vitesses relatives des processus. Ce modèle a été utilisé par Chandra et Toueg [17], dans lequel ils ont proposé une généralisation des deux premiers modèles.

  Dans ce mémoire, on considère le système comme étant partiellement asynchrone dans lequel les bornes ne sont pas connues ; le système considéré fait partie de la famille du troisième modèle.

- Le quatrième modèle M_4 a été proposé par Cristain et Fetzer [15], qui supposent l'existence de périodes assez longues après lesquelles des bornes sur les délais de transmission des messages et les vitesses relatives des processus sont connues

1.5 Les problèmes d'accord

Les problèmes d'accord constituent une classe fondamentale dans les systèmes répartis ; ils suivent tous un modèle commun. Tous les participants au protocole doivent se mettre d'accord sur une décision commune, dont la nature dépend d'un problème spécifique. Par exemple : la décision va être l'ordre de délivrance des messages ou le résultat (commit ou Abort) d'une transaction distribuée.

1.5.1 La diffusion atomique

La diffusion atomique est une extension de la diffusion fiable : elle assure que tous les processus délivrent le même ensemble de messages. Elle assure aussi que tous les processus délivrent les messages dans le même ordre. La diffusion atomique est donc une diffusion fiable avec une fonction d'ordre total.

La diffusion atomique est définie par deux primitives *Broadcast()*, pour diffuser un message, et *Deliver()* pour délivrer un message envoyé.

Formellement, la diffusion atomique est caractérisée par les quatre propriétés suivantes [36] :

- **Validité (validity)** : si un processus correct diffuse un message *m* alors tous les processus corrects délivrent inéluctablement[1] *m*
- **Accord (agreement)** : si un processus correct délivre un message de diffusion *m* alors tous les processus corrects délivrent inéluctablement *m*
- **Intégrité (integrity)** : pour chaque message de diffusion *m*, tout processus correct délivre *m* au plus une fois, et seulement si *m* a été antérieurement diffusé par l'émetteur de *m*
- **Ordre total** : Si deux processus corrects P_1 et P_2 délivrent tous les deux m_1 et m_2, alors P_1 délivre m_1 avant m_2 si et seulement si P_2 délivre m_1 avant m_2. Figure .2

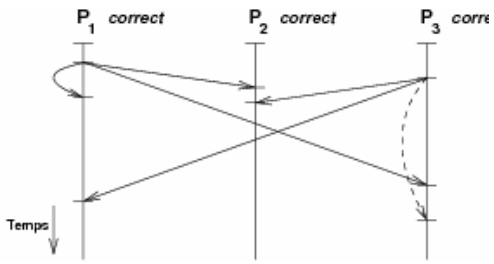

Figure 2 : Propriété d'ordre total

Chandra est Touag [17] ont montré que le consensus et la diffusion fiable sont deux problèmes équivalents, ce qui veut dire qu'une solution à l'un implique une solution à l'autre [21] ; informellement résoudre la diffusion atomique en utilisant le consensus consiste à effectuer une décision sur les messages à délivrer et sur l'ordre de ces messages.

1.5.2 La gestion de groupes

La gestion de groupes est un paradigme très puissant qui vise la conception et l'implantation des applications et des services dans les systèmes répartis tolérants les fautes. Ce paradigme est offert aux concepteurs des applications répartis par plusieurs systèmes ([54] explique quelques systèmes). La gestion de groupes permet aux développeurs d'applications réparties, de construire un service fiable ou d'implémenter un calcul fiable au haut niveau des systèmes répartis non fiables.

Le concept de groupe intègre trois services de base [16, 19] : un service de calcul de la composition du groupe, un service de communication de groupe et un service de synchronisation de vues. De manière générale, le service de calcul de composition du groupe gère la composition et la maintenance de la liste des processus, appelée ***vue***, **formant** le

[1] Inéluctablement : après certain temps

groupe. Le service de communication de groupe garantie la livraison des messages pour les membres de groupe. Le service de synchronisation de vues permet aux membres du groupe de s'entendre sur les messages qu'ils délivrent au sein d'une même vue.

Les trois services de concept de groupes sont considérés comme des problèmes d'accords, ce qui implique que chaque service est réductible aux consensus.
Dans la suite nous présentons les propriétés formelles du service de composition de groupe :

La composition de groupe produit aux processus la composition actuelle de groupe, appelle vue (*view*) ; qui peut être changée par l'ajout, la suppression, l'exclusion et le crash d'un processus.

Formellement, ce service est défini par les propriétés [36] suivantes :

- ***GM-Terminaison :*** Si un processus $p_k \in v_i(g)$ quitte le groupe ou tombe en panne, alors inéluctablement une nouvelle vue $v_i - \{p_k\}$ ou *nil* ou *fin* remplacera v_i. Si un processus correct p_k joint le groupe, alors il existe au moins un processus p_i qui installe v contenant p_k ;

- ***GM-Accord :*** Si un processus $p_k \in v_i(g)$ installe $v_{i+1}(g)$ et un processus $p_{k'} \in v_i(g)$ *installe* $v'_{i+1}(g)$, alors $v_{i+1}(g) = v'_{i+1}(g)$;

- ***GM-Validité* :** Si aucun processus n'est suspecté et un nouveau processus p_k entre dans le groupe, alors si un processus installe $v_{i+1}(g)$, alors $v_{i+1}(g) = v_i(g) \cup \{p_k\}$. Si aucun processus n'est suspecté et $p_k \in v_i(g)$ quitte le groupe, alors si un processus installe $v_{i+1}(g)$, alors $v_{i+1}(g) = v_i(g) - \{p_k\}$.

Tels que :
g : désigne un groupe.
$V_i(g)$: désigne la vue courante du groupe g.
nil : signifie qu'aucune vue n'est installée.
fin : désigne que la vue est finale et que les processus cessent de participer à ce groupe.

La gestion de groupe ne peut être résolue dans un environnement asynchrone [19], mais les services de concepts de groupes se réduisent au consensus, ce qui permet sa résolution, comme la solution proposée par Guerraoui et Schiper [32].

1.5.3 L'élection d'un leader

L'élection d'un leader est un autre problème fondamental dans les calculs distribués, il a fait l'objet de plusieurs travaux [13, 14, 15, 18, 20, 27, 30,].

Dans le problème d'élection d'un leader [62], à tout moment, au plus un processus se considère comme le leader et un nouveau leader doit être élu si le leader tombe en panne. Pour déterminer plus précisément la notion de coordonnateur (leadership), nous supposons que chaque processus a une copie locale d'une variable distribuée, dénotée par *leader*. La copie de *leader* pour un processus p_i est dénotée par $leader_{pi}$, et pour n'importe quel processus p_i $leader_{pi} \in \{vrai, faux\}$.

Nous disons qu'un processus p_i est le *leader* à l'instant *t*, si p_i n'est pas défaillant à l'instant *t* et $leader_{pi} = vrai$. Formellement, nous définissons le problème d'élection d'un leader par les deux propriétés suivantes :

- **Accord** : à l'instant *t* il existe un seul processus leader (deux processus ne peuvent pas être leader en même temps).
- **Terminaison :** à tout moment, il existe finalement un leader.

Sabel et Marzullo [62] ont prouvé que le problème d'élection d'un leader est réductible au problème de consensus. Informellement, une utilisation du consensus pour résoudre l'élection d'un leader consiste à effectuer une décision sur le leader à élire.

En étudiant les problèmes : diffusion atomique, gestion de groupes et l'élection d'un leader, on constate que le consensus est le représentant générique d'une large classe des problèmes d'accord, ce qui fait d'une solution au problème consensus une brique de base pour résoudre les problèmes d'accord

1.6 Le problème de consensus

Le problème du consensus est un paradigme fondamental dans les systèmes distribués tolérants aux défaillances. La recherche de solutions aux problèmes engendrés par la gestion de modèle asynchrone pour avoir la tolérance aux défaillances, a ramené les chercheurs à un problème à la fois abstrait (ce qui permet de le réutiliser dans des situations différentes), et plus simple à énoncer donc plus facile à conceptualiser et à manipuler.

Ce problème fondamental a été décrit pour la première fois par Lamport dans [42], C'est le problème de consensus qui est défini de la manière suivante :

1.6.1 Le consensus

Dans le problème de consensus, chaque processus correct propose une valeur V_i, et tous les processus corrects doivent atteindre une décision commune sur une valeur V.

Cette valeur doit être proposée par au moins un processus du système. Le consensus est implémenté par l'usage de deux primitives, *propose*(V) et *décide*(V). Quand un processus exécute *propose*(V), on dit qu'il propose V ; similairement, quand un processus exécute *décide(V)*, on dit qu'il décide sur V.

- ***Terminaison :*** tout processus doit finir par décider
- ***Validité :*** si un processus décide une valeur V, alors V a été proposé par au moins un processus
- ***Accord :*** deux processus ne peuvent décider différemment

1.6.2 Instances du problème

Le problème de consensus a plusieurs variantes qui diffèrent les unes des autres par au moins une des propriétés du consensus. Parmi ces instances nous avons :

Le consensus probabiliste a été proposé par Ben-Or [10], il diffère du consensus par la propriété de terminaison définie informellement comme suit : *la probabilité que tous les processus corrects décident est égale à1*.

Cette propriété est notée par *R-Terminaison*[1] et s'énonce comme suit :

- ***R-Terminaison*** tous les processus corrects décident avec une valeur égale à 1.

Cette version du consensus est plus facile à résoudre que la version originale.

Le K-consensus "K-set agreement" consiste en un accord sur au plus K ($K \in [1, n]$) valeurs différentes [13]. Il diffère du consensus par la généralisation de la propriété d'accord par :

- ***K-accord*** : le nombre de valeurs décidées est au plus k.

Le consensus uniforme *:* il diffère du consensus original par la propriété d'accord [14, 36, 57, 58] ; l'accord uniforme s'énonce comme suit :

[1] *R-Terminaison* est connu par Random-Terminaison en anglais,

- ***Accord uniforme : si*** un processus décide une valeur *v*, et de même si un autre processus décide une valeur *v'*, alors *v=v'*.

1.6.3 Le résultat d'impossibilité

Les résultats exposés précédemment montrent comment il est intéressant de trouver une solution au problème de consensus. Malheureusement Fisher, Lynch et Paterson ont prouvé dans [26], que le problème de consensus n'avait pas de solution entièrement déterministe dans le cas où, même un processus pouvait être défaillant. Ce résultat tient de fait qu'il n'est pas possible de distinguer d'une manière absolument fiable un processus très lent (ou avec lequel les communications sont très lentes), d'un processus défaillant. Dans ces conditions, toute solution entièrement déterministe peut être poussée à la faute, en l'obligeant à prendre une décision sans l'accord d'un processus très lent, qu'elle considérera comme défaillant, celui-ci peut décider de son coté (et en respectent complètement l'algorithme), une valeur différente sur la base des informations dont il dispose.
Ce résultat pourrait sembler au premier abord condamner l'utilité de chercher une solution aux problèmes d'accord par le biais de consensus. Ce qui a poussé à trouver un moyen de contourner ce résultat d'impossibilité.

1.6.4 Contourner le résultat d'impossibilité

Le besoin d'une solution au problème de consensus à amener les chercheurs à trouver des approches permettant de contourner le résultat d'impossibilité du consensus et de trouver des solutions à ce problème dans un environnement asynchrone.
Ces approches (techniques) peuvent être décomposées en trois classes :

- Approche basée sur le non-déterminisme.
- Approche Contrainte par conditions.
- Approche basée sur l'emploi d'oracles.

Chacune de ces techniques peut se recouper, et il existe des algorithmes les mélangeant habilement de façon à obtenir des solutions plus performantes
Dans ce qui suit, nous donnons plus de détails concernant ces techniques.

1.6.4.1 Le Non-déterminisme

Cette technique se traduit par l'algorithme de Ben-Or [10], qui apporte une solution originale au problème du consensus. Afin de contourner le résultat d'impossibilité attaché au problème de consensus. Ben-Or utilise un algorithme non-déterministe. Il a modifié la propriété de terminaison par la propriété *R-Terminaison* (énoncée à la section 1.6.2).

Cet algorithme nécessite qu'une majorité de processus soient corrects pour terminer. Il fonctionne par ronde (ou rounds) composés chacun de deux phases. Chaque processus utilise une variable _ contenant l'estimation de la valeur finale. Initialement cette valeur contient la proposition du processus. Lors de la première phase, chaque processus diffuse son estimation de la valeur finale. Après quoi, chacun d'eux attend de récolter une majorité de valeurs.

Note : durant la première phase, on suppose qu'une majorité de processus sont corrects et que les canaux de communications sont fiables.

En résumé, pour une ronde donnée, à la fin de la première phase les processus ont stockés dans la variable auxiliaire _ , soit une valeur indéfinie (aucune des valeurs proposées n'est majoritaire), soit une valeur unique dans le système (l'une des valeurs proposées est majoritairement représentée). Après cela, les processus participent à la seconde phase du ronde, en diffusant le contenu de leurs variables. Puis les processus procèdent de nouveau à une phase de synchronisation, en attendant une majorité de valeurs.
Trois cas peuvent se produire :

- Le cas plus favorable qui peut se produire est que le processus déicide sur la valeur reçue (cas ou les processus reçoivent des valeurs significatives), il diffuse cette décision d'une manière fiable.

- Dans le second cas, le processus reçoit à la fois des valeurs significatives et des valeurs non significatives. Il stocke sa valeur dans la variable d'estimation du processus, et entame ensuite une nouvelle ronde.

- Dans le dernier cas, le processus ne reçoit aucune valeur significative, il fixe alors la valeur de sa variable d'estimation en procédant à un tirage aléatoire.

La terminaison de cet algorithme repose sur le fait que la probabilité que tous les processus démarrent une ronde avec la même valeur initiale n'est pas nulle. Ceci est dû au fait qu'en cas de non convergence de l'algorithme, un pas de calcul non déterministe est effectué

(tirage aléatoire de la prochaine valeur initiale). Ceci assure que cet événement arriva avec probabilité égale à 1, même pour cela l'algorithme doit effectuer un grand nombre de ronde.

1.6.4.2 Approche contrainte par conditions

Elle a été proposée par Mostéfaoui et al [48, 49, 50, 52], cette approche permet d'identifier les vecteurs contenant les propositions des processus, à partir desquels le problème de consensus peut être résolu sans ajout de nouvelles hypothèses au système.

Une condition regroupe un ensemble de vecteurs de propositions pour lesquels, il existe un protocole déterministe résolvant le consensus. Formellement, une condition est caractérisée par un prédicat P qui caractérise l'appartenance d'un vecteur à la condition, et une fonction déterministe S retourne une valeur déterministe du vecteur.
Une condition est dite *f-acceptable,* si les prédicats P et les fonctions S correspondants caractérisent le vecteur à partir duquel il existe un protocole résolvant le consensus en dépit de f processus défaillants.

Nous nous intéressons à deux conditions *C1* et *C2* : *C1* engendre une décision sur la plus grande valeur proposée, tandis que *C2* permet une décision sur la valeur la plus fréquemment proposée.

1.6.4.3 Approche basée sur l'emploi des oracles « détecteurs de défaillances »

Cette solution proposée par Chandra et Toueg [16], consiste à augmenter le système par des mécanismes appropriés à la résolution du problème de consensus appelés oracles. Ces oracles peuvent être définis comme un ensemble de modules attachés à chaque processus du système. Chaque module est chargé de fournir au processus auquel il est attaché une liste de processus suspectés d'être actuellement défaillants dans le système. Les informations fournies par ces oracles peuvent être erronées dans certains cas. Ces oracles peuvent commettre deux types d'erreurs :

- ***Erreurs de complétude*** qui concernent la non suspicion de processus défaillants.
- ***Erreurs de précision*** qui concernent la suspicion de processus corrects.

A partir de ces deux conditions, Chandra et Toueg ont défini huit classes de détecteurs de défaillances. Parmi ces classes, la classe *S* est nécessaire est suffisante pour la résolution du problème de consensus dans un environnement asynchrone en présence d'au moins une majorité de processus corrects.

Conclusion

Dans ce chapitre, nous avons abordé des généralités sur les systèmes répartis, qui sont modélisés généralement par des graphes de processus communiquant entre eux par des canaux de communication. Nous avons présenté les modèles de fautes qu'un processus peut subir. Parmi ces modèles de fautes, on s'intéresse au modèle de fautes byzantines, Nous avons présenté aussi, les types et modèles des systèmes répartis qui existent dans la littérature. Nous nous intéressons dans ce mémoire au système asynchrone. Pour avoir une meilleure gestion de ces systèmes et faire face à une majorité des problèmes qui sont regroupés dans une seule famille appelée problèmes d'accord, plusieurs travaux ont montré qu'il faut se focaliser sur un seul problème fondamental dit le consensus. Un premier résultat a établi à une impossibilité connue sous le résultat FLP [25]. La nécessité d'une solution à ce problème a conduit les chercheurs à contourner l'FLP par plusieurs techniques. Dans ce mémoire, la technique retenue est la celle des oracles ou détecteurs de défaillances [16].

CHAPITRE II

DETECTEUR DE DEFAILLANCES : NOTIONS ET IMPLEMENTATIONS

Introduction

Il est très connu que l'accord est un point essentiel dans le calcul réparti, mais il est très difficile de l'avoir dans un environnement sujet à des défaillances. D'une autre façon, sans aucune hypothèse de synchronie (c à d : des bornes sur la vitesse relative des processus et les délais de communication).

L'accord entre un ensemble de processus répartis, qui communiquent via des canaux de communication fiables où au moins un processus peut subir une faute, est impossible [25]. Ce qui prouve que : les propriétés de synchronie sont nécessaires pour fournir aux processus des informations sur les défaillances. Ces informations sont la clé pour avoir un accord lors de la présence de défaillances.

Plusieurs travaux ont été faits pour déterminer cette information exacte sur les défaillances, avec laquelle l'accord dans un environnement réparti asynchrone peut avoir lieu. Ces travaux ont donné naissance à plusieurs techniques, parmi elles, les oracles distribués, connus sous le nome de détecteurs de défaillances. Depuis la première version de la publication de Chandra et Touag, sous le titre « unriable fialure detectors for reliable distributed systems » en 1991, le concept de détecteur de défaillance a été étudié et investi.

Dans ce chapitre, nous allons introduire la technique de détecteur de défaillances et ces concepts pour se familiariser et comprendre son fonctionnement. Ses principes de bases, ses capacités et ses limitations, ainsi quelques implémentations de cette technique seront également introduites.

2.2 Les Détecteurs de défaillances : définitions et concepts

Dans cette section nous introduisons la notion de détecteur de défaillances initialement proposé par Chandra et Touage [16]. Nous définissons les différentes classes de détecteur qui seront utilisées par la suite. Puis nous détaillons le fonctionnement d'un protocole de consensus basé sur une classe de détecteur particulier. Nous introduisons l'extension à apporter au modèle décrit précédemment dans le chapitre 1 (1.4.3), afin de prendre en compte la notion de détecteur de défaillances. Cette section illustre l'utilisation d'oracles afin de contourner le résultat d'impossibilité FLP [25].

2.2.1 Notion de détecteur de défaillances

Dans le but de circonvenir au résultat d'impossibilité attaché à l'un des problèmes d'accord qui est le consensus, la notion de détecteur de défaillances a été proposée par Chandra et Touag. En effet, dans un système réparti asynchrone, l'impossibilité de distinguer de façon fiable un processus lent toujours actif d'un processus en panne, nécessite l'enrichissement d'un tel système par de nouvelles hypothèses de synchronie, d'où l'idée d'augmenter le modèle par l'introduction de détecteur de défaillances. Cependant, l'originalité de la démarche réside principalement dans deux points :

1. Les auteurs ont choisi de définir ceux-ci comme des boites noires dont l'implémentation reste non spécifiée, et dont les propriétés fonctionnelles sont clairement identifiées ;

2. Ces propriétés se divisent en deux sous-classes et permettent d'envisager des classes de détecteurs très fonctionnellement fiables

Plus précisément, on définit un détecteur de défaillances comme un ensemble de modules attachés à chaque processus du système, chaque module est chargé de fournir au processus auquel il est attaché des informations (possiblement incorrectes) sur les processus actuellement en panne dans le système.

On suppose que les modules sont indépendants dans le sens où les informations fournies par le module aux processus dont il dépend, ne sont pas nécessairement identiques à celles fournies par les autres modules. Par ailleurs, on considère que les informations fournies par les détecteurs n'ont pas obligatoirement une valeur irrévocable. Ceci signifie qu'un processus suspecté par un autre processus à un instant donné, peut être retiré plus tard de la liste des

processus suspectés. Un module de détecteur de défaillances peut faire des erreurs par la suspicion des processus corrects ou par la non suspicion des processus crashés, c'est à dire s'il estime plus tard qu'un processus correct a été suspecté par erreur, il peut l'enlever de sa liste de processus suspectés. Ainsi, chaque module peut ajouter ou enlever à maintes reprises des processus de sa liste des processus suspectés.

Pour caractériser un détecteur de défaillances, les auteurs de [16] ont défini deux propriétés :

1. **La propriété de complétude (Completeness) :** qui vise à réduire le nombre de défauts de suspicion

2. **La propriété de précision (Accuracy)**: qui vise à réduire le nombre de fausses suspicions

Ensuite Chandra et Touag ont divisé ces deux propriétés pour obtenir deux propriétés de complétude et quatre propriétés de précision, ce qui donne huit classes du détecteur de défaillances.

Propriétés de Complétude : Les deux propriétés de complétude sont les suivantes :

- **Complétude Forte :** tous les processus défaillants finiront par être suspectés définitivement par tous les processus corrects. Formellement un détecteur de défaillance D satisfait la propriété de complétude forte si :

$$\forall F, \forall H \in D(F), \exists t \in T, \forall p \in crashed(F), \forall q \in correct(F), \forall t' \geq t : p \in H(q, t').$$

- **Complétude faible :** tous les processus défaillants finiront par être suspectés par au moins un processus correct. Formellement un détecteur de défaillances D satisfait la complétude faible si :

$$\forall F, \forall H \in D(F), \exists t \in T, \forall p \in crashed(F), \exists q \in correct(F), \forall t' \geq t : p \in H(q, t').$$

Tel que

F : dénote une fonction de T sur 2^{Π}, où $F(t)$ dénote l'ensemble des processus sujets à des défaillances à l'instant t ;

H : dénote l'horloge d'un détecteur de défaillances, elle présente une fonction de $\tilde{O} \; X \; T$ sur $2^{\tilde{O}}$, où $H(p,t)$ est la valeur de module de détecteur de défaillances du processus p à l'instant t.

T : dénote le range d'une horloge globale discrète, elle prend des valeurs naturelles.

Propriétés de Précision : Les quatre propriétés de précision sont considérées comme suite :

- **Précision forte :** aucun processus correct n'est suspecté. Formellement, D satisfait la précision forte si :

  $\forall F, \forall H \in D(F), \forall t \in T, \forall p, q \in \Pi - F(t) : p \notin H(q, t)$.

- **Précision faible :** il existe un processus correct qui n'est jamais suspecté. Formellement, D satisfait la précision faible si :

  $\forall F, \forall H \in D(F), \exists p \in correct(F), \forall t \in T, \forall q \in \Pi - F(t) : p \notin H(q, t)$.

- **Précision inéluctablement forte :** il existe un instant *t* à partir duquel aucun processus correct n'est suspecté. Formellement, *D* satisfait la Précision inéluctablement forte si

  $\forall F, \forall H \in D(F), \exists t \in T, \forall t' \geq t, \forall p, q \in correct(F) : p \notin H(q, t')$.

- **Précision inéluctablement faible :** il existe un instant *t* à partir duquel au moins un processus correcte n'est jamais suspecté. Formellement, *D* satisfait la précision inéluctablement faible si :

  $\forall F, \forall H \in D(F), \exists t \in T, \exists p \in correct(F), \forall t' \geq t, \forall q \in correct(F) : p \notin H(q, t')$.

A partir de ces propriétés, Chandra et Touage [16] ont défini huit calasses de détecteurs de défaillances. Ces classes sont résumées dans le Tableau 1.

Propriétés de complétude	Propriétés de précision			
	Précisio n forte	Précision faible	Inéluctablement fort	Inéluctablement faible
Complétude forte	*P*	*S*	*?P*	*?S*
Complétude faible	*Q*	*W*	*? Q*	*?W*

Tableau 1 : Classes de détecteurs de défaillances

2.2.2 But d'un détecteur de défaillances

Le but est de fournir une abstraction des contraintes de synchronisation, c'est à dire réaliser un algorithme qui résout le consensus. On peut s'appuyer sur les informations fournies par le détecteur de défaillances à condition de prendre en compte ses caractéristiques sans se soucier des contraintes temporelles des systèmes temps réels. Cela permet d'améliorer la modularité et la probabilité des algorithmes, et surtout quand cela permet de définir exactement quelles sont les informations nécessaires pour résoudre le consensus.

Il est important de retenir que les détecteurs de défaillances sont une abstraction des problèmes temporels, ils ne permettent pas de résoudre le problème de consensus dans un système purement asynchrone. Ils permettent de séparer le problème de consensus en deux parties. «Comment élaborer un algorithme et quelles informations sont nécessaires pour y parvenir ». Cette partie s'appuie sur les propriétés du détecteur de défaillances. Une deuxième partie consiste à déterminer les détecteurs de défaillances réalisables sur le système distribué en fonction des propriétés de synchronisme du système.

2.2.3 Résolution

Un algorithme *A* résout le problème *B* en utilisant le détecteur de défaillances *D* si toutes les exécutions de *A* utilisant *D* satisfont la spécification du problème *B*. Nous disons que *D* résout le problème *B* s'il existe un algorithme qui résout *B* en utilisant *D* [21].
Soit *C* une classe de détecteurs de défaillances, on dit qu'un algorithme *A* résout le problème *B* en utilisant la classe de détecteurs de défaillances *C*, si pour $D \hat{I} C$, *A* résout *P* en utilisant *D*.

Finalement, on dira qu'*un problème P a été résolu en utilisant une classe de détecteurs de défaillances C, s'il existe un algorithme A résolvant P en utilisant cette même classe.*

2.2.4 Utilisation de la classe du détecteur ?S

Dans cette section, nous présentons le protocole de Chandra et Toueg qui permet de résoudre le problème du consensus lorsque l'on suppose d'une part que l'on dispose d'un détecteur de la classe *?S*, et d'autre part qu'une majorité des processus est correct. Contrairement à l'algorithme non-déterministe de Ben-Or (sec 1.6.4.1), celui-ci permet de résoudre le problème du consensus dans son acceptation la plus large (c'est-à-dire que l'ensemble des valeurs initiales n'est pas limité dans sa taille, et il n'y a pas de connaissances préalables à avoir sur cet ensemble).

Cet algorithme est présenté dans (Algorithme 1). Il se compose de deux tâches qui s'exécutent en parallèle. La première tâche constitue le cœur de l'algorithme, c'est-à-dire aboutir à une décision unanime et irrévocable. Cependant la seconde tâche a pour but de disséminer de manière fiable et cohérente la décision prise par un processus.

L'algorithme exécuté par la première tâche repose sur le paradigme du coordinateur rotatif. On entend par là que l'algorithme procède par ronde durant lesquelles un processus particulier joue le rôle de coordinateur. Chaque ronde est divisée en quatre phases. La deuxième et la quatrième phase ne sont effectuées que par le coordinateur du ronde, cependant la première et la troisièmes phase sont effectuées par l'ensemble des processus (coordinateur compris). Durant la première phase, chaque processus diffuse la valeur de son estimation de la valeur finale. Cette valeur est susceptible d'être mise à jour durant une ronde, de la même manière que dans l'algorithme non-déterministe de Ben-Or. Une caractéristique fondamentale de cet algorithme provient du fait que la valeur transportée par ce vote est estampillée au moyen d'un entier. On verra que cette estampille sert à assurer l'unicité de la valeur décidée.

Lors de la seconde phase, le coordinateur attend une majorité de vote qu'il est garantit de recevoir, puisque les canaux de communication sont supposés fiables, et qu'une majorité de processus sont corrects. Il extrait de cette majorité de messages, celui dont l'estampille est la plus grande. Dans le cas où plusieurs votes présentent l'estampille la plus grande, on considère deux cas :

- soit les votes transportent tous la même valeur, si l'estampille qui leurs est associée est supérieure à 0
- soit tous les votes sont estampillés par l'estampille initiale 0, et le coordinateur est habilité à choisir un vote de manière aléatoire parmi ceux reçus

Cette estimation de la valeur finale est diffusée par le coordinateur.

Les processus attendent de recevoir le précédent message du coordinateur lors de la troisième phase, ou bien de le suspecter. Du fait de la propriété de complétude du détecteur *?S*, on est sûr que le message sera finalement reçu, soit que le coordinateur, s'il est défaillant sera suspecté. Cette attente est donc finie. Un processus qui reçoit le message du coordinateur adopte comme estimation celle envoyée par le coordinateur et met à jour son estampille en la

positionnant à la valeur de la ronde courant, puis il vote positivement pour la dernière phase de la ronde. Dans le cas contraire, il vote négativement.

Seul le coordinateur effectue la dernière phase, en collectant une majorité de votes (positifs, et négatifs) envoyés lors de la phase précédente. Cette phase est elle-même non bloquante pour les mêmes raisons évoquées dans la première phase. Dans l'hypothèse où il reçoit une majorité de votes positifs, il décide la valeur qu'il avait transmise aux autres processus lors de la seconde phase. Les processus n'ayant pas pu décider lors de cette ronde, entament la ronde suivante. Au contraire, un processus qui peut décider durant une ronde, termine la première tâche de l'algorithme pour démarrer la seconde. Cette tâche implémente un algorithme de diffusion fiable.

Il requière l'implémentation de deux primitives **Diffusion_Fiable** et la **Reception_Diffusion_Fiable** qui doivent vérifier les propriétés suivantes :

- ***Validité :*** Un message diffusé de manière fiable par un processus correct est finalement délivré par ce processus.
- ***Accord :*** Si un processus (correct ou non) délivre un message *m*, alors tous les processus corrects délivreront inéluctablement ce message.
- ***Intégrité :*** Tout message reçu par la primitive **Reception_Diffusion_Fiable** doit avoir été émis par un processus via la primitive **Diffusion_Fiable**.

Ces propriétés impliquent que les messages diffusés de manière fiable par les processus corrects sont finalement délivrés, cependant que les messages diffusés de manière fiable par les processus potentiellement défaillants sont soit reçus par tous les processus corrects, soit par aucun. Pour obtenir une implémentation correcte de ces propriétés dans un système purement asynchrone, il suffit d'implémenter la primitive **Diffusion_Fiable** comme une simple diffusion. La primitive **Reception_Diffusion_Fiable** doit quant à elle être implémentée de la manière suivante : Tout message reçu est d'abord rediffusé, avant d'être délivré. C'est exactement ce que réalise la seconde tâche. Elle assure par là que les décisions sont propagées de manière fiable à l'ensemble des processus.

L'algorithme repose d'une manière cruciale à la fois sur le mécanisme d'estampille de manière à assurer l'unicité de la valeur décidée, mais aussi sur la propriété de précision faible des détecteurs de défaillances de la classe *?S* afin d'assurer la convergence de l'algorithme, en permettant durant un tour que, son coordinateur (correct) ne soit suspecté par aucun autre des processus corrects qui forment une majorité suffisante pour assurer la convergence de l'algorithme durant ce tour.

Tâche 1

(1) **fonction** propose(v_i)
(2) est_i ? v_i
(3) r_i ? 0
(4) ts_i? 0
(5) $decide_i$? *faux*
(6) **tant que**($\neg decide_i$)
(7) r_i ? $r_i + 1$
(8) c_i ? ($c_i + 1$ mod n)
(9) **Phase** 1
(10) **envoie**(i, r_i, est_i, ts_i) à c_i
(11) **Phase** 2
(12) **si**($c_i = i$) **alors**
(13) **attendre** une majorité de messages (j, r_i, est_j , ts_j)
(14) sg_i ? {(j, r_i, est_j , ts_j) reçus} (15) ts ? le plus grand ts_j tel que $\exists$ (j, r_i, est_j , ts_j)$\in$ msg_j
(16) est_j ? un des est_j tel que (j, r_i, est_j , ts) $\in msg_j$
(17) **diffuse**(i, r_i, est_i)
(18) **fin si**
(19) **Phase** 3
(20) **attendre** un message (c_i, r_{ci} , est_{ci}) de c_i ou $c_i \in D_i$
(21) **si** un message (c_i, r_{ci} , est_{ci}) de ci à été reçu **alors**
(22) est_i ? est_{ci}
(23) ts_i ? r_{ci}
(24) **envoie**(i, r_i, *ack*) à c_i
(25) **sinon**
(26) **envoie**(i, r_i, *nack*) à c_i
(27) **fin si**
(28) **Phase** 4
(29) **si**($ci = i$) **alors**
(30) **attendre** une majorité de messages (j, r_i, ack_j) ou (j, r_i, $nack_j$)
(31) **si** une majorité de messages (j, r_i, ack_j) à été reçus **alors**
(32) **diffuse**(i, est_i, *decision*)
(33) **fin si**
(34) **fin si**
(35) **fin tant que**

Tâche 2

(36) **attendre** un message (j, est_j , *decision*)
(37) **diffuse**(i, est_j , *decision*)
(38) **si**($\neg decide$) **alors**
(39) decide(est_j)
(40) *decide* ? *vrai*
(41) **fin si**

Algorithme 1: L'algorithme de Chandra-Toueg utilisant un détecteur de défaillances

2.3 Impossibilité d'implémentations des détecteurs de défaillances perpétuels dans un système partiellement synchrone

Dans [40], Larea et al ont démontré que parmi les huit classes de détecteurs de défaillances définies par Chandra et Toueg [16], aucun détecteur ne satisfaisant la précision perpétuelle (*Faible ou Forte)*, nommés *P, Q, S,* et *W,* ne peut être implémenté d'une façon déterministe dans un système partiellement synchrone conforme au modèle M_1 ou M_2 [16] (définis dans I.3.3).

À partir de la relation entre les classes de détecteurs de défaillances décrits dans [16], il doit être suffisant de montrer l'impossibilité pour la classe des détecteurs de défaillances *W,* puisque *W* et la plus faible des quatre classes satisfaisant la précision perpétuelle. Larea et al ont démontré leurs résultats sur les classes satisfaisant la précision forte (*P* et *Q*) ensuite pour les classes satisfaisant la précision faible (*S, W*), dans les deux cas l'approche suivie est de supposer l'existence d'un détecteur de défaillances qui satisfait la propriété de complétude, et de montrer que la propriété de précision perpétuelle est violée.

Le principe utilisé par Larea et al [40] pour prouver le résultat d'impossibilité est de considérer des exécutions différentes du système -avec et sans crashs- telles qu'elles semblent identiques pour certains processus corrects jusqu'à un certain temps *t*. Donc, ces processus peuvent amener les mêmes actions dans les deux types d'exécution jusqu'à *t*, en particulier dans ce qui concerne la suspicion d'autres processus.

Ils ont montré que, en faisant ceci, la précision perpétuelle exigée est violée, et donc le détecteur de défaillances n'implémente aucune des quatre classes perpétuelles des détecteurs de défaillances définies dans [16]. Pour construire une exécution sans aucune défaillance, cela semble identique, jusqu'à l'instant *t*, à une exécution avec une défaillance. On suppose que les messages appropriés sont retardés au-delà de l'instant *t*. Ceci peut se produire si la valeur du paramètre ? ou GST est plus grande que *t*. Cette hypothèse est valide puisque les valeurs de ces paramètres sont inconnues, elle peut être choisie librement pour une exécution donnée.

2.4 Implémentations de détecteurs de défaillances

Après la première version de détecteur de défaillances proposée par Chandra et Toueg, dans leur papier sous le titre : Un détecteur de défaillances non fiable pour un système distribué fiable en 1991, le concept de détecteur de défaillance a été extensivement étudié et investi. Et comme beaucoup de problèmes d'accord peuvent être résolus moyennant les

détecteurs de défaillances, l'implémentation de ces derniers a eu lieu dans plusieurs travaux, [1, 9, 16, 28, 30, 36, 37, 38, 59], qui ont adopté plusieurs techniques et modèles.

Dans cette partie, nous présentons une implémentation de détecteur de défaillances, qui est une implémentation du détecteur Omega (?).

2.4.1 Implémentation de Omega avec un minimum d'hypothèses de synchronie

Omega (O) est l'un des plus intéressants détecteurs de défaillances [17]. Avec O chaque processus à une variable locale $leader_p$ qui contient l'identité d'un seul processus que *p* considère opérationnel (p considère ce processus son leader). Initialement les processus peuvent avoir déférents leaders, mais O garantit l'existence d'un temps après lequel tous les processus ont un même leader *non-fautif*. Ce détecteur de défaillance est important pour des raisons théorique et pratique : il est connu qu'il est le plus faible détecteur de défaillances avec lequel en peut résoudre le problème de consensus [17], et qu'il est le détecteur utilisé par plusieurs algorithmes de consensus, parmi lesquels on trouve des algorithmes qui sont utilisés dans la pratique [32, 33, 39 , 43].

Dans [13], M. K. Aguilera et al, ont étudié le problème d'implémentation de O dans un système avec les plus faibles hypothèses de synchronie et de fiabilité, et plus particulièrement ils se sont intéressés à une implémentation avec une communication efficace[3].

2.4.1.1 Modèle du système

Le système distribué considéré pour ce protocole se compose d'un ensemble de n =2 processus ? = {0, . . ., n - 1}, qui communiquent entre eux par l'envoi de messages via un ensemble de liaisons unidirectionnelles ? . Les comportements des processus et des liaisons sont décrits en détail comme suit :

Processus :

Les processus s'exécutent par étapes. Dans une étape, un processus peut recevoir un ensemble de messages, ensuite il change son état, comme il peut envoyer des messages ensuite changer d'état. La valeur d'une variable d'un processus au temps *t* est la valeur de cette variable après que le processus prend une étape au temps *t*. Pour chaque processus non fautif, la fréquence d'exécution (nombre d'étapes par temps d'exécution) a une borne

[3] Implémentation efficace : une implémentation dans laquelle il existe un temps après lequel seulement un seul processus (le leader élu) envoi les messages

maximale et minimale. Les processus ont une horloge qui n'est pas nécessairement synchronisée, mais les processus peuvent calculer avec précision les intervalles de temps.
Un processus peut tomber en panne par le crash définitif, pendant lequel il ne prend aucune étape. Nous disons que le processus *p* est en vie au temps *t* s'il n'est pas crashé au temps *t*, et que le processus est correct s'il est toujours en vie. Un processus est fautif s'il n'est pas correct.
M. K. Aguilera, C. Delporte-Gallet, H. Fauconnier, et S. Toueg considères dans leur modèle du système qu'un nombre quelconque de processus peut tombés en panne.

Liaisons : le réseau est assumé d'être fortement connecté, c'est-à-dire *? = ? × ?* . La liaison unidirectionnelle du processus *p* au processus *q* est dénotée par *p ? q.* Plusieurs types de liaisons sont considérés, chacun satisfait la propriété suivante :

- ***Intégrité :*** Le processus *q* reçoit un message *m* au plus une fois du processus *p*, si et seulement si *p* a envoyé *m* précédemment.

Une liaison *p ? q* est inéluctablement synchrone (*eventually timely*) si elle satisfait la propriété d'intégrité et la propriété suivante :

- ***Inéluctablement synchrone : (****Eventual timeliness)* il existe un *d* et un temps *t* tel que : si *p* envoie un message *m* à un processus correct *q* au temps *t' = t*, alors *q* reçoit *m* de *p* au temps t'+ *d.*

Le délai maximum du message *d* associé à chaque liaison est non connu.
Une liaison qui perd des messages par intermittence satisfait la propriété d'équité (*fairness).* Pour définir cette propriété, nous supposons que chaque message porte un type et des donnes. L'équité (*fairness)* nécessite que si un processus envoie un nombre infini d'un type de messages sur une liaison, alors la liaison doit délivrer le même nombre de ce type de messages. Plus précisément, nous supposons qu'un message est constitué de la paire *m* = (*Type*, *Data*) *?* $S^* \times S^*$ avec S = *{*0, 1*}*. Une liaison *p ? q* est équitable *(fair)* si elle satisfait la propriété suivante :

- ***Equité*** pour chaque ***type*** : si ***p*** envoie plusieurs messages du type ***type*** *au processus correct* ***q*** *alors* ***q*** reçoit infiniment plusieurs messages du type ***type*** de processus ***p***.

Les processus corrects sont classés sur la base de leur liaison en :

- ***Une source inéluctablement synchrone*** (*eventually timely source)* dénotée ***?-source*** ; est un processus correct dont toutes les liaisons sortantes (seulement les sorties, d'où vient la désignation ***source***) sont inéluctablement synchrones.

- ***Hub équité (fair hub) :*** c'est un processus correct dont les liaisons sortantes et entrantes satisfont la propriété d'équité.

Systèmes : trois types de systèmes sont considérés, S^-, S et S^+, comme il est montré dans le tableau 2, qui diffèrent par les propriétés de leurs liaisons (les propriétés de leurs processus sont décrites dans le début de cette section). Dans les trois systèmes, toutes les liaisons satisfont la propriété d'intégrité.

Le système S^- n'a pas d'autres propriétés, en outre, toutes les liaisons sont asynchrones et/ou peuvent perdre des messages.

Dans le système S, il est supposé qu'il existe au moins une source *inéluctablement synchrone* ***?-source*** (son identité est inconnue). A l'exception des sorties de la liaison ***?-source***, toutes les autres liaisons peuvent être asynchrones et/ou perdre des messages, et la majorité des paires de processus peuvent communiquer dans le système S.

S^+ est un système avec au moins une source *inéluctablement synchrone* (*eventually timely source*) et au moins un *hub d'équité* (*fair hub*) dont leurs identités sont inconnues. Notons que dans S^+ la majorité des paires de processus ne peuvent communiquer dans un temps fini, bien que les liaisons équités de Hub(s) inconnu(s) sont toujours asynchrones et/ou peuvent perdre des messages.

Système	***Propriétés***
S^-	***Liaison*** : Asynchrones et Peuvent perdre des messages ***Processus :*** synchronisés et peuvent subir des crashs
S	S^- avec au moins une ***?-source***
S^+	S avec au moins un *Hub équité (fair Hub)*

Tableau 2 : Système considéré dans l'implémentation [13]

2.4.1.2 Le détecteur O

Une définition informelle est donnée dans [16, 17]. Informellement, O fait sortir pour chaque processus *p*, un seul processus dénoté $leader_p$ tel que la propriété suivante est vérifiée :

- Il existe un seul processus *l* et un temps après lequel pour chaque processus en vie (n'est pas en panne), $leader_p = l$.

Si au temps *t*, $leader_p$ contient le même processus pour tout processus *p*, alors ont dit que «*il est le leader au temps t* ». Notons qu'à n'importe quel moment, les processus ne savent pas qu'il existe un leader, mais ils savent qu'un leader va être élu.

2.4.1.3 Algorithme d'implémentation du détecteur O dans S

Cet algorithme assure que les processus à la fin vont se mettre d'accord sur un leader commun, bien que la majorité des pairs de processus peuvent communiquer l'un avec l'autre (rappelons que dans *S* toutes les liaisons peuvent être asynchrones et perdre des messages, sauf les sorties de quelques processus corrects inconnus).
L'idée de base est que chaque processus choisit son leader parmi les processus qui semblent en vie. Mais puisque presque toutes les liaisons dans *S* vont souffrir de l'asynchronisme et/ou de la perte de messages, plusieurs problèmes peuvent se produire. En particulier :

1. Différents processus vont avoir déférentes vues (*views*) des processus qui sont actuellement en vie, et ces déférentes vues ne convergent jamais.

2. Quelques processus vont balancer entre apparaître en vie et en panne et continuant à faire ceci.

Ces problèmes vont compliqués la tâche de sélection d'un leader permanant et commun. Par exemple, le processus *p* ne peut pas sélectionner comme leader le plus petit processus qui semble actuellement en vie : le problème (1) va causer que différents processus vont avoir différents leaders (d'une façon définitive), et le problème (2) va obliger *p* a choisir son leader d'une façon définitive.
Pour surmonter ces difficultés, les processus maintenant des compteurs dits "d'accusation", utilisent d'une manière indirecte les *?-source(s)* inconnus pour les aider à converger au même processus correct qui va être le leader. L'algorithme décrit dans l'Algorithme 2, fonctionne comme suit :

Chaque processus *p* maintient un ensemble *SET2* de processus considérés actuellement en vie : C'est l'ensemble des candidats pour être le leader de *p*. Pour choisir un leader parmi les différents candidats, *p* maintient un compteur (*counter)* [*q*] pour chaque processus *q*, avec lequel *p* garde une estimation de combien de temps *q* a été suspecté précédemment d'être en panne. Le processus *p* choisit comme leader le processus l parmi l'ensemble *SET2* qui a la plus petite paire (*counter*[*l*], *l*). (Le leader est calculé chaque fois que l'ensemble *SET2* ou le compteur est mis à jour).

Pour aider les processus à maintenir leurs ensembles de variables compteurs, chaque processus *q* envoie un message périodiquement de la forme *(Alive, q, counter[q])*, pour chaque période *?*, à tous les processus (notons que la plupart de ces messages vont être perdus ou retardés).

Si un processus *p* reçoit *(Alive, q, counter[q])* directement de *q, p* envoie le message une fois aux autres processus, et réinitialise le compteur de temps local dénoté *timer1* (*q*) pour qu'il expire avant un *'Timeout1* [*q*]' unité de temps, et c'est le délai maximum durant lequel *p* reçoit le prochain message (Alive, *q, counter* [*q*]) *directement* de *q*.

Si *timer1* (*q*) expire avant la réception de ce message directement de *q*, alors *p* envoi un message d'accusation à *q*. Quand *q* reçoit un message d'accusation, il incrémente son *counter*[*q*].

Si *p* reçoit (Alive, *q, counter* [*q*]) d'une façon directe de *q* ou retransmis par un autre processus, alors *p* ajoute *q* à son ensemble *SET2*, il met à jour par conséquence sa variable *counter* [*q*], et réinitialise le *timer2*(*q*), pour qu'il calcule le délai de la réception du prochain message (Alive, *q, counter* [*q*]) (directement ou retransmis). Si le *timer2*(*q*) expire avant la réception de ce message, alors *p* supprime *q* de l'ensemble *SET2*.

M. K. Aguilera et al, ont amélioré la solution précédente dans le système *S* pour quelle soit une implémentation efficace dans le même système, une implémentation qui minimise l'usage des liaisons de communication.

Ils ont proposé ensuite un algorithme pour O qui nécessite un seul processus pour envoyer les messages dans le système *S* avec toutes les liaisons équitables (*fair*), par la suite, ils ont proposé une autre implémentation efficace pour le système S^+ [13].

Code pour chaque processus
procedure *updateLeader*()
1 *leader ?* l such that (*counter*[l], l) = min{(*counter*[*q*], *q*) : *q ? set2}*

On initialization:
2 *? q* ≠ *p* : *Timeout1*[*q*] *?* *?* + 1
3 *? q* ≠ *p* : *Timeout2*[*q*] *?* *?* + 1
4 *? q* ≠ *p* : reset *timer1*(*q*) to *Timeout1*[*q*]
5 *? q* ≠ *p* : reset *timer2*(*q*) to *Timeout2*[*q*]
6 *? q* : *counter*[*q*] *?* 0
7 *set1 ? {p}*; *set2 ? {p}*
8 **start Tasks** 1 and 2

Task 1:
9 **loop forever**
10 send (Alive, *p*, *counter*[*p*]) to every process except *p* every *?* time

Task 2:
11 **upon** receive (Alive, *q*, *c*) from *q'* **do**
12 **if** *q* = *q'* **then**
13 reset *timer1*(*q*) to *Timeout1*[*q*]
14 *set1 ? set1* ∪ *{q}*
15 send (Alive, *q*, *c*) to every process except *p* and *q*
16 reset *timer2*(*q*) to *Timeout2*[*q*]
17 *set2 ? set2* ∪ *{q}*
18 *counter*[*q*] *?* max{*counter*[*q*], *c}*
19 *updateLeader()*
20 **upon** expiration of *timer1*(*q*) **do**
21 send Accusation to *q*
22 *set1 ? set1* - *{q}*
23 *Timeout1*[*q*] *? Timeout1*[*q*] + 1
24 reset *timer1*(*q*) to *Timeout1*[*q*]
25 **upon** expiration of *timer2*(*q*) **do**
26 *set2 ? set2* - *{q}*
27 *Timeout2*[*q*] *? Timeout2*[*q*] + 1
28 reset *timer2*(*q*) to *Timeout2*[*q*]
29 *updateLeader*()
30 **upon** receive Accusation **do**
31 *counter*[*p*] *? counter*[*p*] + 1

Algorithme 2 : Implémentation d' O dans le système S

Conclusion

Dans ce chapitre, nous avons présenté qu'est ce qu'un détecteur de défaillance, sa définition ses notions et concepts, nous avons également présenté ses classes et les propriétés de chaque classe, ainsi que la façon d'utilisation de ces détecteur de défaillances par un algorithme. Nous avons aussi, introduit une implémentation de détecteurs de défaillances pour mieux expliquer l'utilité, la manière d'utilisation et de fonctionnement de ces détecteurs de défaillances.

CHAPITRE III

DETECTEURS DE DEFAILLANCES ET FAUTES BYZANTINES

Introduction

Dans le chapitre précédent nous avons vu la notion de détecteur de défaillances, comment elle est introduite par Chandra et Toueg [16]. Ils avaient un objectif de fournir une abstraction des problèmes de synchronisation afin de permettre la réalisation d'algorithme pour les systèmes répartis.

Dans [17], Chandra et al ont démontré que le problème de consensus peut être résolu par un détecteur de défaillances de la classe *?S* si au moins la majorité des processus demeuraient corrects.

L'approche de détecteur de défaillances a été plus étudiée et étendue dans plusieurs travaux. Dans la plus parts de ces travaux [2, 9, 30, 34, 37, 39, 61], cette approche a été implémenter pour résoudre certain problèmes tel que le consensus ou l'élection d'un leader.

Dans un environnement où des défaillances byzantines peuvent manifester, l'implémentation de cette approche est une tache complexe et très difficile a réalisé, et il n'existe que deux implémentation qui ont été proposé pour résoudre le problème de la détection des défaillances byzantines.

Dans ce chapitre nous présentons ces deux implémentations. En premier nous présentons le protocole de Kihlstrom et al [36]. Ensuite, celui de Doudou et al [23]. Enfin, nous comparons entre ces deux protocoles.

3.2 Le protocole de Kihlstrome et al

3.2.1 Modèle du système

Le modèle du système distribués considéré pour ce protocole se compose de n ≥2 processus ; chaque processus a un identifiant unique. Le système est asynchrone, c'est à dire il n'existe aucune bornes sur les vitesses relatives de processus et les délais de transmission de messages. Chaque processus, a un accès à un son horloge locale, mais les horloges des différents processus ne sont pas synchronisées. Le réseau de communication est complètement connecté et les canaux de communication sont fiables. Un processus peut être correct ou défaillant. Un processus correct se comporte toujours selon sa spécification. Un processus défaillant exhibe un comportement arbitraire (byzantin). Un processus byzantin peut crasher, arrêter l'envoie de certains messages, envoyer des messages différents à des processus différents et etc. l'entier k dénote le nombre maximum des processus byzantins tel que $k = [(n-1)/3]$.

Kihlstrom et al [36] ont supposé qu'un mécanisme d'authentification est disponible, pour qu'un processus puisse vérifier l'émetteur original du message, même si un autre processus a relayé ce message. Ce modèle utilise une clé publique d'un crypto-système tel que RSA [58] dans lequel chaque processus p possède une clé privé connue seulement par lui-même et avec laquelle il peut signer numériquement ses messages. Chaque processus est capable d'obtenir les clés publiques des autres processus pour vérifier l'identité de l'émetteur d'un message. La signature est supposée non forgeable et un moyen de distribution de clés est disponible, c'est à dire tous les processus ont la même clé publique pour chacun des processus. Les hypothèses de synchronie ajoutée au système sont celles du modèle partiellement synchrone M_3 de Chandra et Toueg.

- Les défaillances byzantines : Kihlstrom et al [36] ont défini les défaillances byzantines en terme de déviations de l'algorithme A, que les processus exécutent. Cependant, nous ne pouvons pas détecter les défaillances qui sont commises dans l'état interne d'un processus byzantin, puisque le comportement externe d'un processus byzantin peut être inconsistant avec son état interne. Par exemple, un processus byzantin qui exécute l'algorithme de consensus peut rapporter sa valeur d'entrée, ou peut décider une valeur autre que celle du consensus. Les défaillances Byzantines peuvent être classifiées en deux catégories : *détectable* et *non détectable*. La figure 3 décrit La classification de défaillances byzantines définies par Kihlstrom et al [36].

Les défaillances *non détectables* sont :

(1). Défaillances qui ne sont pas observables (*unobservable*) par un processus en se basant sur les messages qu'il reçoit.

(2). Défaillances qui ne sont pas diagnostiquées (*undiagnosable),* c'est-à-dire ne peuvent être attribuées à un processus particulier.

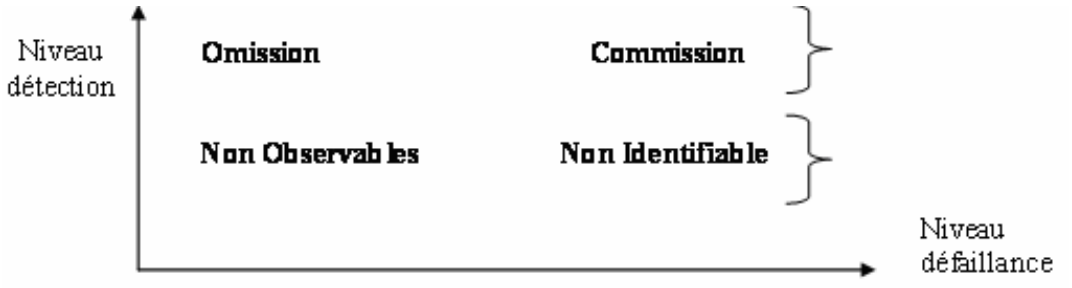

Figure 3 : Classification des défaillances byzantines

Par exemple : si un processus byzantin *p* envoie à tous les processus un message qui porte une valeur *v* et stocke dans ses variables locales une autre valeur *v'*, ce comportement est non observable. Ainsi, si un processus byzantin *p* renvoie un message en imitant un autre processus *q*, avec une signature non valide, alors ce type de défaillances ne peut pas être diagnostiqué.

La détection des défaillances byzantines se réfère aux déviations qui apparaissent dans le comportement externe d'un processus. La déviation détectable sur un algorithme est définie en terme de messages envoyés (ou non envoyés) par les processus durant une exécution particulière de l'algorithme.

Les déviations détectables sur un algorithme *A* sont classifiées en deux type : défaillances de commission et défaillances d'omission. Une défaillance d'omission se produit lorsque un processus n'envoie pas certains messages qui doivent être envoyés dans une exécution *E* de l'algorithme *A*. Une défaillance de commission se produit quand un processus envoie un message qui ne doit pas être envoyer durant une exécution *E* de l'algorithme *A*, c'est à dire un processus byzantin peut envoyer un message, avec la même en-tête et avec des contenus différents, à des processus différent.

3.2.2 Le détecteur de défaillance byzantin non fiable

Kihlstrom et al [36] ont défini une nouvelle propriété de complétude pour le détecteur de défaillances byzantines utilisé par l'algorithme *A*, comme suit :

Complétude byzantine éventuelle forte pour l'algorithme A : il existe un temps après lequel, chaque processus dévie sur l'algorithme *A* sera suspecté indéfiniment, par tous les processus corrects.

Le reste de défaillances seront laissée à l'algorithme *A* pour les détectées. Le but ce protocole est de détecter le maximum de défaillances possibles, (les défaillances d'omission et de commission) expliquées précédemment.

En plus, à la propriété de complétude, la propriété de précision inéluctable définie par Chandra et Toueg [25] aussi est utilisée. Avec ces deux propriétés Kihlstrom et al ont défini deux classes de détecteur de défaillances comme suit :

1. Un détecteur de défaillance pour la classe *?P(Byz,A)* qui satisfait la complétude byzantine inéluctablement forte pour l'algorithme *A*, et la précision inéluctable forte.

2. Un détecteur de défaillances pour la classe *?S(Byz,A)* qui satisfait la complétude byzantine inéluctablement forte pour l'algorithme *A*, et la précision inéluctable faible .

Format des messages

Dans [36], kihlstrome et al ont défini le format de message utilisé dans leur implémentation. Chaque message possède la forme *<m.hdr, d(m.cont), d(m.jst), m.sig, m.cont, m.jst>* , figure 4. Un processus qui reçoit un message doit vérifier sa signature. Si la signature du message est non valide, le processus détruit le message. Un message signé doit être proprement formé, et proprement justifié. Si le message n'est pas bien formé ou non justifié, le processus n'accepte pas d'utiliser ce message dans l'algorithme de consensus.
Les deux parties du message (Statement et le message entier) sont signées, ce qui permet l'extraction de la partie Statement du message, pour l'utiliser dans le champ de justification du message ultérieur sans causer une augmentation dans de la tailler de message

Pour envoyer un message *m* composé d'une entête du message (*m.hdr),* le contenu (*m.cont)* et la justification *(m.jst),* le processus *p* calcule par la fonction digest : d(*m.cont*) et le digest d(*m.jst*). Ensuite il calcule la signature (*m.sig*) selon la combinaison de l'entête et les digest (*m.hdr, d(m.cont), d(m.jst)*).

Un processus *q* qui reçoit un message *m* peut le vérifier s'il est valide, et que *d(m.cont)* et *d(m.jst)* sont respectivement les digests du contenu et du justification. Le processus *q* peut

inclure le Statement signé dans la justification dans d'autres messages *m'*, par l'inclusion de <*m.hdr, d(m.cont), d(m.jst), m.sig, m.cont* >dans le champs justification *m'.jst*.

Un processus *r* qui reçoit *m'*, peut vérifier que *p* qui a signé le Statement contenu dans le champ justification, en vérifiant que *d(m.cont)* est le digest de *m.cont* , et que *m.sig* est une signature valide de *p* calculé avec <*m.hdr, d(m.cont), d(m.jst)*>

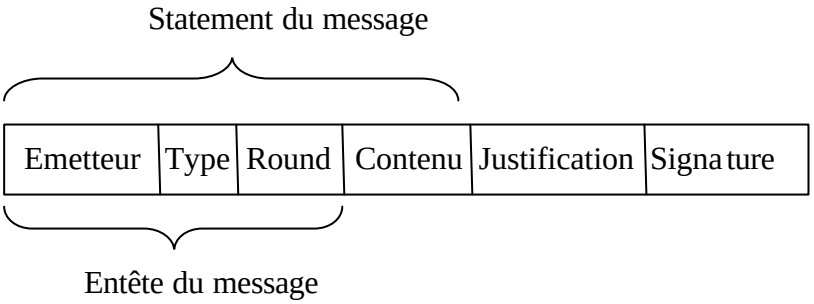

Figure 4 : Format du message [36]

3.2.3 Implémentation du protocole

Dans cette partie, nous présentons l'algorithme d'implémentation d'un détecteur de défaillance $D \in ?S(Byz, A)$ pour un algorithme général *A*, qui utilise le détecteur dans le modèle de système M_3. Le détecteur de défaillances exécuté par un processus *p* (algorithme 3) se compose de trois tâches.

Dans ***la première*** tâche, si un processus *p* attend un message *m*, il fixe un timeout pour ce message. Un processus *p* est dit en attente d'un message exigé *m* ayant une entête particulière quand *m.sender = q*.

Dans ***la deuxième*** tâche, si le timeout expire pour un message attendu *m*, avec *q = m.sender*, *p* ajoute *q* à sa liste $output_p$ des identités des processus suspectés, et il conserve l'information qui concerne l'entête du message attendu dans la liste $expecting_p[q]$.

Dans ***la troisièmes*** tâche, quand le processus *p* reçoit un message *m* signé pour la première fois par *q*, avec *q=m.sender* et *q¹p*, *p re*transmit *m* dans sa forme originale à tous les processus. En suite, *p* vérifie si le message *m* est proprement formé et justifié, ou bien si un message *m'* est retransmis, et il est signé par *q*. Si *m n'est pas proprement formé ou non*

justifié, ou bien il est retransmet, alors *p* ajoute *q* à sa liste $output_p$, et aussi à sa liste $byzest_p$ pour indiquer que *q ne* doit jamais par la suite être supprimé de la liste $output_p$.

```
/* Chaque processus p exécute le code suivant */
/* Initialisation */
output_p ? ∅;                                   /* l'ensemble de processus que p suspecte */
byzset_p ? ∅;                                   /* l'ensemble de processus que p les connus comme byzantines */

Pour claque q Î S
    ?_p(q) ? valeur par défaut ;                /* la Duré de l'intervalle timeout de q */
    expecting_p [q] ? ∅;                        /* messages attendus de q p exigé par p quand le timeout expire */
cobegin                                         /* trois taches concurrents */

/* Tache 1: */
Quand [p attend et exige un message m, avec q = m.sender]
    Posé la duré de timeout ?_p(q) pour le message m;

/* Tache 2: */
Quand [le timeout de p pour un message attendu m, avec q = m.sender]   /* défaillances d'omission*/
    Output_p ? output_p ∪ {q};
    Expecting_p [q] ? expecting_p[q] ∪ {m};

/* Tache 3: */
Quand [p reçoit un message attendu m proprement signé pour la première fois, avec q = m.sender]
Si [m est non proprement formé ou m est non proprement justifié ou m est un message m'
retransmît] Alors                               /* défaillances de commission*/
    Output_p ? output_p È{q};
    Byzset_p ? byzset_p È {q};
Sinon
   Annulé le timeout pour le message m;
Si [m Î expecting p[q]] Alors                    /* une prématuré de timeout a apparu */
    expecting_p[q] ? expecting_p [q] - {m};
    ?_p(q)? ?_p(q) + 1;                         /* incrémenter le timeout pour q */
Si [expecting_p [q] = Æ and q Ï byzset [p] Alors
    Output_p ? output_p - {q};
Si [q _= p] Alors
    Envoyé m à tous les processus ;             /* Retransmit le message m */
coend
```

Algorithme 3 : Une implémentation d'un détecteur *D* Î *?S(Byz, A)*

3.3 Détecteur de défaillances Mutes (Muteness)

Doudou et al [23] ont défini un détecteur de défaillances pour l'environnement byzantin basé sur la notion des processus mutes. Un processus mute est un processus qui arrête d'envoyer des messages ou qui envoie que des messages non signés.

L'implémentation de Doudou et al [23] se base sur une approche similaire a celle utilisé par Malkhi et al [42], dans le sens où leur protocole ne détecte que les défaillances qui empêchent la progression dans les calcules des systèmes, les autres types de défaillances sont laissés au protocole de consensus.

3.3.1 Modèle du système

Ce protocole est implémenté sous un système distribué asynchrone, composé d'un nombre fini $\Omega = \{p_{1....} p_N\}$ de N processus, fortement connecté via des canaux de communication fiable.

Un algorithme distribué A est défini comme un ensemble d'automates distribué A_p qui s'exécutent dans le système par les processus. Dans cette section, nous réfèrons à A_p comme étant un algorithme plus qu'un automate.

Restriction sur le modèle byzantin : Les défaillances byzantines peuvent être séparées en deux groupes, défaillances détectables et indétectables [36].

Les défaillances indétectables sont celles qui ne peuvent pas être détecté si en se base sur les messages reçus, ex : les processus byzantins qui falsifient leur valeur initiale quand ils participent à un protocole d'accord. Face à cette impossibilité de détection d'un tel type de défaillance, un processus qui commit que des défaillances indétectable est considéré comme correct par les autres processus correct.

Les défaillances détectables sont classées en :

(1). ***Défaillances de commission*** : dans ce type de défaillances, les messages reçus ne respectent pas la sémantique de l'algorithme.

(2). ***Défaillances d'omission*** dans ce type de défaillances, les messages attendus par l'algorithme ne seront jamais reçus.

Selon cette classification une défaillance mute est défaillance d'omission permanente, se qui veut dire qu'un processus mute est un processus qui stop arbitrairement l'envoi des messages de l'algorithme à un ou plusieurs processus correcte.

Défaillances mutes

Un processus q est mute par apport à un algorithme A et au processus p si q stoppe inéluctablement l'envoi de certain messages attendus de A à p. Le motif de défaillance mute F est une fonction de $\Omega \times \mathsf{T}$ dans 2^{Ω}, avec $F\{p, t\}$ est l'ensemble des processus qui stoppent l'envoi des messages attendus par l'algorithme A pour le processus p au temps t.

Par définition nous avons $F\{p,t\} \subseteq F\{p,t+1\}$. Nous avons aussi $quit_p(F) = \cup_{t \in \mathsf{T}} F(p,t)$ et correct(F) = Ω - $\cup_{p \in \Omega} quit_p(F)$.

Nous disons que $\boldsymbol{q}$ est mute pour $\boldsymbol{p}$ si $\boldsymbol{q} \in quit_p(F)$.

Si $\boldsymbol{q} \in$ correct(F) alors il est correct. Les défaillances mutes sont un sous ensemble de l'ensemble des défaillances byzantines qui incluent les défaillances par crash. La figure 5 explique cette idée, B qui dénote l'ensemble de toutes les défaillances byzantines, M est l'ensemble des défaillances mutes et C est l'ensemble des défaillances par crash. Un processus crashé est un processus mute mais $M \not\subset C$ par ce qu'un processus mute stoppe l'envoi des messages mais il n'est pas crashé.

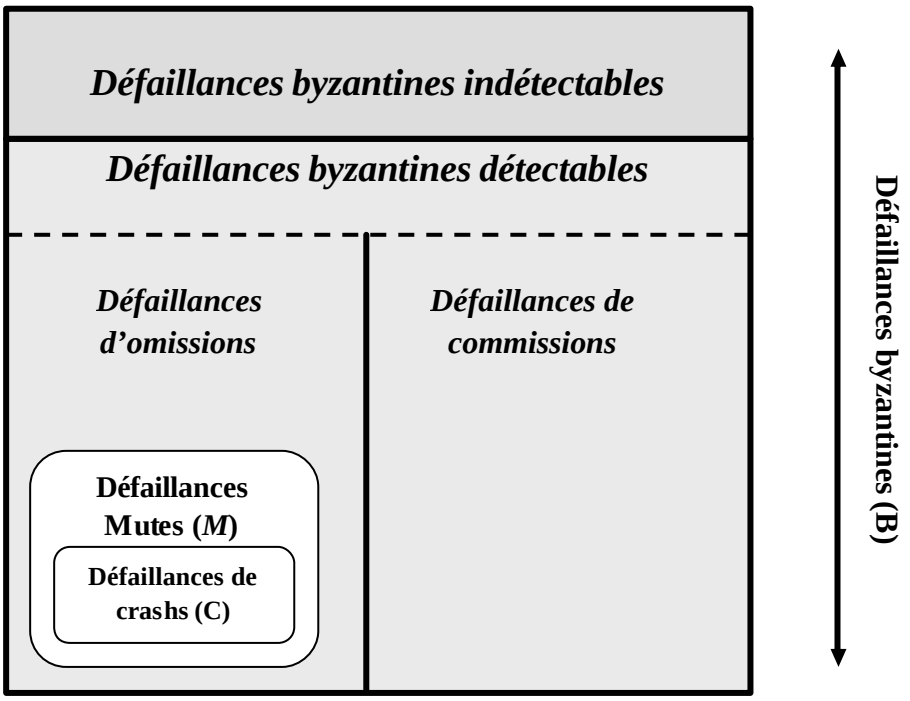

Figure 5 : Classification des défaillances byzantines [23]

3.3.2 Détecteur de défaillances Mutes

Un détecteur de défaillances mutes est un oracle distribué, qui a comme objectif de détecter les processus mutes. Formellement le détecteur de défaillances mutes $?M_A$ est exprimé en terme des propriétés suivantes :

Complétude éventuelle mute il existe un temps après lequel chaque processus mute pour le processus correct *p*, est suspecté par *p* définitivement

Précision éventuelle faible il existe un temps après lequel le processus correcte *p* n'est plus suspecté d'être mute.

3.3.3 Implémentation du protocole

Dans le modèle de défaillances par crash, l'implémentation d'un certain détecteur de défaillances de type *D* peut se faire indépendamment des messages envoyés par l'algorithme qui utilise le détecteur de défaillances *D.* Cette indépendance est réalisable puisque un processus crashé stoppe complètement son exécution. Par contre, dans le modèle de défaillances mutes, les processus peuvent stopper l'envoie de messages de l'algorithme sans être crashés. Donc, la réception périodique de messages par le module de détection de défaillances de certains processus *p* ne garantit pas qu'un message de l'algorithme va être inéluctablement reçu du processus *p*. Par conséquence, une implémentation de $?M_A$ ne peut pas être indépendants de l'algorithme qui l'utilise.

1 : {Chaque processus ***p*** exécute le code suivant}
2 : $?_p$? *init*$_?$; *output*$_p$? Æ ; *critical*$_p$? Æ ; **{Initialisation}**

3 : **Pour tous** $q \in$ *critical*$_p$ **faire** **{Tache 1}**
4 : **Si** (q $\notin$ *output*$_p$) **et** (*p* na pas reçu « q-is-not-mute » pendant $?_p$ temps) **Alors**
5 : *output*$_p$? *output*$_p \cup q$

6 : **Quand recevoir** « q-is-not-mute » **de** A_p **{Tache 2}**
7 : **Si** (q $\notin$ *output*$_p$) **Alors**
8 : *output*$_p$? *output*$_p$ - *q*

9 : **Quand** recevoir nouveau_ *critical*$_p$ **de** A_p **{Tache 3}**
10 : *critical*$_p$? nouveau_ *critical*$_p$
11 : $?_p$? $g_A(?_p)$

Algorithme 4 Une implémentation ID du détecteur de défaillances $?M_A$

Une implémentation $\boldsymbol{I_D}$ du détecteur de défaillances $\boldsymbol{?M_A}$ est donnée dans l'Algorithme 4. Elle est réalisée en utilisant le mécanisme de timeout, et elle se compose de trois tâches concurrentes. La variable $\boldsymbol{?_p}$ indique le timeout courant et elle est initialisée pour tous les processus par une variable arbitraire $\boldsymbol{init_?} > 0$. $\boldsymbol{I_D}$ maintient une liste $\boldsymbol{output_p}$ des processus actuellement suspectés et un ensemble $\boldsymbol{critical_p}$ contient les processus qui peuvent être ajouté à la liste $\boldsymbol{output_p}$ par $\boldsymbol{p}$. Ces deux ensembles sont initialement vides.

Un nouveau processus suspecté est ajouté à la liste $\boldsymbol{output_p}$ par la tâche 1 comme suit :
Si $\boldsymbol{p}$ ne reçoit pas un message «q-is-not-mute » pendant un temps $\boldsymbol{?_p}$ de certain processus $\boldsymbol{q}$ qui est dans $\boldsymbol{critical_p}$, $\boldsymbol{q}$ est suspecté d'être défaillant et il est inséré dans l'ensemble $\boldsymbol{output_p}$.

Interaction entre A_p et $?M_A$

La figure 6 résume l'interaction entre l'algorithme A_p exécuté par un processus correct p et $?M_A$. L'implémentation $\boldsymbol{I_D}$ traite deux interactions entre A_p et $?M_A$. Les tâches 2 et 3 sont responsables pour ces interactions (flèche 1). Tous le temps p reçoit des messages de quelques processus q (flèche 2), l'algorithme A_p délivre « q-is-not-mute » à $\boldsymbol{I_D}$ (flèche 3). Par conséquence, la tâche 2 supprime le processus q de l'ensemble $output_p$ dans le cas où q a été suspecté. Au début de chaque tour, A_p délivre l'ensemble $nouveau\_critical_p$ à'I_D (Flèche 4), contiennent les processus critique du nouveau tour. La tâche 3 met à jour l'ensemble $critical_p$ selon l'ensemble $nouveau\_critical_p$.

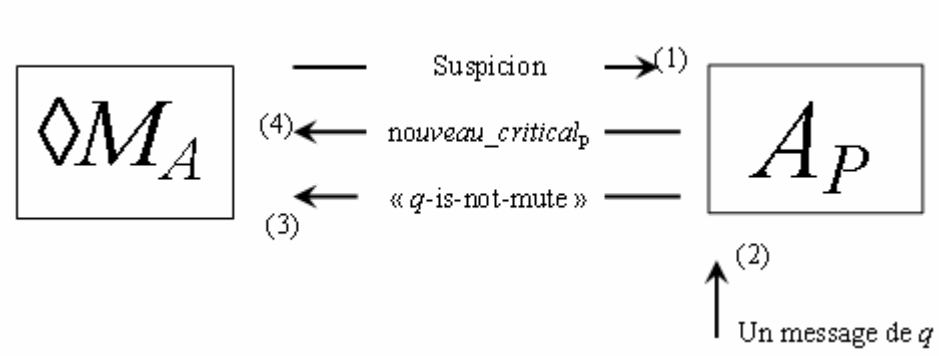

Figure 6 : L'interaction entre l'algorithme A_p et $?M_A$

En plus, la tâche 3 calcule la nouvelle valeur pour le timeout $?_p$ par l'application d'une certaine fonction g à la valeur courante de $?_p$. comme le timeout est renouvelé dans chaque tour, il existe une fonction $?_A : R ? T$ qui fait correspondre pour chaque tour n son timeout associer $?_A$(n). Par exemple : si la fonction g_A double le timeout courent $?_p$, alors $?_A$ (n) = 2^{n-1} *init*?

3.4 Comparaison entre les deux protocoles

Dans cette partie, nous avons présenté deux protocoles qui implémentent un détecteur de défaillances byzantines, chaque protocole traite un type particulier de défaillances. Dans cette section, nous comparons entre le protocole de Kihlstrom et al [36] et Doudou et al [23], selon la classe de détecteur implémentée, les défaillances qui peuvent être détectées, l'instance du consensus liée au détecteur et les inconvénients de chaque implémentation. Le tableau 3 présente certaines caractéristiques pour chaque implémentation.

Après le travail de Malkhi et Reiter [41] dans le quel une extension de détecteurs de défaillances dans un environnement byzantin a été proposée. Kihlstrom et al [36] ont introduit la première implémentation d'un détecteur de défaillances byzantin. Leur protocole implémente un détecteur de défaillances de la classe *?S(byz, A)* qui résous le problème de consensus uniforme. L'inconvénient majeur de cette implémentation qu'elle n'est pas convenable pour résoudre le problème de consensus dans environnement byzantin (un processus byzantin peut proposer une valeur différente que celle qui doit être proposée ensuite, il se comporte correctement, c'est-à-dire il n'existe aucun moyen pour que les autres processus détectent cette défaillance. Par conséquence les processus corrects peuvent décidés sur cette valeur proposée par un processus byzantin. Pour remédier à cet inconvénient, Doudou et al [23] ont utilisé le consensus sur vecteur, c'est à dire les processus corrects du système vont décider sur un vecteur et non sur une valeur. En se basant sur la décision sur un vecteur un processus byzantin peut être détecté facilement. Dans [23], Doudou et al ont défini une nouvelle classe de détecteurs de défaillances, dénoté $?M$, pour détecter les processus mutes. Ce protocole satisfait les propriétés d'accord, terminaison et validité d'un vecteur qui porte au moins $f+1$ (f est le nombre maximum de processus défaillants) valeurs proposées par des processus corrects. Les deux protocoles considèrent que le nombre de processus de système est $n \geq 3f+1$ (n est le nombre totale de processus du système et f est le nombre maximum de processus défaillants).

.

Propriété / Détecteur	Classe de détecteur	Modèle de faut	Type de consensus	Nombre de faute détecter	Inconvénient
Doudou et al	?*M*	Processus mute	Vecteur consensus	$F = \left\lceil \frac{(n-1)}{3} \right\rceil$	- Le protocole ne détecte que les processus mute
Kihlstrom et al	?*S*(*byz*, *A*)	Processus (*Byz*, *A*)	consensus fort	$F = \left\lceil \frac{(n-1)}{3} \right\rceil$	- Le protocole ne détecte que les processus qui dévient de l'algorithme *A*. - La propriété de validité définie par le consensus uniforme n'est pas convenable pour résoudre consensus dans un environnement byzantin

Tableau 3 : Comparaison entre les deux implémentations

Conclusion

Dans ce chapitre, nous avons présenté deux protocoles implémentant des classes détecteurs de défaillances byzantines différentes.

Chacun de ces deux protocoles se base un modèle de défaillances différent et aucun de ceux-ci ne couvre le problème de la détection de défaillances byzantines en entier, et chacun présente des avantages et des inconvénients par apport à l'autre.

Nous concluons à travers ces deux implémentations [23, 35], et les autres travaux qui ont étudié le problème de détection de défaillances byzantines dans les systèmes répartis asynchrones [42, 59], est que la résolution des problèmes d'accords dans un environnement réparti asynchrone sujet a des défaillances byzantines est une tâches très difficiles à réalisé.

L'objectif principal de ce chapitre est de se familiariser avec les mécanismes de détections de défaillances byzantines afin de proposer des solutions ou des algorithmes implémentant une classe de détecteurs de défaillances byzantines parmi celles existantes.

CHAPITRE IV

UNE IMPLEMENTATION DE OMEGA BYZANTIN AVEC PEU DE SYNCHRONIE

Introduction

Un détecteur de défaillances est un mécanisme qui fournit aux processus des informations concernant les processus défaillants à un instant donné. En combinant les deux propriétés qui caractérisent les détecteurs de défaillances dans un système réparti asynchrone qui sont *la complétude* et *la précision*, plusieurs classes de ceux-ci peuvent être définies [16]. Parmi celles-ci la classe du détecteur de défaillances ? qui a été proposée dans [17]. Cette classe est utilisée pour résoudre le problème d'élection d'un leader dans un système distribué asynchrone sujet à des défaillances par crash. Pour résoudre les problèmes d'accord, plusieurs solutions existantes se basent sur l'augmentation du système par un détecteur de cette classe.

Le détecteur de défaillances ? permet aux processus d'invoquer une primitive *leader* qui retourne une identité d'un processus à chaque fois est invoquée. ? garantit, après certain temps non connu mais fini, que toutes ces invocations retournent la même identité d'un processus correct du système. Ce type de détecteurs, n'est pas fiable, c'est-à-dire nous ne pouvons pas connaître quand un leader correct sera élu, et nous pouvons avoir plus d'un leader avant la vérification de propriétés de synchronie considérées dans le système.

Dans un système distribué purement asynchrone sujet à des défaillances par crash, aucun détecteur de défaillances de la classe ? ne peut être implémenté [6]. En ajoutant des hypothèses de synchronie, ? peut être implémenté [2, 26]. La plus part des implémentations de ? qui ont été proposées dans la littérature considèrent un système partiellement synchrone [16, 22], c'est-à-dire tous les canaux de communication qui interconnectent les processus du système sont inéluctablement synchrones (n^2 pour un système composé de n processus). Dans [4] Aguilera et al ont prouvé qu'il est possible d'implémenter un détecteur ? avec seulement t canaux de communication inéluctablement synchrones, tel que t est le nombre maximum de processus qui peuvent être défaillants. Cette hypothèse est très faible que celles proposées dans [16, 22]. Toutes les implémentations de ? dans un système distribué sujet à des défaillances par crash considèrent que $N \geq 2t+1$ (N le nombre de processus du système et t est le nombre maximum de processus qui peuvent être défaillants), c'est-à-dire plus de la moitié de processus sont corrects.

Dans un système distribué sujet à des défaillances byzantines, il y a très peu de travaux qui ont été proposés [23, 36]. Dans [23] Doudou et al ont proposé la classe de détecteurs de défaillances *Mute* (un cas particulier de défaillances byzantines) pour la détection de processus muts. Un processus muet envoie seulement les messages de détecteur mais il n'envoie pas ceux de l'algorithme qui l'utilise. Dans [36], Kihlstrom et al ont proposé une implémentation pour la classe $\Diamond S$ dans un environnement byzantin. Ces deux implémentations considèrent le modèle partiellement synchrone proposé dans [16], c'est-à-dire tous les canaux sont inéluctablement synchrones.

Une solution pour un problème d'accord dans un environnement byzantin exige qu'au moins deux tiers de processus soient corrects $(N \geq 3t+1)$. Ce nombre est exigé pour assurer la correction de n'importe quelle solution proposée, c'est-à-dire aucune solution n'est possible pour n'importe quel problème d'accord si cette condition n'est pas vérifiée.

Pour résoudre le consensus byzantin, Aguilera et al [6] ont proposé un protocole déterministe qui utilise une hypothèse de synchronie plus faible que celles proposées dans [16, 22]. Ce protocole exige seulement la présence de *2N* canaux de communication inéluctablement synchrones. Très récemment, Moumen et al [44] ont proposé un protocole déterministe pour résoudre le consensus byzantin. Pour cela, ils ont ajouté au système une hypothèse de synchronie très faible que celles existantes dans la littérature. Ce protocole exige la présence de 4*t* canaux de communication inéluctablement synchrones.

Dans ce chapitre, nous proposons un nouveau protocole, le premier selon nos connaissances, pour implémenter un détecteur de défaillances ? dans un environnement byzantin. Pour cela nous allons reprendre les hypothèses de synchronie les plus faibles que celles existantes. Ces hypothèses sont celles de Moumen, Mostéfaoui et Trédan [43].

4.2 Modèle du système et problème d'élection d'un leader

4.2.1 Modèle du système

Nous considérons dans ce chapitre un système distribué composé d'un ensemble fini de n processus complètement connectés $\Pi = \{p_1, p_2, ..., p_n\}$ $(n \geq 3)$. Ainsi, au maximum il existe *t* processus qui peuvent être exhibés à un comportement byzantin, c'est-à-dire un tel processus peut se comporter arbitrairement. Ceci est le modèle de défaillances le plus sévère : un processus byzantin peut crasher, arrêter l'envoie ou la réception des messages, envoyer

aléatoirement des messages, commencer arbitrairement son exécution, envoyer des valeurs différentes à des processus différents, etc. Un processus qui exhibe un comportement byzantin est dit défaillant. Autrement, il est correct.

Le réseau de communication : Le réseau de communication est fiable, c'est-à-dire qu'un message envoyé par un processus correct à un autre processus correct il sera reçu après un temps fini. Tel type de réseaux peut être implémenté même avec des canaux de communications qui perdent les messages en utilisant une simple technique de retransmission.

Propriété de synchronie et bisource : Chaque processus exécute un algorithme qui se consiste d'un ensemble d'étapes atomiques (envoyer un message, recevoir un message ou exécuter un calcul local), nous supposons que les processus sont partiellement synchrones
C'est à dire chaque processus exécute au moins une étape chaque fois que le processus le plus rapide du système exécute q étapes (q est non connue). En particulier, le délai de transfert non connu mais borné d est le temps durant lequel n'importe quel processus exécute d étapes lorsque le message inéluctablement synchrone est en transit. Par conséquent, nous pouvons utiliser une simple étape pour calculer les timeouts des différents messages. Ci-après, nous reprenons la définition de [16, 22] pour définir plus formellement les canaux synchrones et un processus bisource.

Définition 1 : *Un canal qui relie le processus p et n'importe quel autre processus q est dit synchrone à l'instant t si (1) aucun message envoyé par p à l'instant t est reçu par q après l'instant $(t + d)$ ou (2) le processus q n'est pas correct.*

Définition 2 : *Un processus p est un x-bisource à l'instant t si :*

- *(1) p est correct.*
- *(2) il existe un ensemble X de cardinalité x, tel que : pour chaque processus q dans X, les deux canaux entre p et q et entre q et p sont synchrone à l'instant t . Les processus de X sont dits les voisins privilégiés de p.*

Définition 3 : *un processus p est un ?x-bisource s'il existe un instant t tel que, pour tous $t' \geq t$, p est un x-bisource à l'instant t' .*

Pour le reste du chapitre, nous considérons un système partiellement synchrone où seulement les propriétés de synchronies supposées sont celles exigées par le *?x-bisource*. Ceci

cela signifie que tous les autres canaux qui ne participent pas dans le *?x-bisource* peuvent être complètement asynchrones.

Authentification : Un processus peut être byzantin et dissémine une valeur erronée (différentes de la valeur qui doit être obtenue s'il se comporte correctement). Pour prévenir une telle dissémination, le protocole utilise des certificats. Ceci implique l'utilisation des signatures de niveau applicatif (clé public tel que les signatures *RSA*).

Par exemple : le processus *p* peut relayer une valeur (*dit v*) reçue de la part du processus *q.* le processus *q* signe ses messages et les envoient à *p*. le processus *p* ne peut pas relayer une autre valeur *v'* s'il ne peut pas forger sa signature. Naturellement, *p* peut dire qu'il n'a reçu aucune valeur de *q* (aucun ne peut contrôler si c'est vrai ou non), mais s'il relaye une valeur de *q* il est nécessairement que la valeur est actuellement reçue du processus *q*. Ceci signifie que dans notre modèle, nous supposons que les processus byzantins ne sont pas capables de subvertir les primitives de la cryptographie. Maintenant, supposons que *p* va envoyer pour tous les processus la valeur majoritaire parmi toutes les valeurs qu'il a reçu. Le certificat, sera consisté de l'ensemble de messages reçus et signés (n'importe quel processus peut contrôler que la valeur que *p* a envoyé est réellement la valeur majoritaire). Les certificats ne peuvent pas prévenir tous les mauvais comportements de processus byzantins. Comme dans plusieurs protocoles asynchrones, durant un échange tous-à-tous, un processus attend la réception de (*n-t*) messages, autrement le processus peut être bloqué indéfiniment (naturellement un processus peut recevoir plus que (*n-t*) messages). Dans le cas général, deux ensembles de (*n-t*) messages peuvent avoir deux valeurs majoritaires différentes (chacune parmi celles-ci peut être certifiée). Un processus byzantin qui reçoit plus de (*n-t*) messages peut envoyer des valeurs majoritaires certifiées à des processus différents (dans ce cas le certificat signifie seulement que la valeur envoyée est une valeur possible).

4.2.2 Le problème d'élection d'un leader :

L'élection d'un leader est un problème fondamental dans les calculs distribués, il a été objet de plusieurs travaux [01, 02, 03, 05, 04, 19, 24, 26, 29].

Dans le problème d'élection d'un leader [05], à tout moment, au plus un processus se considère comme le leader et un nouveau leader doit être élu si le leader tombe en panne. Pour déterminer plus précisément la notion de coordonnateur (leadership), nous supposons que chaque processus a une copie locale d'une variable distribuée, dénotée par *leader*. La

copie de *leader pour* un processus p_i est dénotée par $leader_{pi}$, et pour n'importe quel processus p_i $leader_{pi} \in \{vrai, faux\}$.

Nous disons qu'un processus p_i est le *leader* à l'instant *t*, si p_i n'est pas défaillant à l'instant *t* et $leader_{pi} = vrai$. Formellement, nous définissons le problème d'élection d'un leader par les deux propriétés suivantes :

- **Accord** : à l'instant *t* il existe un seul processus leader (deux processus ne peuvent pas être leader en même temps).

- **Terminaison :** à tout moment, il existe finalement un leader.

Sabel et Marzullo [61] ont prouvé que le problème d'élection d'un leader est réductible au problème de consensus. Informellement, une utilisation du consensus pour résoudre l'élection d'un leader consiste à effectuer une décision sur le leader à élire.

Dans [17], Chandra, Hadzilacos et Toueg ont proposé un nouveau type de détecteurs de défaillances, dénoté Ω, et ont montré qu'il est le détecteur de défaillances le plus faible pour résoudre le consensus dans un système réparti asynchrone où la majorité des processus sont correct. Informellement, le détecteur de défaillances Ω est une entité distribuée qui augmente les processus avec une fonction *leader ()*. Cette fonction retourne une identité d'un processus à chaque fois est invoquée. Formellement, le détecteur de défaillances Ω doit, satisfaire la propriété suivante :

- ***Leadership inéluctable* :** il existe un instant *t* et un processus correct *p* tel que, après *t*, chaque invocation de *leader ()* par n'importe quel processus correct retourne *p*.

4.3 Le protocole byzantin :

4.3.1 Le principe du protocole :

Le protocole proposé (Algorithme 5) utilise un mécanisme d'authentification, expliqué précédemment, pour vérifier la validité de message reçus par des processus corrects. Notre protocole exige la présence d'un ?*2t-bisource*.

Chaque processus exécute l'algorithme 5. Le protocole est constitué de trois tâches différentes et une procédure StartRound(*s*). Le rôle de la procédure est de garder la confiance dans le processus *s* que représente le leader actuel puisque il est correct et les canaux de

communication, qui les relie avec les autres processus du système, sont inéluctablement synchrones. Dans le cas contraire, les processus vont passer au prochain ronde *s+1*.

Intuitivement, les processus exécutent des rondes (ou rounds) asynchrones $r = 1,2,...$ Pour commencer une ronde *k*, chaque processus envoie un message (*START, k*) à un processus spécialement désigné et déclenche le timer (linge 02), appelé le *''leader de la round k''* ; ceci est le processus (*k* mod *n*), met *k* dans *r* (ligne3), met l'identité du processus (*k* mod *n*) dans la sortie de Ω (ligne 4). Ensuite, chaque processus p_i attend la réception d'un message *(Ok, r)* du processus (*s* mod *n*) ou l'expiration du timeout. Quand le timeout expire après l'attente d'une réponse du leader de ronde courante, $?_i[s \bmod n]$ est incrémenté, ce qui permet d'atteindre une borne sur le délai d'aller-retour entre le processus p_i *et* le processus leader, si sont des voisins privilégiés. De plus, ceci prévient p_i du blocage éternel lors de l'attente d'une réponse d'un leader défaillant (ligne 9). Quand le leader de la ronde *r* reçoit un message valide *(Start, s)*, contient un numéro de ronde correcte (peut être son message lui-même), pour la première fois durant le tour (*ligne 15*), il répond par un message *(Ok, r)* à tous les processus.

Le message du leader est envoyé à partir d'une autre tâche (tâche 2) parallèle puisque le leader de ronde courante peut être occupé dans des rondes précédentes, et s'il ne peut pas répondre rapidement, le timeout comptabilisé par l'émetteur du message peut être expiré.

Si le leader courant est un *?2t-bisource*, il a au moins 2*t* voisins privilégiés parmi lesquels au moins *t* sont des processus corrects. Par conséquence, au moins *t* +1 processus corrects (les *t* voisins corrects et le leader lui-même) reçoivent la valeur *Ok* du leader et mettent leurs variables locales *H* à *Ok*. Si le leader courant est byzantin, il peut rien envoyer pour certains processus, et peut envoyer des valeurs différentes à des processus différents.

Durant la tache 1 du protocole chaque processus correct p_i attend la réponse du leader courant, s'il reçoit cette réponse, il met à jour sa variable local H_i par la valeur *reçu* si non (expiration du timeout) la variable H_i reçoit la valeur *Nok (ligne 8, 10),* ensuite, il relaye à tous les autre processus un message *Relay(r,* H_i*)* contenant la valeur H_i (ligne 11). Après un échange total de messages *Relay(r, H)* entre tous les processus du système, chaque processus attend la réception d'au moins (*n-t*) messages *Relay(r, H)* de processus distinctes, et stocke les valeurs reçus *H* dans un vecteur local V_i (ligne 12). Si un processus correct p_i collecte au moins une seule valeur *Ok*, il choisit comme leader celui de l'identité stockée dans la variable local *leader*, sinon, il décide de passer à la ronde suivante.

La tâche 3 du protocole décrit l'interaction entre un algorithme général A_i exécuté par le processus p_i, et le détecteur de défaillances Ω implémenté par le protocole de l'Algorithme 5.

Code for each process p_i:

(1) **procedure** *StartRound*(*s*)
(2) **send** (*START*, *s*) **to** *s* mod *n*; *set _ timer*($\Delta_i[s \bmod n]$);
(3) $r \leftarrow s$;;
(4) *leader* $\leftarrow s \bmod n$

On initialization:

(5) $\Delta_i[1..n] \leftarrow 1$
(6) *StartRound*(1)
(7) $\Delta_{A_i} \leftarrow 1$

Task 1:

(8) **wait until** (*OK*, *r*) received from (*s* mod *n*) or *time-out*
(9) **if** (*time-out*) **then** $H_i \leftarrow Nok$ **;** $\Delta_i[s \bmod n] \leftarrow \Delta_i[s \bmod n]+1$
(10) **else strore value in** H_i**;** *disable_ timer* **endif**;
(11) *send* RELAY(*r*, H_i) to all;
(12) **wait until** (RELAY(*r*, *) received from at least (*n* - *t*) distinct processes)
store values in V_i;
(13) **if** ($(\#_{oK}(V_i) \geq 1)$ **then** *StartRound(s*) **else** *StartRound(s*+1) **endif**;

Task 2:

(14) **loop forever**
(15) **upon** receipt of (*START*, *s*) for the first time for round r: send (*OK*, *r*) to all;

Task 3:

(16) **if** not receive ((*s* mod *n*) *is not a byzantine*) from A_i during *timeout* **then**
(17) *send* FILT(*r*, *0*) to all; $\Delta_{A_i} \leftarrow \Delta_{A_i} + 1$
(18) **when** ((*s* mod *n*) *is not a byzantine*) **received** from A_i **then** *send* FILT(*r*, *1*) to all;
(19) **wait until** (FILT(*r*, *) received from at least (*n* - *t*) distinct processes)
store values in W_i;
(20) **if** ($(\#_1(W_i) < (n-2t)$ **then** *StartRound(s*+1) **endif**;

Algorithme 5 : Un protocole implémentant Omega avec des défaillances byzantines et un processus 2t-bisource

Cette tâche est ajoutée au protocole puisque un processus byzantin peut envoyer correctement les messages liés au détecteur de défaillances, mais il peut ne pas envoyer ceux de l'algorithme qu'utilise le détecteur et aucun mécanisme ne peut le forcer pour faire ça. Le but

de cette tâche est de vérifier que le leader courant n'est pas byzantin, c'est a dire il envoie les messages qui doivent être envoyés et reçoit les messages qui doivent être reçus. La validité des messages échangés entre le détecteur et l'algorithme A_i est vérifiée grâce au mécanisme d'authentification et de certificat expliqué précédemment. Dans la section suivante, nous allons expliquer avec plus de détail l'interaction entre l'algorithme A_i et Ω.

4.3.2 Interaction entre l'algorithme A_i et ?

La figure 7 montre comment l'algorithme *Ai* exécuté par un processus p_i et Ω interagissent. Notre implémentation de Ω manipule deux interactions avec *Ai.* La tâche 3 est responsable sur ça. A chaque fois le processus p_i reçoit un message valide du processus (*s* mod *n*) qui est le leader de la ronde courante, l'algorithme A_i délivre '' (*s* mod *n*) *_is_not_byzantin''* à Ω et ce dernier envoie un message FILT(*r*, *1*) à tous les processus. (Ligne 18). Par conséquence, la tâche 3 déclenche une nouvelle ronde (*s* +1) si le leader de la ronde (*s*) est suspecté d'être byzantin. Au début de chaque ronde, le détecteur Ω fournit à l'algorithme A_i l'identité du processus leader par une simple invocation de la fonction *leader().* Si le processus p_i n'a pas reçu un message '' (*s* mod *n*) *_is_not_byzantin''* pendant un timeout Δ_{A_i}, alors il va envoyer à tous les processus un message FILT(*r*, *0*) et il incrémente Δ_{A_i}. Après cet échange, all-to-all, chaque processus attend la réception de (*n-t*) messages FILT(*r*, ***) envoyés par des processus différents et stocke les valeurs (0 ou 1) envoyées dans ceux-ci dans un vecteur W_i. Si le nombre de 1 dans W_i est inférieur à *t*+1, alors Ω déclenche une nouvelle ronde. Cela signifie que ces valeurs peuvent être envoyées par des processus byzantin.

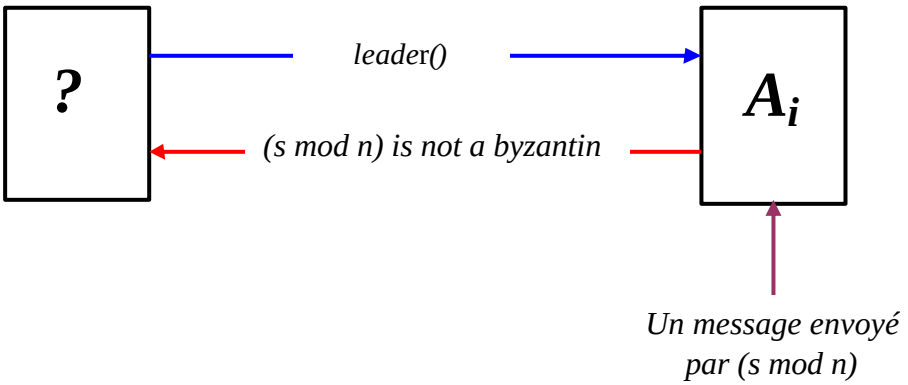

Figure 7 : Interaction entre l'algorithme A_i et un module $?_i$ associés à p_i

4.3.3 Hypothèses sur Ai pour assurer la correction de ?

Prouver la correction de notre implémentation, même quand nous considérons le modèle de synchronie définie dans [44], n'est pas une tâche facile. En effet, les délais d'échange des messages de l'algorithme *A* ne dépendent pas seulement des délais de transmission, mais ils dépendent aussi de modèle de communication de l'algorithme *A*.

Pour que notre implémentation soit correcte, nous ajoutons deux hypothèses de synchronies qui doivent être satisfaites par l'algorithme A_i.

Dans l'ordre de prouver la correction de l'implémentation du détecteur ? , nous comptons sur (1) les hypothèses de synchronies, définies précédemment, sur le détecteur ? , et (2) les hypothèses de synchronies que nous allons définir sur l'algorithme *A*. En plus, nous supposons qu'il existe toujours (*n-t*) processus corrects qui participent à l'algorithme *A*.

Hypothèses sur *A* : pour que notre implémentation soit correct l'algorithme *A* doit satisfaire les hypothèses suivantes :

- **Hypothèse (1) :** l'algorithme distribué *A* doit satisfaire les hypothèses de synchronies exigées par le détecteur de défaillances ?, c'est-à-dire il existe un processus *?2t-bisource* dans le système pour que l'algorithme *A* se termine. Si cette hypothèse n'est pas satisfaite donc si possible qu'il y aura un blocage éternel du détecteur ? ou de *A*, c'est-à-dire le modèle d'échange de messages pour ? et *A* doit être le même. les hypothèses sur les délais de communication pour *A* ne doivent pas être plus fortes que celles de ? , puisque ? et *A* sont deux entités distribués exécutées par les même processus du même système.

- **Hypothèse (2) :** Il existe un instant *t'* tel que, après *t'*, il existe des bornes non connues mais finies sur les délais d'interaction entre l'algorithme *A* et le détecteur de défaillances ? . Ces bornes ne dépendent pas de délais de transmission entre les processus du système, mais dépendent des vitesses relatives de processus, puisque ces interactions entre les deux entités sont des communications locales. Plus précisément, pour chaque processus correct p_i et après *t'*, il existe un $?_A$ non connu mais fini tel que : un message envoyé par A_i doit être reçu avant l'expiration du timeout $?_A$.

4.4 Preuve du protocole :

Lemme 1: *soient V_i et V_j les deux ensembles de messages collectés par deux processus corrects p_i et p_j respectivement aprés un échange de messages. Nous avons :*

$$V_i \cap V_j \neq \emptyset$$

Preuve : la preuve est par contradiction, supposons que $V_i \cap V_j = \emptyset$ et soit S l'ensemble de tous les messages qui peuvent être reçus par les processus p_i et p_j durant l'échange de messages, c'est à dire les messages envoyés à p_i et p_j.

Nous avons $|S| = |V_i| + |V_j|$. Ainsi $|S| = 2 \times (n-t)$ puisque chaque processus attend $(n-t)$ messages durant la phase du collection de messages d'un échange.

De plus, si f est le nombre actuel de processus byzantins ($f = t$). depuis que, les $(n-t)$ processus corrects envoient (en respectant le protocole) le même message aux deux processus et les f processus byzantins peuvent les envoient des messages différents, nous avons $|S| = 1 \times (n - f) + 2 \times f = (n+f)$, et par conséquent, $|S| = (n + t)$, puisque $f = t$.

Nous avons, $|S| = 2 \times (n - t)$ et $|S| = (n + t)$. Ceci mène à $2 \times (n - t) = (n + t)$, c'est-à-dire : $n = 3t$ est une contradiction, comme nous supposons que $n > 3t$.

Lemme 2 : *Après l'échange de messages (lignes 2 et 12), chaque processus reçoit au moins une valeur ok.*

Preuve : Nous considérons une exécution où le processus (s mod n) est un *?2t-bisource.* Un processus p_i collecte au moins un message qui porte la valeur Ok. Cette valeur est envoyée directement par le processus (s mod n) ou après le ralyement de messages. Par le *lemme 1*, y a pas un autre processus p_j qui peut exhiber un ensemble de $(n-t)$ valeurs *Nok*.

Lemme 3 : *Après une interaction entre ? et A, chaque processus reçoit au moins (t+1) messages qui portent la valeur 1.*

Preuve : Nous considérons une exécution où le processus (s mod n) est un *?2t-bisource* et les hypothèses de synchronie sur *A* sont satisfaites. Un processus p_i collecte $(n-t)$ processus dans chaque interaction entre *A* et ? . Ces messages portent au moins $(t+1)$ valeurs 1. Dans le plus pire des cas, p_i reçoit t valeurs 0 envoyées par des processus byzantins et $(t+1)$ valeurs envoyées par t processus corrects et le processus lui-même.

Théorème1 (Accord) : *deux processus corrects ne peuvent pas être leader en même temps.*

Preuve : A partir du lemme 2, du lemme 3 et le fait qu'un processus correct p_i ne peut pas commencer une nouvelle ronde s'il a reçu une valeur Ok et (t+1) valeurs durant l'interaction dans une ronde donnée, mais il va refaire la même ronde c'est-à-dire il va choisir le même leader, nous pouvons déduire facilement le Théorème1.

Lemme 4 : *Si aucun processus correct n'a pas reçu un message (OK, r) et (t+1) messages qui portent la valeur 1 durant une ronde* $r' \leq r$ *alors tous les processus correct commencent r+1.*

Preuve : Un processus correct ne peut pas être bloqué indéfiniment à la ligne 09 et la à ligne 17 à cause de timeout**.** Au début d'une ronde *r*, chaque processus p_i envoie un message (*START, s*) au leader de la ronde courante (*s* mod *n*). S'il ne répond pas, puisque il est byzantin ou les timeouts sont expirés, chaque processus p_i reçoit (*n-t*) messages qui portent seulement la valeur *Nok* et il commence une nouvelle ronde *r+1*. Si le leader de la ronde courante (*s* mod *n*) répond sur le message (*START, s*), alors le processus p_i reçoit au moins une valeur Ok et il examine que le processus (*s* mod *n*) envoie et reçoit correctement les messages de l'algorithme *A* avec lequel il est lié. Si p_i reçoit moins de (*t*+1) messages qui portent la valeur 1, donc il commence une nouvelle ronde, c'est-à-dire ces valeurs, si possiblement, sont envoyées par des processus byzantins.

Théorème 2 (*Terminaison*): *S'il existe un ?2t-bisource dans le système, alors chaque invocation de leader() retourne l'identité du processus ?2t-bisource.*

Preuve : A partir du le Lemme 4 et le fait qu'un processus *?2t-bisource* existe dans le système, nous pouvons facilement prouver le Théorème 2. Si le processus (*s* mod *n*) ne répond pas sur le message (*START, s*) (puisque il est défaillant), tous les processus corrects du système ne reçoivent pas sa réponse (puisque les timeouts sont expirés) ou il ne respecte pas la spécification de l'algorithme *A*, donc tous les processus commencent une nouvelle ronde et ainsi de suite jusqu'à ce que le processus (s mod n) soit un *?2t-bisource*, c'est-à-dire un ?*2t-bisource* est le leader. A partir de l'instant dans lequel un *?2t-bisource* est le leader, chaque invocation de la fonction *leader()* retourne l'identité du *?2t-bisource qui égale à* (*s* mod *n*).

4.5 Discussion :

La spécification de détecteurs de défaillances est indépendante des algorithmes qui utilisent tels détecteurs. Dans le contexte byzantin, aucune spécification de détecteur de défaillances ne peut respecter cet aspect. Ceci dû à la nature inhérente de défaillances byzantines.

Notre implémentation ne peut pas conserver la notion de l'approche modulaire proposée dans [16], pour implémenter un détecteur de défaillances crashs indépendamment de l'algorithme qui l'utilise. Cette approche rend possible de voir un détecteur de défaillances comme une *boite noire* dont chaque implémentation peut être changée sans aucun impact sur l'algorithme qui l'utilise, tant qu'il obéisse à ses spécifications, c'est-à-dire il conserve ses propriétés. Un détecteur de défaillances byzantines ne peut pas être spécifié indépendamment de l'algorithme qui l'utilise et son implémentation ne peut être vue comme une boite *noire*, puisque un processus byzantin ne peut pas respecter les spécifications du détecteur, mais il peut violer les spécifications de l'algorithme qui utilise le détecteur. Cette dépendance peut même restreindre l'ensemble d'algorithmes qui peuvent utiliser les détecteurs de défaillances.

Dans [16], tous les aspects liés à la détection de défaillances sont encapsulés dans le module de détecteur de défaillances. Dans le contexte byzantin, les défaillances ont plusieurs faces, certain de celles-ci ne peuvent pas encapsuler dans le module de détecteur de défaillances.

4.6 Conclusion :

Dans ce chapitre, nous avons proposé un nouveau protocole pour implémenter un détecteur de défaillances Omega (le premier selon nos connaissances) dans un système réparti asynchrone sujet à des défaillances byzantines. Notre protocole exige la présence d'au moins de $4t$ canaux de communication inéluctablement synchrones. Ces canaux connectent le même processus ($2t$ entrants et $2t$ sortants). Cette hypothèse de synchronie est la plus faible que celles existantes [44]. La conception du protocole proposé est très simple que ceux existants.

L'objectif principal de ce chapitre est de démonter que le détecteur de défaillances Oméga peut être implémenté avec des hypothèses de synchronie très faibles dans un système distribués sujet à des défaillances byzantines.

CHAPITRE V

UNE IMPLEMENTATION ASYNCHRONE DE OMEGA BYZANTIN

Introduction

Dans un système réparti purement synchrone aucune implémentation d'un détecteur de défaillances n'est possible sans l'ajout des hypothèses supplémentaires. Il existe deux type d'implémentations de détecteurs de défaillances crashs : celles qui utilisent le temps physique [16] et celles qui ne l'utilisent pas [45, 46], et enfin nous avons des solutions hybride [46, 51]. Dans le contexte byzantin, toutes les implémentations qui ont été proposées dans la littérature utilisent le temps physique (timeout).

Dans ce chapitre, nous allons définir une nouvelle hypothèse sur le comportement d'échange de messages pour implémenter le détecteur de défaillances byzantines Ω. Pour cela, nous reprenons la définition du canal *wining* proposée par Mostéfaoui et al dans [46, 51] pour définir la notion du processus *2t-wining*.

Dans ce chapitre, nous présentons une nouvelle implémentation, la deuxième selon nos connaissances, pour Ω dans un environnement byzantin et nous montrons qu'une implémentation de Ω est possible avec un processus *2t-wining*.

5.2 Modèle du système

Nous considérons dans ce chapitre un système distribué composé d'un ensemble fini de n processus complètement connectés $\Pi = \{p_1, p_2, ..., p_n\}$ $(n \geq 3)$. Ainsi, au maximum il existe t processus qui peuvent être exhibés à un comportement byzantin Ω, c'est-à-dire un tel processus peut se comporter arbitrairement. Ceci est le modèle de défaillances le plus sévère : un processus byzantin peut crasher, arrêter l'envoie ou la réception des messages, envoyer aléatoirement des messages, commencer arbitrairement son exécution, envoyer des valeurs différentes à des processus différents, etc. Un processus qui exhibe un comportement byzantin est dit défaillant. Autrement, il est correct.

Le réseau de communication :

Le réseau de communication est fiable, c'est-à-dire qu'un message envoyé par un processus correct à un autre correct il sera reçu après un temps fini. Tel type de réseaux peut

être implémenté même avec des canaux de communications qui perdent les messages en utilisant une simple technique de retransmission.

Une propriété sur le comportement du système :

Comme il est impossible d'élire un leader dans un système réparti asynchrone sujet à des défaillances byzantines, nous allons définir une hypothèse supplémentaire sur le modèle d'échange de messages. Ces hypothèses n'utilisent pas le temps physique et se basent sur le comportement du système et la notion de canal *wining* défini dans [46, 51].

Définition 1: *Un processus p est un x-wining à l'instant t si :*

- *(1) p est correct.*
- *(2) il existe un ensemble X de la cardinalité x, tel que : pour chaque processus q dans X, les deux canaux de p à q et de q à p sont les canaux les plus rapides du système à l'instant t . Les processus de X sont dits les voisins privilégiés de p.*

Définition 2 : *un processus p est un ?x-wining s'il existe un instant t tel que, pour tous $t' \geq t$, p est un x-wining à l'instant t' .*

Ceci peut être interprété comme suit : parmi les *n* processus, il existe un processus correct qui possède *x* voisins privilégies avec lesquels il communique plus rapidement que les autres processus. Plus précisément, si *x* processus correct du système envoient un message *m* à un processus correct *p* et envoient un autre message *m'à* tous les autres processus à l'exception de *p,* si ces *x* processus reçoivent toujours la réponse de *p* sur *m* avant la réception d'une réponse sur *m',* alors nous pouvons dire que *p* est un *x-wining.* Ceci signifie que les communications entre *p* et ses *x* voisins privilégies sont toujours les plus rapides dans le système par rapport aux autres communications.

Authentification :

Pour vérifier la validité de messages échangés, nous allons reprendre le même mécanisme d'authentification utilisé dans le chapitre précédent.

5.3 Le protocole byzantin :

5.3.1 Principe du protocole :

Le protocole présenté par l'Algorithme 6 propose une implémentation de détecteur de défaillances O en se basant sur l'existence d'au moins d'un *?2t-wining* dans le système. Le principe de fonctionnement de ce protocole est similaire à celui présenté dans le chapitre précédent à l'exception de certaines modifications pour l'adapter à la nouvelle hypothèse considérée au début de ce chapitre.

Code for each process p_i:

1) **procedure** *StartRound*(*s*)
2) **send** (*START, s*) **to** *s* mod *n*;
3) **send** *Q*(specific) to (? - (*s* mod *n*));
4) $r \leftarrow s$;
5) *leader* $\leftarrow$ *s* mod *n*
6) **on initialization:**
7) *StartRound*(1)

Task 1:
8) **wait until** (*OK*, *r*) received from (*s* mod *n*) **or** receipt *R*(*specific*) for the first time four round *r*
9) **if** *R*(*specific*) **then** $H_i \leftarrow Nok$ **; else** $H_i \leftarrow ok$ **endif**;
10) *send* RELAY(*r*, H_i) to all;
11) **wait until** (RELAY(*r*, *) received from at least (*n* - *t*) distinct processes) **store values** in V_i;
12) **if** ($(\#_{oK}(V_i) \geq 1)$ **then** *StartRound(s)* **else** *StartRound (s+1)* **endif**;

Task 2:
13) **loop forever**
14) **upon** receipt of Q(*specific*) for the fiirst time for round r : send *R*(*specific*) to (? - (*s* mod *n*));
15) **upon** receipt of (*START, s*) for the first time for round r: send (*OK*, *r*) to all;

Task 3:
16) **loop forever**
17) **send** *Q*((*s* mod *n*) *is it a byzantine?)* to A_i;
18) **send** *Q*(specific1) to (all - p_i);
19) **if (** not receive ((*s* mod *n*) *is a byzantine*) from A_i before receipt of *R*(*specific1*) **)then** *send RELAY1(r, 0)* to all;
20) **else** *send* RELAY1(r, 1) to all;
21) **wait until** (RELAY1(*r*, *) received from at least (*n* - *t*) distinct processes) **store values** in W_i;
22) **if** ($(\#_1(W_i) < n - 2t)$ **then** (*START, s*+1) **endif**;

Algorithme 6 : Un protocole implémentant Omega byzantin avec un ?2t-wining

Dans ce protocole, nous avons ajouté un autre échange de messages, chaque processus envoie *Q*(specific) à tous les processus à l'exception du processus (*s* mod *n*) (ligne 03) et attend la réception d'un message *(Ok, r)* du processus (*s* mod *n*) ou de la première réponse *R*(*specific*) (ligne 09) d'un processus quelconque. Cette interogation-réponse (*Q*(specific), *R*(*specific*)) est

ajoutée pour remplacer les timeouts du protocole précédemment présenté et pour éviter tout blocage du protocole dans le cas où le processus (*s* mod *n*) ne répond pas.

Dans la tâche 3, nous avons remplacé les timeouts Δ_A par un autre échange de messages. Chaque processus p_i envoie un message Q*((s* mod *n) is it a byzantine?),* et juste après il envoie *Q*(specific1) à tous les processus à l'exception de p_i, et il attend la réception de ((*s* mod *n*) *is a byzantine*) de A_i ou la réception de *R*(*specific1*) dans le cas où p_i est défaillant. Pour le reste du protocole il n'y a aucun changement par rapport au protocole du chapitre précèdent.

5.3.2 Interaction entre l'algorithme A_i et ?

La figure 8 montre comment l'algorithme A_i et Ω s'interagissent. L'interaction entre *Ai* et Ω est la même que celle présentée dans le chapitre précédent à l'exception de quelque modification. Au début de la tâche 3, chaque processus p_i envoie à A_i un message *(s mod n) is it a byzantin ?* , et attend la réception de *((s mod n) is not a byzantin)* ou de *R*(*specific1*) dans le cas où p_i est défaillant.

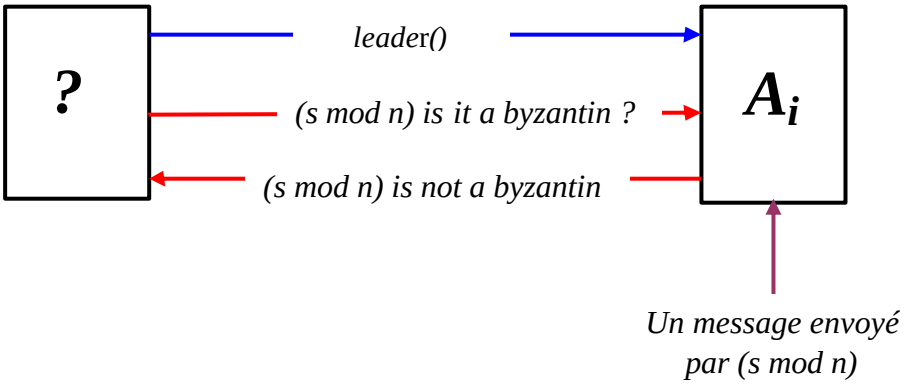

Figure 8 : Interaction entre l'algorithme A et un module ? $_i$ associé à p_i

5.3.3 hypothèses pour prouver la correction de ?

Pour que notre implémentation du détecteur de défaillances ? soit correcte, nous ajoutons deux hypothèses comportementales qui doivent être satisfaites par l'algorithme A_i.

- **Hypothèse (1) :** l'algorithme distribué *A* doit satisfaire l'hypothèse comportementale exigée par le détecteur de défaillances ? , c'est-à-dire il existe un *?2t-wining* dans le système pour que l'algorithme *A* se termine.
- **Hypothèse (2) :** Il existe un instant t'' tel que, après t'', chaque processus correct p_i , lorsque il envoie *((s mod n) is it a byzantin ?)* et *Q*(*specific1*), reçoit un message ((*s*

mod n) is not a byzantin) avant la réception de premier message *R*(*specific1*), c'est-a-dire la communication entre A_i et Ω est la plus rapide par rapport aux autres communications.

5.4 Preuve du protocole :

Lemme 5: *soient V_i et V_j les deux ensembles de messages collectés par deux processus corrects p_i et p_j respectivement aprés un échange de messages. Nous avons :*

$$V_i \text{ I } V_j \neq f$$

Preuve : la preuve est par contradiction, supposons que V_i I $V_j = f$ et soit *S* l'ensemble de tous les messages qui peuvent être reçus par les processus p_i et p_j durant l'échange de messages, c'est à dire les messages envoyés à p_i et p_j.

Nous avons |S|= |V_i| + |V_j|. Ainsi |S| = 2 × (*n*-*t*) puisque chaque processus attend (*n*-*t*) messages durant la phase du collection de messages d'un échange.
De plus, si *f* et le nombre actuel de processus byzantins (*f* = *t*). depuis que, les (*n*-*t*) processus corrects envoient (en respectant le protocole) le même message aux deux processus et les *f* processus byzantins peuvent les envoient des messages différents, nous avons |S| = 1 × (*n* – *f*) + *2*× *f* = *(n+f)*, et par conséquent, |S| = (*n* + *t*), puisque *f* = *t*.

Nous avons, |S| = 2 × (*n* - *t*) et |S| = (*n* + *t*). Ceci mène à 2 × (*n* - *t*) = (*n* + *t*), est à dire *n*= 3t est une contradiction, comme nous supposons que *n* > 3t.

Lemme 6 : *Après l'échange de messages (lignes 2 et 11), chaque processus reçoit au moins une valeur ok.*

Preuve : Nous considérons une exécution où le processus (*s* mod *n*) est un *?2t-wining*. Un processus p_i collecte au moins un message qui porte la valeur Ok. Cette valeur est envoyée directement par le processus (*s* mod *n*) ou après le ralyement de messages. Par le lemme 5, y a pas un autre processus p_j qui peut exhiber un ensemble de (*n*-*t*) valeurs *Nok*.

Lemme 7 : *Après une interaction entre ? et A, chaque processus reçoit au moins (t+1) messages qui portent la valeur 1.*

Preuve : Nous considérons une exécution où le processus (*s* mod *n*) est un *?2t-winnig* et l'hypothèse 2 sur *A* est satisfaite. Un processus p_i collecte (*n*-*t*) processus dans chaque

interaction entre *A* et ?. Ces messages portent au moins (*t*+1) valeurs 1. Dans le plus pire des cas, p_i reçoit *t* valeurs 0 envoyées par des processus byzantins et (*t*+1) valeurs envoyées par *t* processus corrects et le processus lui-même.

Théorème 3 (Accord) : *deux processus corrects ne peuvent pas être leader en même temps.*

Preuve : A partir du lemme 6, du lemme 7 et le fait qu'un processus correct p_i ne peut pas commencer une nouvelle ronde s'il a reçu une valeur Ok et (t+1) valeurs 1 durant l'interaction dans une ronde donnée, mais il va refaire la même ronde c'est-à-dire il va choisir le même leader, nous pouvons déduire facilement le Théorème3.

Lemme 8*: Si aucun processus correct n'a reçu un message (OK, r) et (t+1) messages qui portent la valeur 1 durant une ronde* $r' \leq r$ *alors tous les processus correct commence r+1.*

Preuve : Un processus correct p_i ne peut pas être bloqué indéfiniment à la ligne 08 et a la ligne 19 à cause de ne pas recevoir un message *R(specific)* puisque, il envoie un message *Q(specific)* à tous les processus du système à l'exception du leader de la ronde courante, donc il va recevoir au moins (*n-t*-1) réponses**.** Au début d'une ronde *r*, chaque processus p_i envoie un message (*START, s*) au leader de la ronde courante (*s* mod *n*). S'il ne répond pas, puisque il est byzantin ou il a reçu un message *R(specific)*, chaque processus p_i reçoit (*n-t*) messages qui portent seulement la valeur *Nok* et il commence une nouvelle ronde *r+1* . Si le leader de la ronde courante (*s* mod *n*) répond sur le message (*START, s*), alors le processus p_i reçoit au moins une valeur Ok et il examine que le processus (*s* mod *n*) envoie et reçoit correctement les messages de l'algorithme *A* avec lequel il est lié. Si p_i reçoit moins de (*t*+1) messages qui portent la valeur 1, donc il commence une nouvelle ronde, c'est-à-dire que ces valeurs, si possiblement, sont envoyées par des processus byzantins.

Théorème 4 (terminaison) : *S'il existe un ?2t-wining dans le système, alors chaque invocation de leader () retourne l'identité du processus ?2t-wining.*

Preuve : A partir du le Lemme 8 et le fait qu'un processus *?2t-wining* existe dans le système, nous pouvons facilement prouver le Théorème 4. Si le processus (*s* mod *n*) né répond pas sur le message (*START, s*) (puisque il est défaillant), tous les processus corrects du système ne reçoivent pas sa réponse (puisque ils ont reçu un message *R(specific)*) ou il ne respectent pas la spécification de l'algorithme *A*, donc tous les processus commencent une nouvelle ronde et ainsi de suite jusqu'à ce que le processus (*s* mod n) soit un *?2t-wining*, c'est-à-dire un ?*2t-*

wining est le leader. A partir de l'instant dans lequel un *?2t-wining* est le leader, chaque invocation de la fonction *leader()* retourne l'identité du *?2t-wining* qui est (*s* mod *n*).

5.5 Conclusion :

Dans ce chapitre, nous avons proposé un autre protocole pour implémenter un détecteur de défaillances Omega (le deuxième selon nos connaissances) dans un système réparti asynchrone sujet à des défaillances byzantines. Notre protocole exige la présence d'au moins de 2*t* canaux de communication inéluctablement *winings.* Cette hypothèse sur le comportement du système (le modèle d'échange de message) est utilisée, pour la première fois, pour implémenter un algorithme distribué dans un environnement byzantin. L'objectif principal de ce chapitre est de monter que le détecteur de défaillances Oméga peut être implémenté avec des hypothèses comportementales très faibles. Dans le meilleurs cas, après l'existence d'un ?*2t-wining*, notre algorithme assure que chaque invocation de *leader()* retourne un leader correct dans la première ronde, c'est dire le processus p_1 est un *?2t-wining*.

CONCLUSION GENERALE

D'une manière générale, l'objectif principal de ce mémoire résidait dans l'étude des implémentations de mécanismes de détection de défaillances dans le modèle augmenté par les deux hypothèses les plus sévères dans les systèmes répartis qui sont l'Asynchronisme et les défaillances Byzantines. Implémenter un algorithme réparti dans tel type de systèmes est une tâche ardue et très difficile à réaliser. Un premier objectif concerne une introduction et une synthèse sur les systèmes répartis, les problèmes d'accord, et les détecteurs de défaillances crashs. Un deuxième objectif est une étude des algorithmes implémentant les détecteurs de défaillances dans un environnement byzantin. A cause des difficultés rencontrées dans ce type de système, il y a seulement deux implémentations qui ont été proposées dans la littérature. Nous avons étudié les hypothèses ajoutées au modèle du système pour chaque implémentation afin de comparer entre celles-ci. Enfin, et comme dernier objectif, nous avons proposé deux implémentations de détecteurs de défaillances byzantines de la classe *Omega* qui est la classe la plus faible et la plus importante pour résoudre le problème du consensus. Dans la première implémentation, nous avons réutilisé l'hypothèse de synchronie la plus faible que celles existantes. Cette hypothèse est la notion de *2t-bisource* définie par Moumen, Mostéfaoui et Trédan [44]. Dans la deuxième implémentation, nous avons défini la notion de *2t-wining* qu'est une hypothèse suffisante pour implémenter Omega byzantin. La seule différence entre les deux implémentations que nous avons proposées est le type d'hypothèses ajoutées au modèle de système. La première manipule les horloges physiques de processus et la deuxième se base sur le comportement du système ou le modèle d'échange de messages. Nous avons prouvé également la correction de chacune des ces implémentations proposées

Perspectives

Les perspectives à ce mémoire sont diverses, nous en citons ci-après deux entre elles.

Hypothèses minimales pour l'implémentation des détecteurs de défaillances :

Nous avons réutilisé la notion de *2t-bisource* proposée par Moumen, Mostéfaoui et Trédan et nous avons défini la notion de 2t-*wining*. Ces deux notions sont des hypothèses suffisantes pour l'implémentation de détecteurs de défaillances byzantines de la classe *Oméga*. Un problème reste ouvert concerne les hypothèses minimales (nécessaires) pour qu'une implémentation de *Oméga* soit possible.

Résoudre le consensus :

Nous avons vu que les deux protocoles proposés peuvent être utilisés pour résoudre le consensus. Un protocole du consensus qui se base sur l'un des deux mécanismes de détections de défaillances byzantines doit se baser sur les mêmes hypothèses (*2t-wining* et *2t-bisource*) ou sur d'autres plus faibles.

Bibliographie

[01] M. K. Aguilera, W. Chen, and S. Toueg. Heartbeat: A timeout-free failure detector for quiescent reliable communication. In Proceedings of the 11th International Workshop on Distributed Algorithms (WDAG'97), September 1997, 126-140.

[02] M. K. Aguilera, C. Delporte-Gallet, H. Fauconnier, and S. Toueg. On implementing Omega with weak reliability and synchrony assumptions. Technical Report, LIAFA, Universit´e Paris 7-Denis Diderot (France), 2003.

[03] Marcos K. Aguilera, Carole Delporte-Gallet, Hugues Fauconnier, and Sam Toueg. Stable leader election. Lecture Notes in Computer Science, In Proceedings of the 15th International Conference, DISC 2001, Lisbon, Portugal. Vol. 2180, 2001, 108–122.

[04] Aguilera M.K., Delporte-Gallet C., Fauconnier H. and Toueg S., On Implementing Omega with Weak Reliability and Synchrony Assumptions. 22th ACM Symposium on Principles of Distributed Computing, 2003, 306-314.

[05] Aguilera M.K., Delporte-Gallet C., Fauconnier H. and Toueg S., Communication Efficient Leader Election and Consensus with Limited Link Synchrony. 23th ACM Symposium on Principles of Distributed Computing, 2004.

[06] Aguilera M.K., Delporte-Gallet C., Fauconnier H. and Toueg S. Consensus with byzantine failures and little system synchrony. In Proceedings of International Conference on Dependable Systems and Networks (DSN'06), Philadelphia, 2006, 147-155.

[07] Alvisi, L. Malkhi, D. Pierce, E. and Reiter, M. K. Fault detection for Byzantine quorum systems. IEEE Trans. Parallel Distrib. Syst. Vol.12, 2001 996–1007.

[08] Bracha, G. (1987) Asynchronous Byzantine agreement protocols. Inform. Computat., vol. 75, Nov. 1987, 130-143.

[09] Bertier M., Marin O. and Sens P., Implementation and Performance Evaluation of an Adaptable Failure Detector. In Proceedings of int. IEEE Conference on Dependable Systems and Networks, IEEE Computer Society Press, Washington D.C. Vol. 2, 2002, 354-363,

Bibliographie

[10] M. Ben-Or. Another advantage of free choice: Completely asynchronous agreement protocols. In 2nd ACM Symp. On Principles of Distributed Computing, Vol. 83, 1983, 27-30.

[11] M. Ben-Or. Fast asynchronous Byzantine agreement. In Proceedings of the 4th ACM Symposium on Principles of Distributed Computing, 1985, 149-151.

[12] Coan, B. A. A compiler that increases the fault tolerance of asynchronous protocols. IEEE Trans. Comput., Vol. **37**, 1988, 1541–1553.

[13] B.Charron-Bost and A.Schiper. Uniform consensus is harder than consensus (extended abstract). Technical Report, EPFL, DSC/200/028 2000.

[14] F. Cristian and C. Fetzer. The timed asynchronous distributed system model. IEEE Trans. On Parallel & Distributed Systems, Vol. 10(6), ,June1999, 642–657.

[15] G. Chockler, I. Keidar, and R. Vitenberg. Group communication specifications: A comprehensive study. ACM Computing Surveys, Vol. 33(4): May 2001, 427–469.

[16] T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. Journal of the. ACM, Vol. 43(2): March 1996, 225-267

[17] T. D. Chandra, V. Hadzilacos, and S. Toueg. The weakest failure detector for solving consensus. J. ACM, Vol. 43(4): July 1996, 685–722.

[18] T. D. Chandra, V. Hadzilacos, S. Toueg, and B. Charron-Bost. On the impossibility of group member ship. In Proceedings of 15th Annual ACM Symp. On Principles of Distributed Computing. 685–722,Vol 86, May 1996, 322–330.

[19] F. Chu. Reducing omega to ◊w. Information Processing Letters, Vol. 67(6): 1998, 293–298.

[20] D. Dolev, C. Dwork, and L. Stockmeyer. On the minimal synchronism needed for distributed consensus. Journal of the ACM, Vol. 34(1), January 1987, 77-98.

Bibliographie

[21] C. Delporte-Gallet, H. Fauconnier, R. Guerraoui. A Realistic Look At Failure Detectors. In Proceedings of the IEEE International Conference on Distributed Computing Systems. Washington DC. Vol. 2, Jun. 2002, 345- 353.

[22] Dwork C., Lynch N.A. and Stockmeyer L., Consensus in the presence of partial synchrony. Journal of the ACM, Vol. 35(2): 1988, 288-323.

[23] Doudou A. and Schiper A. Muteness Failure Detector for Consensus with Byzantine Processes. In Proceedings of. 17th ACM symposium on principles of Distributed Computing. EPFL, Lausanne, Switzerland. Vol. 97/30, 1998, 315.

[24] Eli Gafni and Leslie Lamport. Disk paxos. In proceedings of the 14th International Conference on Distributed Computing.Vol 42, 2000, 330-344.

[25] M. Fisher, N. Lynch, and M. Paterson. Impossibility of distributed consensus with one faulty process. Journal of the ACM, Vol. 32(2): 1985, 374-382.

[26] Greg N. Frederickson and Nancy A. Lynch. Electing a leader in a synchronous ring. Journal of the ACM, Vol. 34(1): 1987, 98–115.

[27] P. Feldman and S. Micali. An Optimal Algorithm For Synchronous Byzantine Agreement, Technical report MIT/LCS/TM-425 Massachusetts Institute of Technology. Vol. 26, 1997 June 1990, 873–933.

[28] Fetzer Ch., Raynal M. and Tronel F., An Adaptive Failure Detection Protocol. In Proceeding of 8th IEEE Pacific Rim Int. symposium on Dependable Computing (PRDC'01), IEEE Computer Society Press, Seoul (Korea), 200, 146-1531.

[29] Hector Garcia-Molina. Elections in a distributed computing system. IEEE Transaction on Computers, Vol. 31(1): January 1982, 47–59.

[30] Gupta I, Chandra T.D. and Goldszmidt G.S., On Scalable and Efficient Distributed Failure Detectors. In Proceedings of 20th ACM Symposium on Principles of Distributed Computing, ACM Press. Vol. 01: 2001, 170-179.

Bibliographie

[31] R. Guerraoui and P. Dutta. Fast indulgent consensus with zero degradation. In Proceedings of the 4th European Dependable Computing Conference, Oct. 2002.

[32] R. Guerraoui and A. Schiper. A modular approach for building fault-tolerant agreement protocols in distributed systems. In Proceedings of the IEEE Intenational Symposium on Fault-Tolerant Computing Systems (FTCS), Sendai (Japon), June 1996, 168-177.

[33] E. Gafni and L. Lamport. Disk paxos. In Proceedings of the 14th International Symposium on Distributed Computing, Springer-Verlag. Vol.1914: 2000, 330–344.

[34] Hurfin M. and Raynal M., A Simple and Fast Asynchronous Consensus Protocol Based on a Weak Failure Detector. To appear in Distributed Computing, IRISA, France. Vol. 12(4), 1999.

[35] V. Hadzilacos and S. Toueg. Reliable broadcast and related problems. Jounal Of ACM, Vol. 32(2), April 1985, :374–382.

[36] K. P. Kihlstrom, Louise E. Moser, and P. M. Melliar-Smith. Solving consensus in a Byzantine environment using an unreliable fault detector. In Proceedings of the International Conference on Principles of Distributed Systems (OPODIS). Vol. 75: December 1997, 61.

[37] Larea M., Arévalo S. and Fernàndez A., Efficient Algorithms to Implement Unreliable Failure Detectors in Partially Synchronous Systems. Proc. 13th Symposium on Distributed Computing (DISC'99), Bratislava (Slovakia), Springer Verlag. Vol. 1693, 1999, 34-48.

[38] M. Larrea, A. Fernandez, and S. Arévalo. Eventually consistent failure detectors. Technical Report, Universidad Publica de Navarra, April 2000. Accepted as brief announcement in the 14th International Symposium on Distributed Computing (DISC'2000), Toledo, Spain. Vol. 65(3): 2005, 361-373.

[39] M. Larrea, A. Fernandez, and S. Arevalo. Eventually consistent failure detectors. In ACM Symposium on Parallel Algorithms and Architectures. Vol 65: 2001, 326–327.

[40] M. Larrea, A. Fernandez, and S. Arévalo. On the Impossibility of Implementing Perpetual Failure Detectors in Partially Synchronous Systems. In Proceedings of the 10th Euromicro Workshop on Parallel, Distributed and Network-based. Vol. 02: 2002, 1066-6192.

[41] L. Lamport, R. Shostak, and L. Pease. The Byzantine general problem. In ACM Transaction on Programming Languages and Systems.1982, 382-401.

[42] Malkhi D. and Reiter M. Unreliable Intrusion Detection in Distributed Computations. In Proc. of the 10th IEEE Computer Security Foundations Workshop, pp. 116_124, Rockport (MA), June 1997.

[43] A. and M. Raynal. Leader-based consensus. Parallel Processing Letters, Vol. 11(1): 2001, 95–107.

[44] Moumen. H, Mostéfaoui. A and Gilles. T. Byzantine Consensus with Few Synchronous Links. IRISA, Université de Rennes 1, Campus de Beaulieu, 35042 Rennes, France

[45] A. Mostéfaoui, E. Mourgaya, M. Raynal, and Ph. Raïpin. Evaluating the condition-based approach to solve consensus. In Proc. of the Int. Conf. on Dependable Systems and Networks (DSN03), San Francisco, CA, Juliet 2003.

[46] Mostefaoui A., Powell D., and Raynal M., A Hybrid Approach for Building Eventually Accurate Failure Detectors. In Proceedings of 10th IEEE Int. Pacific Rim Dependable Computing Symposium. Vol. 04: 2004, 57-65.

[47] A. Mostéfaoui, S. Rajsbaum, and M. Raynal. Conditions on input vectors for consensus solvability in asynchronous distributed systems. In Proceedings of the 33rd ACM Symposium on Theory of Computing (STOC'01), ACM Press. 2001, 153-162

[48] A. Mostéfaoui, 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). IEEE, 2002.

[49] Mostéfaoui, S. Rajsbaum, M. Raynal, and M. Roy. Efficient condition-based consensus. In Proceedings of the 8th Int. Colloquium on Structural Information and Communication Complexity. Vol. 01: 2001, 275 – 291.

[50] A. Mostéfaoui, M. Raynal, and F. Tronel. The best of both worlds: a hybrid approach to solve consensus. In Proceedings of the Int. Conference on Dependable Systems and Networks (FTCS/DCCA), New York, NY. IEEE. Jun 2000, 513-522.

[51] A. Mostéfaoui, M. Raynal, and F. Tronel. The best of both worlds: a hybrid approach to solve consensus. In Proceedings of the Int. Conference on Dependable Systems and Networks (FTCS/DCCA), New York, NY, IEEE. Jun 2000, 513-522.

[52] Neiger, G. and Toueg, S. Automatically increasing the fault-tolerance of distributed algorithms. J. Algorithms. Vol. **11**: 1990, 374–419.

[53] D. Powell. Special issue on group communication. In CACM. Vol. 39 (4) 1996.

[54] K.J. Perry and S. Toueg. Distributed agreement in the presence of processor and communication faults. IEEE Transactions on Software Engineering. Vol. 3: 1986, 477-482.

[55] Rivest, R. L. The MD5 Message Digest Algorithm. Network Working Group, MIT Laboratory for Computer Science and RSA Data Security, Inc., Cambridge, MA. RFC 1321, 1992.

[56] P. Raipin Parevédy and M. Raynal. Uniform agreement despite process omission failures. In Proceedings of IEEE IPDPS workshop on fault-tolerant Parallel and distributed Systems (FTPDS'03). Vol. 212: 2003.

[57] P.Raipin Parevédy and M. Raynal. Optimal early stopping uniform consensus in synchronous system with process omission failures. In Proceedings of Sixteenth ACM symposium on Parallelism in Algorithms and Architectures (SPAA'04). ACM Press, 2004.

[58] Rivest, R. L., Shamir, A. and Adleman, L. A method for obtaining digital signatures and public-key cryptosystems. In proceedings of ACM. Vol. 21: 1978, 120–126.

[59] Roberto, B. Jean-Michel, H. Michel, R. Lenaik, T. Consensus in Byzantine Asynchronous Systems. In proceedings of ACM, Vol. 43(4): July 2003, 685–722.

[60] A. Schiper. Early consensus in an asynchronous system with a weak failure detector. Distributed Computing, Vol. 10(3): April 1997, 149-157,.

[61] L. Sabel and K. Marzullo. Election Vs. Consensus in Asynchronous Systems. Technical Report, Cornell UnivAlso, CS95-411, UCSD, 1995.

[62] A. Zamsky. A randomized byzantine agreement protocol with constant expected time and guaranteed termination in optimal (deterministic) time. In Proceeding of the 15h Annual ACM Symposium on Principles of distributed computing., Vol. 96: May 1996, 201 – 208.