République Algérienne Démocratique et Populaire
Ministère de l'Enseignement Supérieur et de la Recherche Scientifique.
Université A/Mira de Béjaia
Faculté des Sciences Exactes
Département d'Informatique
Ecole Doctorale d'Informatique

Mémoire de Magister En Informatique

Option : **Réseaux et Systèmes Distribués**

Présenté par : NEZZAR Abderafik

Thème

Accord byzantin dans les systèmes répartis

Soutenu le : 24 avril 2008

Jury :

- Président	**: BIBI Mohand Ouamer**	**Professeur, Université de Béjaia**
- Rapporteur	**: BELMEHDI Ali**	**Professeur, Université de Béjaia**
- Examinateur	**: BOUKERRAM Abdallah**	**M C, Université de Sétif**
- Examinateur	**: MOUSSAOUI Abdelouahab**	**M C, Université de Sétif**
- Invité	**: MOUMEN Hamouma,**	**M A, Université de Béjaia**

Remerciements

Je remercie vivement Monsieur le Professeur BELMEHDI Ali, mon promoteur, pour son accueil, pour ses valeureux conseils, sa patience et de m'avoir accompagné tout au long de ce travail dont la réussite lui revient.

J'exprime ma profonde reconnaissance à mon co-promoteur Monsieur MOUMEN Hamouma, pour avoir dirigé mes travaux, j'exprime mes vifs remerciements pour son encadrement, ses conseils, son soutien tout au long de ce travail, et pour m'avoir encouragé dans les moments les plus difficiles.

Je tiens à exprimer ma profonde gratitude à M. Abdelkamel TARI, Chef du département d'informatique de l'Université de Bejaia et le responsable de l'Ecole Doctorale, ainsi que nos enseignants qui ont veillé au succès de notre promotion.

Je remercie les membres du jury de nous honorer de leurs présences et de leurs savoirs pour juger ce travail.

Un très grand merci à Mr M. Naoun maître assistant à l'université de Batna pour ses conseils, son accueil chaleureux, son soutien et son attention.

Je remercie également mes amis de la promotion et spécialement : Pedrou, Ammar, khaled Mouh, Fateh, Nabil et Bilal.

Je remercie mes parents pour avoir cru en moi et m'avoir permis d'arriver jusqu'ici. Merci pour les racines et les ailes...

Finalement, je remercie tous ceux qui dans l'ombre ont participé au bon déroulement de ce travail et particulièrement Dr SLIMANI Sami, et mes amis Lamine, Mansour, Foufa et Adel…

Dédicaces

A la mémoire de mes très chers regrettés grands-parents, Je dédie ce travail avec beaucoup d'amour et une immense gratitude :

A mes parents, les deux personnes les plus proches de mon cœur.

A mon cher frère Adlen.

A mes chères sœurs et à mon beau frère Farouk..

A Mes tantes, mes oncles et leurs enfants.

À tous ceux qui m'ont aidé à réaliser ce mémoire.

A mes Amis Hakou, Youcef, Sofiane, Nabil, Foufa, Fateh, Badreddine, Adel, Lamine, Mansour, Mouh et Farid.

Résumé

Fischer, Lynch et Paterson ont prouvé que les problèmes d'accord n'ont pas de solutions déterministes dans un système asynchrone si, au plus un seul processus est crashé. En effet, le crash des processus est un type de fautes inclus dans un autre type plus général appelé la panne byzantine où les processus se comportent d'une manière aléatoire; la même impossibilité de FLP est appliquée au consensus byzantin dans un système asynchrone. Une des solutions pour surmonter ce résultat d'impossibilité est l'utilisation des hypothèses temporelles. Dans ce travail nous présentons deux protocoles basés sur ce modèle ; le premier est un protocole indépendant du temps physique du système, il est basé sur une propriété comportementale du modèle d'échange de messages pour résoudre le consensus byzantin avec peu de canaux winnings (2t-◊WC). Le deuxième protocole se base sur une combinaison d'hypothèses, celle qui utilise le temps physique et celle qui ne l'utilise pas. Ce protocole montre que les deux types d'hypothèses peuvent être combinés pour obtenir un protocole hybride du consensus byzantin profitant du mieux de tous les deux modèles (2t-[◊WC, ◊bisource]).

Mots clé : systèmes distribués, fautes byzantines, algorithmes distribués, canal gagnant, problèmes d'accords, consensus byzantin.

Abstract

Fischer, Lynch and Paterson in their seminal paper proved that it is impossible to solve the agreement problem in a deterministic manner of an asynchronous distributed system if even a single process can fail. This class of faults is included in a more general class called byzantine failure where processes behave in a random way. So the same impossibility of FLP is applied to the byzantine consensus in an asynchronous system, one of the solutions to overcome this impossibility result is the use of extended models that provide additional timing assumption. In this work we propose two protocols, the first one is a time free protocol that solve the byzantine consensus with few winning channels (2t -◊WC), while the second protocol is the combination of a time-free assumption on the message pattern with a synchrony assumption on process speed and message delay. It shows that both types of assumptions can be combined to obtain an hybrid byzantine consensus protocol benefiting from the best of both worlds (2t-[◊WC, ◊bisource]).

Keywords: distributed systems, byzantine faults, distributed algorithms, winning channel, agreements problems, and byzantine consensus.

Table des matières

Table des matières

Table des matières

Liste des figures

Introduction Générale

Historiquement, le développement des systèmes répartis est principalement lié à plusieurs besoins, tels que la communication entre entités géographiquement distantes, l'accélération des calculs suite à l'augmentation des ressources, la fiabilisation des systèmes dû à la redondance des moyens de calcul. Ainsi, tout ordinateur défaillant peut être remplacé par n'importe quel autre ordinateur opérationnel. Pour effectuer de tels remplacements, il convient de disposer d'algorithmes qui résistent aux défaillances et qui sont capables de maintenir la spécification générale du système même si plusieurs ordinateurs ont un comportement incohérent. Il existe plusieurs types de défaillances, dont la plus difficile à traiter est la défaillance byzantine, qui correspond à l'intrusion d'un pirate dans le système et à l'exécution d'un code totalement différent et incontrôlable sur les machines concernées.

L'exemple idéal de fautes byzantines est présenté dans l'article M. PEASE, R. SHOSTAK et L. LAMPORT [PSL80], dans lequel, des généraux de l'armée Byzantine qui assiégeaient une cité ennemie, communicant à l'aide de messagers doivent coordonner leurs mouvements afin d'assurer la victoire. Au cas où un des messagers est mal interprété ou délibérément faussé par l'un de ces généraux traîtres, la victoire serait alors compromise. Pour palier à ce problème, il suffit de trouver un algorithme pour que les généraux loyaux arrivent à se mettre d'accord sur un plan de bataille. Ce problème complexe, portant le nom de consensus byzantin, a dû être résolu par une méthode de communication impliquant un haut niveau de redondance des messages.

Ce dernier est un des problèmes dits d'accords. Cette notion d'accord définit une vaste classe de problèmes regroupant, par exemple, les problèmes de synchronisation d'horloges, de diffusion fiable de messages, de validation dans une base de données distribuées, de gestion d'un groupe et de l'élection d'un leader ...

Le consensus byzantin n'a pas de solution déterministe dans un système asynchrone [FLP85], cette impossibilité peut être surmontée par l'utilisation d'oracles, parmi lesquels les détecteurs de défaillances introduits par Chandra et Toueg [CT96] dans un modèle de fautes par crash. Melkhi et Reiter [MR97] sont les premiers à avoir étendu ce type de détecteur vers le modèle byzantin, ensuite plusieurs autres détecteurs de fautes byzantines ont été étudiés dans la littérature présentée [DS97,BHRT03, KMM03, FMR05].

Ces détecteurs de défaillances byzantines font une abstraction des propriétés temporelles du système, alors qu'il est possible de contourner l'impossibilité de FLP seulement par des hypothèses temporelles. Une de ces hypothèses est de supposer que le système est partiellement synchrone [DLS88]. Un système partiellement synchrone signifie que les n^2 liens sont partiellement synchrones (où n est le nombre de processus). D'autres solutions plus faibles [ADFT06,MMT07] assurent le consensus byzantin avec un nombre de liens inéluctablement synchrones inférieurs de $n^{2,}$, respectivement 2n et 4t (avec $n > 3t$).

L'objectif général de ce mémoire est d'apporter des réponses à la question suivante : comment construire un consensus fiable, tolérant les fautes byzantines avec des hypothèses temporelles les plus faibles possibles? Pour cela, nous présentons un aperçu sur les solutions existantes, les conditions nécessaires et suffisantes pour assurer le consensus byzantin dans un système asynchrone et à la fin nous proposons deux protocoles.

Notre travail est présenté dans ce mémoire sous quatre chapitres : Dans le premier chapitre, nous présentons le modèle des systèmes distribués et les modèles temporels avec tous les modes de défaillances auxquels ils sont assujettis. Le chapitre 2 définit les problèmes d'accords et leurs réductibilités, de même que le résultat d'impossibilité de FLP et les solutions pour la contourner. Le troisième chapitre décrit les protocoles résolvant le consensus byzantin dans un système asynchrone. Dans le quatrième chapitre, nous proposons deux implémentations du consensus byzantin : le premier protocole est un protocole non temporel (2t-◊WC) et le deuxième protocole est un protocole hybride 2t-[◊WC, bi-source]. Finalement, nous concluons notre travail et nous donnons les perspectives de recherche future.

Chapitre 1

Modèles de système distribué

1 Introduction

En décrivant tout système, il est nécessaire de présenter clairement les hypothèses qui ont été prises en considération. En d'autres termes, on définit le modèle du système. Dans notre travail, nous nous sommes concentrés sur les systèmes distribués où la communication est assurée par échange de message (nous ne nous intéressons pas au système distribué à mémoire partagée), les propriétés qui caractérisent ce type de système sont les processus et les canaux de communication

2 Processus

Un système distribué est un ensemble de processus qui s'échangent des messages via les canaux de communication. Un processus peut être correct ou incorrect. Un processus est dit correct s'il se comporte selon sa spécification alors qu'un processus incorrect dépend de son comportement lorsque il est défaillant. Les fautes peuvent être classifiées comme suit (voir Figure 1.2) :

2.1 Panne franche (crash)

Une faute par arrêt de processus se traduit par l'arrêt soudain du processus, tous ses calculs sont stoppés et il ne reçoit ni n'envoie, plus, de message. L'arrêt est définitif, le processus ne participe plus au fonctionnement du système.

2.2 Panne d'omission

Un processus commet une faute par omission lorsqu'il omet d'effectuer une action. Par la suite on prend en considération l'omission de message en envoi et /ou en réception.

Ces fautes surviennent lorsque les tampons de réception ou d'émission sont pleins. Contrairement au crash, les fautes par omission n'ont pas de caractère définitif, le modèle de fautes par omission est inclus dans le modèle de faute par arrêt.

2.3 Panne de performance

Ces fautes se caractérisent par un non respect des délais de la part des processus fautifs. Il en résulte une dégradation de la qualité de service du processus, par exemple un processus fautif peut traiter les messages qui lui arrivent au bout d'un temps infini ce qui donne l'impression aux autres processus qu'il exhibe une omission.

2.4 Panne byzantine

On peut définir les fautes byzantines en termes de déviation d'un processus de l'algorithme qu'il exécute, par exemple il envoie un message qui n'est pas prévu dans l'algorithme ou bien lorsque l'algorithme a besoin d'une diffusion alors il diffuse un message M à un sous ensemble et un autre message M' aux restes des processus.

Cependant on ne peut pas détecter les fautes qui ne se manifestent qu'à l'état interne du processus byzantin, parce que le comportement externe d'un processus peut être inconsistant avec son état interne par exemple. KIM POTTER KIHLSTROM et al [KMM03] ont proposé une classification des fautes byzantines en termes de fautes détectables ou non (Figure 1.1) :

2.4.1 Fautes byzantines non détectables : ce sont les fautes non observables par les processus se basant sur les messages reçus, et les fautes qui ne sont pas diagnostiquables.

Par exemple, si un processus diffuse un message alors que son état interne utilise un autre message, c'est un comportement non observable. De même que si un processus byzantin envoie un message qui a été envoyé par un autre processus mais avec une signature erronée alors en absence d'informations supplémentaires ces fautes ne sont pas diagnostiquables.

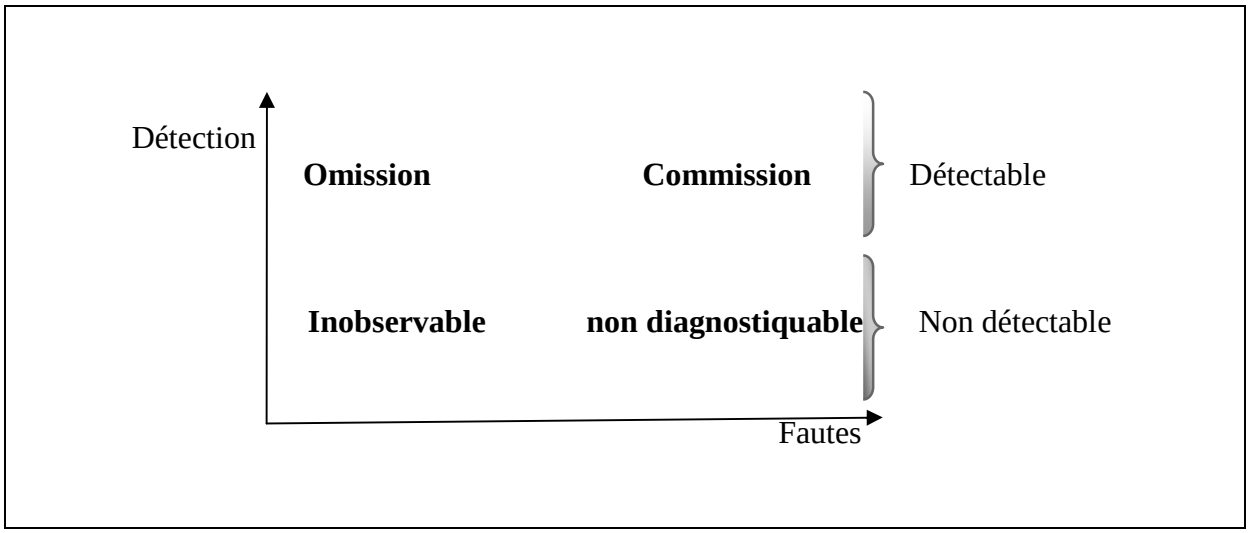

Figure 1.1 : Classification des fautes byzantines

2.4.2 Fautes byzantines détectables : En référence à la déviation évidente dans le comportement externe d'un processus, une déviation détectable d'un algorithme est définie en termes de messages envoyés par un processus pendant une exécution particulière de cet algorithme.

Il existe deux types de fautes, les fautes d'omission et les fautes de commission :

- Faute d'omission [2.2 panne d'omission].
- Faute de commission : Aura lieu lorsque un processus envoie un message qui ne doit pas être envoyé pendant l'exécution d'un algorithme, par exemple si un processus diffuse un message mutant c'est à dire deux messages différents pour la même diffusion.

Dans n'importe quelle exécution d'un algorithme distribué, il faut s'assurer que:

- Le processus doit envoyer le même message vers tous les processus.
- Le message envoyé doit être conforme à l'algorithme exécuté

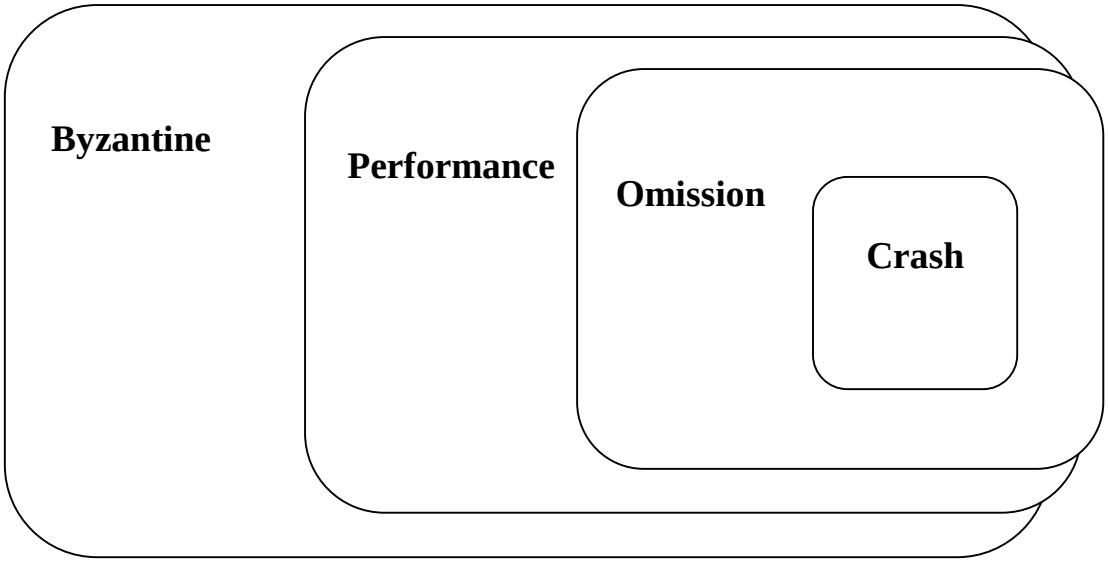

Figure 1.2: Classification des fautes

3 Canaux de communication

Parmi les propriétés des canaux de communication nous retrouvons les propriétés structurelles et les propriétés comportementales

3.1 Propriétés structurelles

Caractérisent la topologie du maillage des liaisons logiques de communication appelée graphe de communication.

- Maillage en anneau : Un processus ne connaît directement que son suivant (anneau unidirectionnel) ou bien ces deux voisins immédiats (anneau bidirectionnel).
- Maillage en étoile : Il existe un processus particulier Pi=1 ne connaît que P1 et P1 connaît tous les autres sites.

- Maillage en arbre : P1 ne connaît que ses fils (P1 est la racine de l'arbre); Si Pi = 1 a des fils alors Pi = 1 ne connaît que ses fils et son père ; Si Pi = 1 n'a pas de fils alors : Pi = 1 ne connaît que son père (processus feuille ou terminal) ;
- maillage complet : type Internet => tous les processus se connaissent.

La connaissance de cette topologie est fondamentale lorsqu'on définit des algorithmes distribués. On peut être amené à rajouter une "couche" sur un réseau X pour simuler un maillage Y. Ainsi, l'anneau ne peut être que virtuel => alors il ne faut pas confondre entre un réseau physique et un maillage logique.

3.2 Propriétés comportementales

Elles sont diverses, les plus fréquemment rencontrées sont :

- H1 : la transmission sur la voie se fait sans duplication de message ;
- H2 : la transmission se fait sans altération de message ;
- H3 : pour tout couple de processus, l'ordre de réception des messages est identique à l'ordre d'émission (pas de déséquencement) ;
- H4 : le délai d'acheminement des messages est fini : tout message envoyé est reçu au bout d'un temps fini ;
- H5 : le délai d'acheminement est borné : si un message n'est pas reçu après X secondes, il est perdu.
- Dans le cas où le protocole de communication garantit les hypothèses H1, H2, H3 et H4, on dit que les canaux de communication sont fiables et FIFO.

3.3 Les fautes des canaux de communication

Lorsqu'un canal de communication n'assure pas une des trois propriétés H1, H2 ou H3 on dit que le canal est défaillant. Par exemple, la négation de la propriété H3 donne un canal défaillant qui altère les messages.

Remarque

Les réseaux de communication peuvent être statiques, c'est à dire que l'ensemble Π, de processus, est statique et les canaux de communication sont connus initialement. Au contraire des réseaux statiques, les réseaux dynamiques sont des réseaux dont les processus ou les canaux de communication peuvent être ajoutés ou supprimés à tout moment [MRTPA05].

4 Modèles temporels de communication par messages

On présente dans cette section un état de l'art concernant les modèles de communication par message. Ces modèles sont caractérisés par les hypothèses sur les délais de transmission des messages et les vitesses des processeurs. Ils se déclinent du modèle synchrone au modèle asynchrone en une multitude de modèles dits partiellement synchrones.

4.1 Le modèle synchrone

Le modèle synchrone est caractérisé par les hypothèses temporelles les plus fortes que l'on puisse faire sur un modèle de communication par message. Un tel modèle est caractérisé par l'existence de bornes sur les délais de transmission des messages. Ces bornes sont connues et fixées. Les systèmes exhibant un comportement synchrone ont un taux de couverture très faible [DLS88], de ce fait, les modèles synchrones sont, en pratique, très peu utilisés.

4.2 Le modèle partiellement synchrone

La notion partiellement synchrone introduite par Dwork, Lynch et Stockmeyer dans [DLS88], se comporte de manière asynchrone pendant un moment, puis, il se stabilise et commence à se comporter d'une manière synchrone.

Donc l'idée est d'affaiblir le modèle asynchrone par des propriétés temporelles. La terminaison des algorithmes basés sur ce modèle est garantie une fois que ces propriétés sont satisfaites

Dwork et al [DLS88] ont présenté deux modèles partiellement synchrones, chacun prolonge le modèle asynchrone avec des propriétés temporelles :

1. Le premier modèle considère que, quel que soit l'exécution considérée, il existe des bornes temporelles sur les délais de transmission de messages et sur les temps d'exécution des tâches par les processus. Ces bornes ne sont pas connues mais elles existent.

2. Le deuxième modèle est caractérisé par le fait qu'il existe un moment (non connu) appelé temps global de stabilisation (GST), à partir duquel des bornes temporelles, sur les délais de communication et sur les temps d'exécution des tâches par les processus, existent et sont connues. Ce modèle a été utilisé par Dwork et al [DLS88] pour résoudre des problèmes d'accord dans les systèmes distribués.

Chandra et Toueg ont proposé un troisième modèle qui est semblable, mais plus faible [CT96] :

3. Dans ce troisième modèle, il existe un moment (non connu) appelé temps global de stabilisation, à partir duquel des bornes temporelles existent et ne sont pas connues. Il diffère du deuxième par une méconnaissance des bornes sur les délais de communication et sur les temps d'exécution des tâches par les processus.

Un quatrième modèle a été proposé par F. Cristian et C. Fetzer [CF99]

4. Le quatrième modèle se caractérise principalement par l'existence d'une horloge physique et par des services temporisés. Ces hypothèses ne suffisent pas pour la résolution de certains problèmes spécifiques aux systèmes distribués. De plus, Cristian et Fetzer ont introduit une hypothèse de progrès qui se traduit par l'existence d'une période de stabilité d'une longueur minimum après une période d'instabilité.

4.3 Le modèle asynchrone

Nous finalisons cette présentation des modèles de communication par celui dit asynchrone [PSL80]. Il se caractérise par une absence d'hypothèse temporelle. Contrairement au modèle partiellement synchrone, Il n'utilise pas d'horloge physique. Ce modèle a été largement utilisé par les scientifiques pour obtenir des résultats fondamentaux.

5 Conclusion

Dans notre travail nous nous intéressons aux systèmes distribués dont la communication est via l'échange de messages. Les types de fautes à prendre en considération sont les fautes byzantines. La transmission des messages se fait sans duplication ni altération de message ce qui donne un lien fiable. En ce qui concerne la synchronisation, nous nous basons sur le modèle asynchrone.

Chapitre 2

Problèmes d'accord

1 Introduction

La gestion d'un système distribué soulève plusieurs problèmes, tels que: le problème d'élection d'un leader, la diffusion atomique, la gestion de groupe et bien sûr le consensus qui est l'assise pour la résolution des autres problèmes d'accord.

2 Problèmes d'accord

Dans ce chapitre, nous proposons l'identification de quelques problèmes d'accord, ensuite nous exhibons le problème de base qui est le problème du consensus.

2.1 Validation atomique (Atomic Commitment)

Dans les bases de données distribuées, une des tâches critiques est d'assurer la terminaison d'une transaction d'une manière consistante. Les sites où leur bases de données sont mises à jour par cette transaction doivent se mettre d'accord sur le résultat de cette transaction soit par validation (commit) ou bien par annulation (abort).

Chaque processus doit donner son opinion sur la transaction. Un processus vote pour COMMIT si les traitements locaux sur cette transaction sont achevés avec succès, sinon il vote ABORT c'est à dire l'annulation de cette transaction.

La transaction est acceptée si tous les processus ont accepté cette transaction, sinon elle est annulée. (Tous ou rien). Les conditions qu'il faut satisfaire et qui assurent la correction d'une AC sont [BERN01] :

- Accord : deux processus ne peuvent décider différemment ;
- Validité :
  a) Si un processus initialement vote non, alors ABORT est la seule solution possible.
  b) Si tous les processus vote oui alors COMMIT est la seule décision possible
- Terminaison : nous avons deux cas possibles
  a) terminaison faible : si aucune défaillance ne survient tous les processus doivent décider.
  b) terminaison non bloquante : inéluctablement, tout les processus corrects décident sur une valeur.
- Intégrité : Un processus décide au plus une fois

Une des variantes du protocole AC est la validation atomique non bloquante (NBAC pour Non-Blocking Atomic Commitment), cette dernière assure la terminaison de AC même en présence de défaillance des processus.

Le principe de la validation atomique est comme suit : le processus coordinateur diffuse une demande de réalisation d'action à tous les processus. Chaque processus décide s'il sera capable d'effectuer cette transaction ou non et il répond au coordinateur. Une fois que le coordinateur reçoit toutes les décisions des processus, il diffuse la décision finale qui sera respectée par tous les processus. Si tous les processus avaient répondu « oui », alors la décision est de valider et chaque processus exécutera l'action .Si un processus au moins avait répondu « non », alors la décision est d'annuler et aucun processus n'exécutera la transaction.

La transaction doit être, en effet, exécuté par tout ou rien

Pour que la validation fonctionne (plus ou moins bien) dans un contexte de fautes, deux grands types d'algorithmes [SBCM93] :

- Validation à 2 phases : 2 phases commit (2PC)
- Validation à 3 phases : 3 phases commit (3PC)

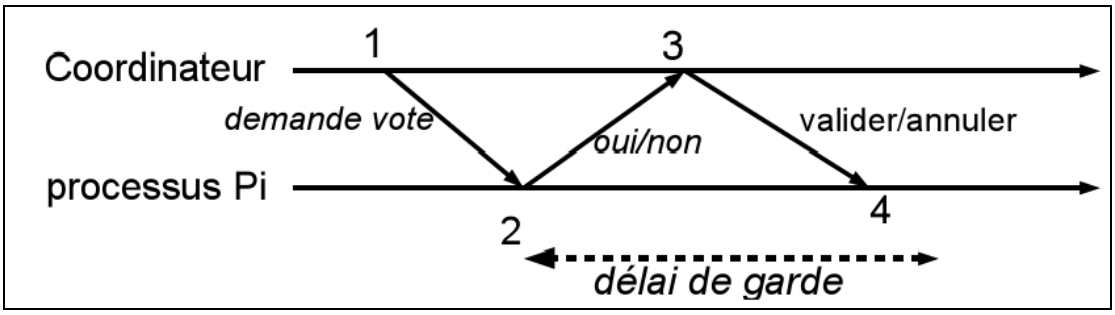

Figure 2.1 : validation à deux phases 2PC

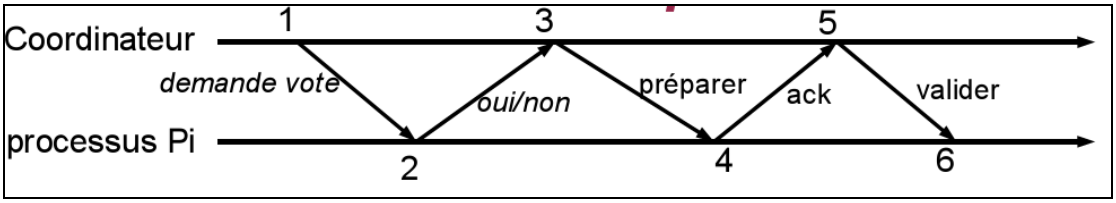

Figure 2.2 : validation à trois phases 3PC

De nombreux travaux ont été élaborés dans cet axe et le résultat le plus important est l'impossibilité de Charon-bost et S. Toueg [CT03] qui ont montré que NBAC ne peut pas être réduit au consensus uniforme. Ensuite Guerraoui dans [G95] a affaibli NBAC et il a montré que nous pouvons réduire NBWAC au consensus uniforme, NBWAC qui est le NBAC faible (Non-Blocking Weak Atomic) qui ressemble au NBAC sauf qu'il tolère les processus défaillants qui ne sont pas suspectés.

2.2 LA diffusion atomique

Avant de présenter la diffusion atomique, il est impératif de présenter la diffusion fiable.

2.2 .1 La diffusion fiable

La diffusion fiable assure que tous les processus délivrent un message qui a été précédemment diffusé par un processus qui peut être en panne, il est difficile d'assurer quand l'expéditeur du message s'effondre en envoyant le message : dans ce cas, il est possible que seulement un sous-ensemble de tous les processus reçoit le message. Plus formellement,

l'émission fiable est définie par deux primitives la R-diffusion (m) et R-délivre(m) où le m est un message. L'émission fiable est spécifiée comme suit : [BHG87]

- Terminaison : si un processus correct diffuse un message m, alors tous les processus corrects délivrent m ;
- Validité : si un processus délivre un message m, alors m a été diffusé par au moins un processus ;
- Intégrité : un message m est délivré au plus une fois.
- Accord : Si un processus correct délivre un message m, alors tous les processus corrects délivrent m ;

2.2.2 La diffusion atomique

La diffusion atomique est une extension de la diffusion fiable: en plus de l'assurance que tous les processus reçoivent les messages, elle assure aussi que les processus voient les messages dans le même ordre. Plus formellement, la diffusion atomique est définie par deux primitives A-diffuse (m) et A-Délivre (m) où le m est un message. La diffusion atomique est spécifiée avec les propriétés de diffusion fiable plus la propriété suivante [BHG87] :

- Ordre total : si deux processus corrects p et q délivrent deux messages m et m', alors p délivre m avant m' si et seulement si q délivre m avant m'.

2.3 L'élection d'un leader

Dans un environnement distribué, la plupart des applications exigent souvent qu'une entité agisse comme un contrôleur central pour coordonner l'exécution d'une tâche particulière. Le besoin d'un coordinateur résulte du désir de simplifier la conception du protocole dans le cas d'un problème complexe et dans d'autres cas, la présence d'un coordinateur est exigé par la nature du problème lui même. Ce problème est appelé ***l'élection d'un leader***. Formellement, on définit ce problème 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 un leader.

Récemment A. Fernandez et al [FJR06] ont proposé une solution au problème d'élection mais avec des suppositions faibles sur la connaissance initiale des processus, ainsi que sur la synchronisation. Un autre résultat est de K.Aguilera et al [ADFT04] dont la solution est basée sur un nombre minimal de liens inéluctablement synchrones ; une autre solution qui exploite les propriétés comportementales des messages a été présentée par Mostefaoui et al [MRT06].

2.4 La Gestion de groupe

Un autre problème d'accord, est le problème de gestion de groupes dans un système distribué. La tâche d'un service de gestion de groupe est de maintenir une liste de processus corrects. Cette liste peut changer par l'ajout des membres et le départ des anciens membres ou bien la défaillance des processus. Chaque processus a une vue de cette liste, et une fois qu'un changement a lieu, le service de gestion de groupes annonce ce changement aux processus en installant une nouvelle vue. Ce problème peut être défini formellement par les propriétés suivantes [HRCWS03] :

- Accord : Si nous avons deux processus corrects p et q, alors la vue Vi contient les mêmes membres pour les deux processus
- L'inclusion automatique : Si un processus correct p installe une vue Vi, donc Vi inclut p
- La validité : Si un processus correct p installe une vue Vi, donc tous les membres corrects de Vi installeront Vi.
- L'intégrité : Si la vue Vi inclut p mais la vue suivante Vi+1 exclut p, alors p a été suspecté par au moins un processus correct

Il y a d'autres services dans la gestion de groupe, c'est le service de communication et le service de synchronisation des vues; de nombreux travaux ont été élaborés sur le domaine de la communication dans un groupe et surtout en présence de fautes byzantines en d'autres termes les intrus [HRCWS03, AAH00, KMM01].

2.5 Le consensus

Le problème du consensus est un problème fondamental dans les systèmes distribués, car il représente une brique de base permettant de construire d'autres protocoles d'accord. L'objectif du consensus est de permettre à un ensemble de processus, chacun avec sa propre valeur initiale, de décider d'une manière irrévocable sur une de ces valeurs initiales. Le consensus peut être binaire ou bien multivalué

Un algorithme du consensus binaire a comme objectif la réalisation d'un consensus sur une valeur binaire $v \in \{0, 1\}$. Chaque processus propose sa valeur initiale et décide sur une valeur v. Le problème peut être formellement défini en termes de trois propriétés :

- Validité : si un processus décide une valeur v, alors v a été proposée par au moins un processus ;
- Accord : deux processus ne peuvent décider différemment.
- Terminaison : inéluctablement, tout processus doit décider ;

Les deux premières propriétés sont des propriétés de ***sûreté***, c'est-à-dire, les propriétés qui disent qu'aucune mauvaise chose ne peut arriver, tandis que la dernière est une propriété de ***vivacité***, c'est-à-dire, une propriété qui expose les bonnes choses qui doivent arriver.

Le consensus multivalué est apparemment semblable au consensus binaire, sauf que le jeu de valeurs possède la taille arbitraire, c'est-à-dire, $v \in V$ et $|V| > 2$. Les algorithmes du consensus ont été étudiés dans la littérature en employant plusieurs propriétés de Validité, tandis que l'Accord et la Terminaison sont restés essentiellement les mêmes. Certaines publications utilisent la propriété de Validité suivante [DLS88, KMM03, MR97] :

Validité 1 : si tous les processus corrects proposent la même valeur v, alors n'importe quel processus correct qui décide, décide v.

D'autres emploient la propriété de validité suivante [BHRT03, DS97] :

Validité 2 : si un processus correct décide v, alors v est la valeur initiale de certains processus.

Ces deux propriétés sont peu faibles. La validité 1 ne relate rien de ce qui est décidé quand les processus corrects ne proposent pas la même valeur v, tandis que la Validité 2 ne s'exprime pas sur la valeur décidée.

Récemment, est apparue une définition qui donne plus de détail de ce qui est décidé [CNV06]. La définition a trois propriétés de Validité :

- Validité 1 : si tous les processus corrects proposent la même valeur v, donc n'importe quel processus correct qui décide, décide v.
- Validité 2 : si un processus correct décide v, alors v est la valeur initiale de certains processus ou bien v =⊥.
- Validité 3 : si une valeur v a été proposée seulement par des processus corrompus, alors aucun processus correct ne décide v.

Les deux premières sont essentiellement les propriétés de Validité déjà présentées, sauf que la Validité 2 permet la décision d'une valeur $\perp \notin V$.

La troisième propriété est inspirée d'après la définition originale dans le contexte "des Généraux Byzantins", la métaphore employée dans la publication classique par Lamport et d'autres. [LSP82].

2.5.1 Les instances du problème du consensus

Le problème du consensus est défini formellement par les trois propriétés (terminaison, validité et accord) et si on reformule une de ces propriétés, on obtient une nouvelle instance du consensus, parmi ces instances :

2.5.1.1 Le consensus probabiliste

Le consensus probabiliste a été proposé et résolu par Ben-Or [B83]. Le consensus probabiliste diffère du consensus par la propriété de terminaison. Cette propriété est informellement définie comme suit :

La probabilité que tous les processus corrects décident est égale à 1. Cette propriété de terminaison est aussi noté R-terminaison et s'énonce formellement comme suit :

- R-terminaison: Tous les processus corrects décident avec une probabilité égale à 1.

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

2.5.1.2 Le consensus uniforme

Dans la définition du consensus, les processus défaillants peuvent décider différemment des processus corrects. Cette définition ne convient pas dans certaines applications qui demandent un niveau de sûreté très élevé. Pour interdire ce défaut, le consensus uniforme [HT93, CS00, RR03, RR04] a été défini et diffère du consensus de base par la propriété d'accord :

- **Accord uniforme** : Si un processus décide une valeur v, alors tous les processus décident v. (ou bien si un processus décide une valeur v1, et de même si un autre processus décide une valeur v2 alors v1 = v2)

Le problème du consensus uniforme est plus difficile à résoudre que le consensus simple, et il n'a aucun sens dans les systèmes sujets à des fautes arbitraires.

2.5.1.3 Le consensus ensembliste

Le k-consensus ($k \in [1; n]$) est plus connu sous sa dénomination anglaise, à savoir le "k-set agreement". Le k-consensus consiste en un accord sur au plus k valeurs différentes. Le problème du consensus correspond au 1-consensus. La définition de ce problème généralise celle du consensus par la propriété d'accord, pour $k \in [1; n]$:

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

Ce problème est introduit par S. Chaudhuri [C90] pour montrer que plus la propriété d'accord est faible (plus k est grand) plus la tolérance aux défaillances est forte. Plus précisément, le k-consensus a une solution déterministe si et seulement si $k > f$ (f nombre de processus byzantins).

2.5.1.4 Le consensus strict

Malkhi et Reiter [MR17] considèrent le problème du consensus strict comme une variante du consensus traditionnel. Dans le consensus strict, ils ont substitué la propriété de validité par la propriété de validité stricte. Cette dernière assure que la valeur décidée est celle d'un processus correct si et seulement si tous les processus corrects ont la même valeur initiale.

L'intérêt d'une telle spécification est de garantir que la décision est une valeur initiale d'un processus correct, moyennant une condition sur les valeurs initiales des processus corrects. Dans le modèle byzantin, on considère la validité stricte plutôt que la validité traditionnelle, car cette dernière a l'inconvénient de ne pas exclure que la décision est la valeur initiale d'un processus byzantin.

Toutefois, le consensus strict est trop restrictif pour servir à la résolution d'autres protocoles d'agreement tel que l'ordre total.

2.5.1.5 Le Consensus vectoriel

Afin de relâcher la restriction imposée par le consensus strict, Doudou et al ont introduit le consensus vectoriel [DS97], ce dernier permet à un ensemble de processus corrects, chacun ayant sa propre valeur initiale, de décider sur un vecteur qui contient au moins f +1 valeurs initiales de processus corrects, et ceci en dépit de la présence de processus byzantins. D'une manière formelle, le consensus vectoriel est défini par les propriétés traditionnelles d'accord et de terminaison, et une nouvelle propriété de ***validité vectorielle***

- **Validité vectorielle** : les processus décident sur un vecteur *vect* de taille n tel que : (1) pour tout i et tout pi correct, *vect*[i] est soit valeur initiale de pi, soit *null*, et (2) au moins f +1 éléments de *vect* sont des valeurs initiales de processus corrects.

3 Réduction des problèmes d'accord

Une des techniques pour résoudre un problème d'accord donné, est de le réduire vers un autre problème d'accord équivalent dont la résolution est possible ou bien facile. Dans cette section, nous allons aborder la réduction des problèmes d'accord selon la définition suivante :

Définition : Pour un modèle de calcul M et deux problèmes P et P', on dit que P' est réductible en P si le protocole qui résolve P' dans le modèle M peut être dérivé de n'importe quel protocole qui résolve le problème P dans le même modèle de calcul M

Remarque : Il est important de remarquer que la réduction est relative à un modèle donné. En effet, il se peut qu'un problème P' est réductible à P dans un modèle M, mais pas dans un autre modèle M'. Si on a deux problèmes P et P' dont P' est réductible à P dans le modèle M', et le modèle M' est plus faible que le modèle M. Clairement, on obtient le diagramme suivant:

	(1)	
P peut être résolu dans M'	$\Rightarrow$	P' peut être résolu dans M'
(3) $\not\Uparrow$		$\Downarrow$ (2)
P peut être résolu dans M		P' peut être résolu dans M

(1) La première implication est valide car P' est réductible à P dans M'.

(2) N'importe quel problème qui peut être résolu dans M' peut également être résolu en M.

(3) Pas d'implication car M et M' ne sont pas équivalents.

Ainsi, ce diagramme prouve que la réductibilité de P' à P en M ne peut pas être impliquée.

3.1 Quelques résultats sur la réduction des problèmes d'accord

Le consensus et le consensus uniforme sont équivalents dans le modèle FLP avec des détecteurs de défaillances non fiables [G95].

Le consensus et la diffusion atomique sont équivalents indépendamment du modèle de calcul [CT96]. Plus précisément, dans tous les modèles permettant une résolution de la diffusion fiable.

La validation atomique non bloquante est plus difficile à résoudre que le consensus. Ce problème est impossible à résoudre dans le modèle asynchrone étendu avec les détecteurs de défaillances non fiables. Cependant, une version affaiblie de ce problème est réductible au consensus (NBWAC validation atomique non bloquante faible) [G95].

De ces résultats, nous constatons que le consensus est le problème d'accord le plus étudié et de plus il est la brique de base pour bâtir d'autres problèmes d'accord tels que la validation atomique, la diffusion atomique et le consensus uniforme (voir Figure 2.3). Dans ce mémoire, nous nous intéressons aux consensus dans un environnement asynchrone avec un modèle de défaillances byzantines.

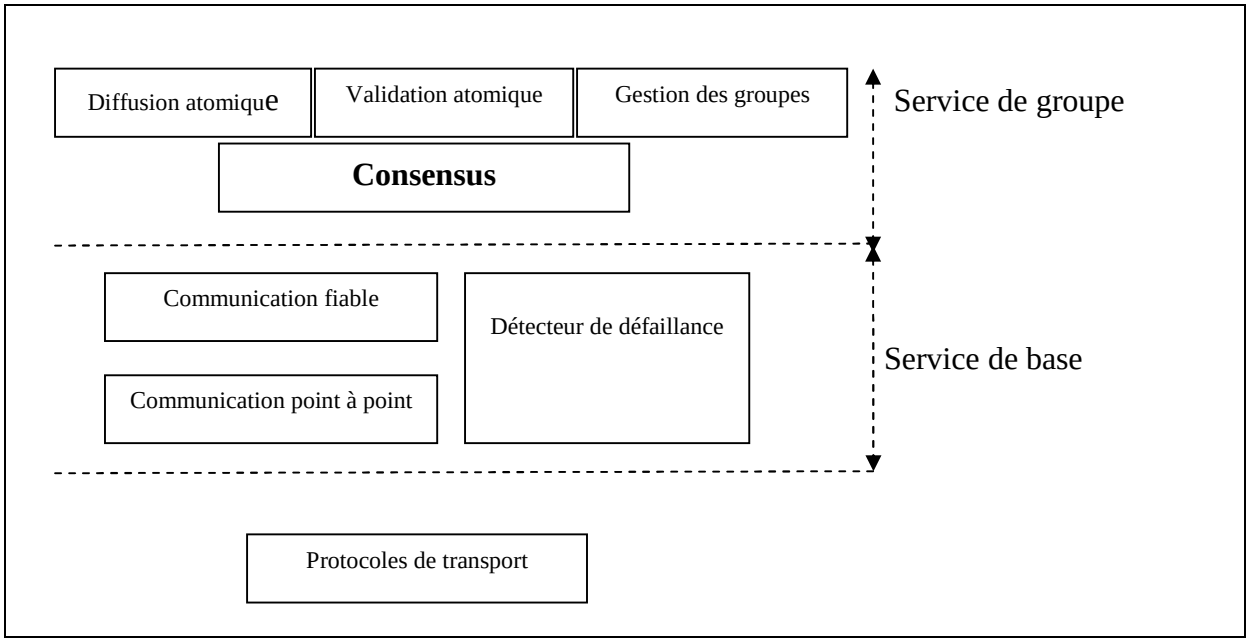

Figure 2.3 : Relation entre le consensus et les autres problèmes d'accord

4 Résultat d'impossibilité de FLP

L'article le plus cité au sujet du consensus est probablement celui qui prouve l'impossibilité de résoudre le consensus d'une manière déterministe dans un système asynchrone en présence d'un seul processus crashé [FLP85]. Ce résultat souvent appelé le résultat d'impossibilité de FLP. En effet, dans un système asynchrone il est impossible de différencier entre un processus correct mais lent (canaux de communication lents) d'un autre qui est crashé. Le nouveau problème à résoudre est donc: quelles sont les modifications minimales à apporter à la définition du système réparti asynchrone ou au problème du consensus pour que ce dernier soit solvable ?

Pour résoudre le consensus, l'algorithme doit contourner ce résultat d'impossibilité de FLP

Le mot, contourner, est tout à fait imprécis; ainsi il est important d'expliquer sa signification. L'idée est d'appliquer une légère modification sur le modèle du système ou bien sur la définition du problème considéré, pour rendre le problème solvable.

Il existe différentes techniques pour contourner le résultat d'impossibilité de FLP, parmi ces techniques nous trouvons :

- Augmentation du système par un oracle (détecteur de défaillance).
- Modification de la définition du problème (affaiblir la définition du problème).
- L'ajout des suppositions temporelles.
- Le sacrifice du déterminisme.

5 Contourner le résultat d'impossibilité de FLP

5.1 Le sacrifice du déterminisme :

Le résultat d'impossibilité de FLP s'applique aux algorithmes déterministes, ainsi une solution pour l'éviter est l'utilisation de la randomisation pour concevoir des algorithmes probabilistes [B83, R83, TOU84,]. Plus spécifiquement, l'idée est de substituer une des propriétés qui définissent le consensus par une propriété semblable qui est satisfaite seulement par une certaine probabilité. Pour les raisons mentionnées ci-dessus, presque tous les algorithmes choisissent de modifier la propriété de terminaison, qui devient:

- R-terminaison:Tous les processus corrects décident avec une probabilité égale à 1.

Les algorithmes du consensus randomisés ont été étudiés depuis les publications de Ben-Or et de Rabin [B83, R83]. Pratiquement tous les algorithmes du consensus randomisés sont basés sur une opération aléatoire, en utilisant un générateur des nombres aléatoires, qui renvoie les valeurs 0 ou 1 avec une probabilité égale.

Ces algorithmes peuvent être divisés en deux classes selon la façon dont l'opération de génération du nombre aléatoire: il y a ceux qui emploient un générateur local dans chaque processus (commençant par le travail de Ben-Or's [B83]), et ceux basés sur un générateur partagé qui donne les mêmes valeurs à tous les processus (lancés avec le travail de Rabin's [R83]).

5.2 L'ajout des suppositions temporelles

La deuxième solution pour contourner l'impossibilité de FLP est l'ajout d'hypothèses temporelles. Dans le modèle FLP, le système est asynchrone, donc les auteurs [DLS88] ont affaibli cette propriété par la définition d'un nouveau modèle intermédiaire entre le modèle synchrone et le modèle asynchrone. C'est le modèle partiellement synchrone, c'est à dire que le système est asynchrone jusqu'à un instant δ (GST temps globale de stabilité) où le système devient synchrone [DLS88] (voir section 3.2 chapitre1).

5.3 Modification de la définition du problème

Cette approche consiste en une modification d'une ou de plusieurs propriétés du consensus, à savoir la propriété de validité ou d'accord, pour rendre le nouveau problème obtenu solvable dans un modèle asynchrone. Parmi ces modifications on trouve le consensus ensembliste (K set agreement) (voir section 2.5.1.3) dont la propriété d'accord est substituée par une autre propriété appelé k-accord où l'accord se fait sur k valeurs au lieu d'une seule dans le consensus traditionnel [C90].

5.4 Les détecteurs de défaillances

Une des solutions pour surmonter le résultat d'impossibilité de FLP, est d'augmenter le système par des oracles, parmi ces oracles nous trouvons les détecteurs de défaillances qui ont été introduits par Chandra et Toueg [CT96]. Un détecteur de défaillances est associé à chaque processus du calcul distribué et est chargé de détecter les défaillances externes.

La détection des pannes des processus est classiquement réalisée en utilisant le mécanisme des limites de temps (timeout). Les détecteurs de défaillances sont une formalisation de ce mécanisme et de tout autre mécanisme de détection de pannes. Un détecteur de défaillances donne des informations, qui peuvent être incorrectes, sur le schéma de pannes.

On peut caractériser un détecteur de défaillances suivant les informations qu'il donne sur le schéma de pannes. On aura donc un ensemble de classes de détecteurs de défaillances qui fourniront des informations plus ou moins fortes sur les pannes.

Chandra et Toueg [CT96] ont défini formellement ce qu'est un *détecteur de défaillances* et ont spécifié des classes de détecteurs de défaillances qui permettent de résoudre de grandes catégories d'applications distribuées.

5.4.1 Spécifications de base

Chaque processus *p* est équipé d'un module local de suspicion de défaillances. A chaque instant t, ce module donne à *p* une liste de processus, suspectés d'être défaillants. Si un processus *q* appartient à cette liste, on dira que le processus *p* **suspecte** *q* à l'instant t.

Les informations données par le détecteur de défaillances peuvent être inexactes. Cependant un détecteur de défaillances ne sera utile à une application distribuée que si une partie de ces informations sont exactes. Les informations exactes peuvent concerner les processus corrects ou incorrects. On peut vouloir qu'un processus correct ne soit pas suspecté (**Exactitude**), et qu'un processus incorrect (**Complétude**) le soit.

5.4.2 Les classes des détecteurs de défaillance

Un détecteur de défaillances est caractérisé par deux propriétés abstraites: la complétude et l'exactitude. En général, la complétude exige qu'un détecteur de défaillances doit suspecter tous les processus défaillants, tandis que l'exactitude restreint les erreurs (suspections erronées) qu'un détecteur de défaillances peut faire.

Une spécification d'un détecteur de défaillances est formée par une combinaison de ces deux types de propriétés. Chandra et Toueg [CT96] définissent plusieurs classes de détecteurs de fautes franches.

Ces classes sont caractérisées par des propriétés d'Exactitude et de Complétude plus ou moins fortes:

- **Complétude forte:** Chaque processus incorrect sera suspecté par tous les processus corrects.
- **Complétude faible:** Chaque processus incorrect sera suspecté par au moins un processus correct.
- **Exactitude forte:** Aucun processus ne sera suspecté avant de tomber en panne.
- **Exactitude faible:** Au moins un processus correct n'est jamais suspecté.
- **Exactitude ultime forte:** À partir d'un certain instant, aucun processus correct ne sera suspecté par les processus corrects.
- **Exactitude ultime faible:** À partir d'un certain instant, il existe un processus correct qui ne sera plus jamais suspecté par aucun processus correct.

D'un point de vue combinatoire, les différentes combinaisons des propriétés d'Exactitude et de Complétude définissent huit classes de détecteurs de défaillances

Complétude	**Exactitude**			
	Forte	**faible**	**ultime forte**	**ultime faible**
forte	Perfect P	Strong S	Eventually Perfect ◊P	Eventually Strong ◊S
faible	 Q	Weak W	 ◊Q	Eventually Weak ◊W

Figure 2.4 : Classes de détecteur de défaillances

Chandra et Toueg [CT96] ont prouvé qu'il existe un algorithme de passage de la classe de complétude forte vers la classe de complétude faible en conservant l'exactitude, en d'autre termes nous avons l'équivalence suivante : (P ≡ Q, S ≡ W, ◊P ≡ ◊Q et ◊S ≡ ◊W), les quatre classes de détecteurs de défaillances sont :

- ***P*:** La classe des détecteurs de défaillances parfaits (Complétude forte et Exactitude forte): toutes les pannes sont détectées de manière exacte.

- ***S*:** La classe des détecteurs de défaillances forts (Complétude forte et Exactitude faible): tous les processus en panne seront suspectés et au moins un processus correct ne sera pas suspecté (mais pas tous).

- ***◊P*:** La classe des détecteurs de défaillances ultimement parfaits (Complétude forte et Exactitude ultime forte), tous les processus en panne seront détectés de manière exacte au bout d'un certain temps.

- ***◊S*:** La classe des détecteurs de défaillances ultimement forts (Complétude forte et Exactitude ultime faible), tous les processus en pannes seront suspectés et à partir d'un certain instant au moins un processus correct ne sera plus suspecté.

Une autre classe de FD est la classe omega Ω. La sortie d'un détecteur de défaillances de la classe Omega ne fournit pas à chaque processus une liste de processus suspectés d'être incorrects. Mais désigne *un processus* (le **leader**) suspecté d'être correct. Ainsi au lieu de suspecter un ensemble de processus, chaque processus fait confiance à *un processus*. Cette classe est équivalente à ◊S: Ω ≡ ◊S [ADFT04]

6 Conclusion

Le résultat d'impossibilité de FLP s'applique aussi aux fautes byzantines car la panne franche est incluse dans le comportement aléatoire, et comme le consensus est la brique de base pour résoudre d'autres problèmes d'accords, on se focalise sur le problème du consensus byzantin dans un environnement asynchrone.

Chapitre 3

Consensus byzantin dans un système asynchrone

1 Introduction

L'écriture d'algorithmes distribués résistants aux pannes byzantines dans un modèle asynchrone peut se révéler délicate et complexe. Nous pouvons avoir une approche modulaire pour développer des algorithmes dans un modèle asynchrone enrichi d'une primitive qui encapsule des propriétés temporelles du système.

Ainsi, nous pouvons choisir, soit de rester à bas niveau en considérant le système comme partiellement synchrone, par exemple ; soit d'être plus abstrait en considérant le système comme asynchrone mais enrichi par un détecteur de défaillances ou une autre abstraction permettant de pouvoir résoudre le consensus byzantin.

Dans ce chapitre, nous allons faire un tour d’horizon sur les différentes approches pour résoudre le consensus byzantin et les conditions liées à celles-ci.

2 Conditions pour assurer un consensus byzantin

Le consensus Byzantin a été présenté par Pease, Shostak et Lamport [PSL82] dans un modèle synchrone, où ils ont montré que la condition nécessaire et suffisante pour résoudre le problème est que le nombre des processus corrects doit être supérieur a 2t (t nombre de fautes byzantines) bien sûr sans l'utilisation d'authentification. Cette condition a été étendue par G. Bracha et S. Toueg [BT85] vers le modèle asynchrone mais avec l'utilisation d'authentification ou bien l'utilisation des mécanismes de diffusion fiable. Dans [DLS88], Dwork et al ont prouvé qu'il est impossible d'assurer un consensus byzantin qui tolère n/3 processus byzantins c'est-à-dire le système tolère au maximum (n - 1) /3 fautes byzantines. Cette condition est la plus faible dans le modèle de fautes byzantines, cependant cette dernière devienne plus forte en fonction de la conception de l'algorithme utilisé ; parmi ces conditions fortes $t<n/6$ et $t<n/5$ dans [FMR05].

Et comme il n'existe pas de solution du consensus byzantin dans un système complètement asynchrone d'une manière déterministe, même l'utilisation des détecteurs de défaillance ne résolve pas le problème car l'implémentation des détecteurs de défaillance se base sur le modèle partiellement synchrone selon Larea et al [LFA02].

Donc la deuxième condition est que le système doit être partiellement synchrone ou bien un sous ensemble du système est inéluctablement synchrone [ADFT06, MMT07]

3 Restriction du comportement byzantin

Le problème du consensus byzantin devient plus facile si on restreint la capacité des processus à altérer les messages. Cela peut être atteint lorsque chaque nœud attache une signature digitale à son message. La définition d'un message signé suppose que :

1. Un message signé par un processus non fautif ne peut pas être contrefait.
2. Toute corruption du message est détectable.
3. La signature peut être authentifiée par tout autre processus.

Dans ce cas, aucun nœud ne peut altérer le contenu du message d'un processus corrects sans que l'altération ne soit détectée par les autres processus non fautifs.

De même elle permet d'identifier l'émetteur du message. Ainsi lorsqu'un processus non fautif envoie un message aux autres, le processus byzantin ne peut pas altérer son message et l'envoyer aux autres nœuds. S'il tente de le faire, l'altération sera détectée.

Évidemment, cela limite le comportement d'un processus fautif, Cependant la signature toute seule ne permet pas de détecter les fautes qui ne s'envisagent qu'à l'état interne du processus byzantin. Pour cela les certificats sont appliqués afin de garantir une consistance avec l'état interne,

Le certificat (ou les preuves) est l'ensemble de messages signés reçus dans les échanges précédents, cet ensemble est attaché à un message pour permettre à son récepteur de vérifier la validité du contenu du message reçu, [DS97, KMM03]. En d'autres termes, il garantit que la valeur envoyée est conforme aux messages reçus durant les étapes précédentes. Le consensus byzantin devient plus simple si les messages sont signés (ou authentifiés) et certifiés.

Une autre solution pour restreindre le comportement byzantin est l'utilisation de la diffusion fiable [BT85][ADFT06] qui empêchent les processus byzantins de diffuser un message mutant durant une ronde quelconque. K. Aguilera et al [ADFT06] ont présenté un mécanisme de diffusion fiable similaire à celui de [BT85], appelé la diffusion unique consistante CUB (consistent unique broadcast) cette diffusion consistante assure que:

1 les processus corrects délivrent le même ensemble de messages : si un processus correct diffuse un message m, alors tous les processus corrects délivrent m.
2 durant une diffusion, un processus délivre au maximum un seul message pour une ronde donnée.

Donc, pour assurer un consensus byzantin déterministe dans un système asynchrone, les concepteurs des algorithmes font un choix entre l'utilisation des mécanismes de diffusion fiables dont le nombre de messages générés est important ou bien d'utiliser les certificats signés dépendants de la couche d'application ce qui rend le système lourd.

4 Protocoles résolvant le consensus byzantin asynchrone

4.1 Protocoles à Oracles

L'idée originale des détecteurs de défaillance est de détecter ou bien, de suspecter le crash d'un processus. Chaque processus attache un module de détection de défaillance, l'ensemble de tous ces modules forme le détecteur de défaillance. Ce paradigme a été pendant longtemps considéré uniquement dans des modèles avec défaillances franches. Malkhi et Reiter [MR97] ont été les premiers à étendre cette notion au modèle byzantin

Les principales différences par rapport aux détecteurs de crash sont que :

(1) Le détecteur de fautes Byzantines ne peut pas être complètement indépendant de l'algorithme dans lequel il est employé [MR97].
(2) L'impossibilité de détecter tous les fautes Byzantines, mais seulement un sous-ensemble [DS97, KMM03,].

4.1.1 Le détecteur de défaillance ◊S(bz)

Malkhi et Reiter [MR97] ont défini une classe ◊S(bz) de détecteurs de fautes inéluctablement forte et ils ont prouvé que ce détecteur de faute ◊S(bz) peut être employé pour résoudre le problème du consensus binaire dans un système réparti asynchrone tolérant (n -1)/3 fautes Byzantines.

Le travail présenté par Malkhi et Reitre se situe dans un modèle où les processus communiquent grâce à une primitive de diffusion fiable qui suppose de plus un ordre causal. Le détecteur de cette solution suspect les processus qui omettent la diffusion des messages (suspecte l'émetteur de ces messages), et les autres fautes seront masquer par l'algorithme qui résout le consensus.

La deuxième contrainte dans cette solution est l'utilisation de la validité stricte à la place de la validité traditionnelle, dont le but est d'assurer que la valeur décidée est une valeur initiale d'un processus correct et non pas d'un autre qui peut être byzantin (la valeur décidée est une valeur initiale d'un processus correct si et seulement si tous les processus corrects ont la même valeur initiale), mais cette solution est trop restrictives.

4.1.2 Le détecteur du mutisme et le consensus vectoriel

4.1.2.1 Propriétés et hypothèses

En vue d'affaiblir les contraintes posées par Malkhi et al [MR97], Doudou et Al [DS97] ont présenté une solution au consensus byzantin mais avec des hypothèses plus faibles c'est à dire la primitive de diffusion est simple (pas d'ordre causal ni de diffusion fiable définie par Malkhi et Al). Ils ont opté un nombre t des processus byzantins, inférieur à n/3 et pour que les canaux de communication soient FIFO et fiables.

Doudou et al ont introduit la notion de vecteur dans le consensus byzantin qui devient le consensus vectoriel définit par les anciennes propriétés (la terminaison, l'accord) et par la validité vectorielle qui réquisitionne qu'au moins t+1 éléments du vecteur de décision sont des valeurs initiales des processus corrects. (D'un coté pour empêcher un processus de changer sa valeur initiale et d'un autre coté pour assurer que la valeur décidée est une valeur initiale d'un processus correct voir section 2.5.1.5 chapitre 2)

Parmi les comportements des processus byzantins on trouve les processus qui refusent d'émettre des messages nécessaires au déroulement de l'algorithme qu'ils exécutent. Ces processus sont appelés les processus *muets.* A partir de cette notion Doudou et al [DS97] ont définit le nouveau détecteur de défaillance du mutisme noté ◊M et qui est spécifié par la complétude du mutisme et l'exactitude faible :

- **Complétude du mutisme** : il existe un temps à partir duquel tout processus $p_{i,}$ qui est muet pour un processus p_j , est suspecté pour toujours par p_j

- **Exactitude faible** au moins un processus correct qui n'est pas suspecter par n'importe quel processus

De plus pour détecter les autres comportements des processus byzantins, Doudou et al ont utilisé les signatures et les certificats. La signature consiste en l'utilisation d'une technique d'authentification tel que le cryptage asymétrique (clé privé et clé public) pour signé le message dont l'objectif est de vérifier l'identité de l'émetteur du message pour empêcher les processus byzantins d'impersonner les processus corrects (ils ont utilisé RSA comme technique). Et comme il est impossible de contrôler les variables locales d'un

processus byzantin alors ils ont utilisé les certificats pour vérifier ou bien certifier une valeur de même que pour empêcher les processus byzantins de diffuser deux messages différents en même temps.

Remarque : il existe deux techniques pour crypter (authentifier) les messages, soit par l'utilisation de la technique RSA ou bien par l'utilisation des fonctions d'hachages

4.1.2.2 Principe du protocole

La solution proposée par Doudou [DS97] résout le consensus vectoriel, alors en premier ils ont commencez par réduire le consensus vers le consensus vectoriel ensuite ils ont construit le vecteur de valeurs initiales signées dans une première étape. La deuxième étape étant le module du consensus

Dans la première étape chaque processus diffuse sa valeur initiale vers tout les processus et collecte (n - t) messages certifiés. A partir de ces messages, chaque processus construit son propre vecteur initial (null quand le message n'est pas reçu), cette étape s'exécute une seule fois et s'achève une fois que le vecteur est construit pour donner la main au module du consensus pour démarrer.

Avant d'entamer la deuxième étape, il est intéressant de mentionner qu'un message n'est accepté dans cet algorithme que s'il est correctement formé.

Messages correctement formés : lorsque un processus p_i reçoit un message M d'un autre processus p_j, il vérifie la validité du message M, cela oblige l'émetteur p_j d'ajouter des informations supplémentaires 'Certificats'. Chaque certificat est composé d'un ensemble de (2 n + 1) / 3 messages de la même ronde mais des différents processus. On dit qu'un message est correctement formé si les certificats et la signature sont valides bien sûr pour une ronde donnée.

La deuxième étape, se base sur le principe du coordinateur tournant et se déroule en une ou plusieurs rondes asynchrones. Une ronde est divisée en deux phases,

Dans la première phase le coordinateur diffuse sa valeur d'estimation et attend l'acquittement des autres processus. Une fois qu'il a construit l'ensemble de (2n+1) / 3 acquittements il diffuse un autre message appelé « estimatec ».

Chaque processus correct recevant « estimatec » pour la première fois le rediffuse à son tour. Dés qu'un processus correct collecte une majorité de « estimatec », il décide sur cette valeur et diffuse cette décision pour ne pas bloquer les autres processus. Un processus décide donc dans deux cas, soit lors de la réception d'un message de décision des autres processus ou bien s'il accumule une majorité de « estimatec ».

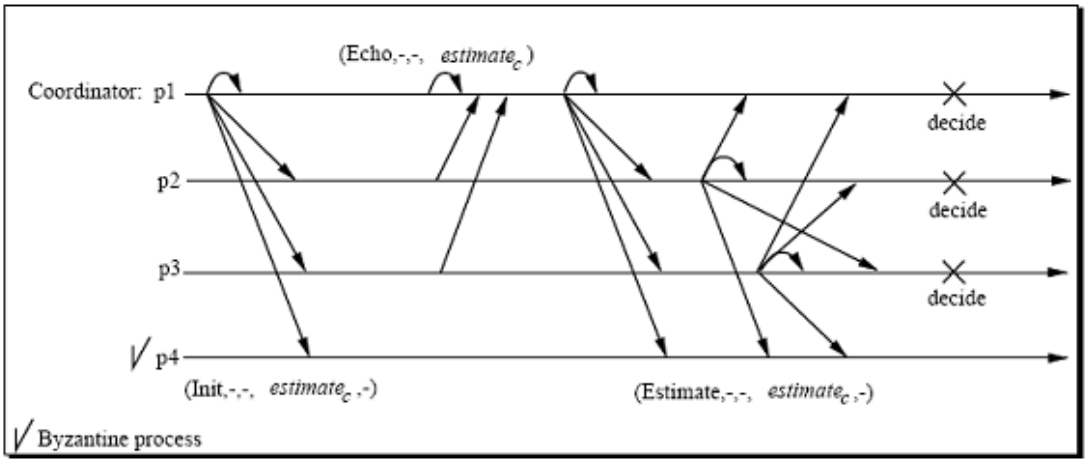

Figure 3.1 : Etapes de communication durant une ronde de [DS97]

Toutefois, si le coordinateur est suspecté par une majorité de processus corrects, la phase 2 intervient afin de garantir que, si une décision v est prise dans la ronde actuelle, alors tout les processus corrects commenceront la prochaine ronde avec leurs valeurs courantes égales à v.

```
Function ConsensusWithCertifiedInitValue(vect_i) ;
Cobegin
|| Upon recepton of Well Formed (Decide, p_j, r_j, v_j, certifDecision_j) from p_j :
     Send_all (Decide, p_i, r_j, v_j, CertifDecision_j) ; return v_j

|| Loop
    Estimate_i ← vect_i; phase_i ←1;currentRoundTerminated_i←false;
    certifDecision_i←{}; certifSuspection_i←{}; certifEstimate_i←{};
    coordSuspected_i←false;c_i←(r_i mod n)+1
    if (i = c_i) then /* p_i is the coordinator for this round  */
     Send_all (Init, p_i, r_i, v_i,estimate_i, certifIitn_i) ;

   While not currentRoundTerminated_i

    Upon reception of Well Formed (Init, p_j, r_i, v_j, estimate_j, certifIitn_j) ;
      If (j = c_i) ∧ (p_j has no already sent (Echo, p_j, -) ) ∧ (estimate_i is a certified init vale)
        Then Send (Echo, p_i, r_i,estimate_j) ; to p_c

    Upon reception of (Echo, p_j, r_i,estimate_j) from p_j
      If (i = c_i) then
        certifEstimate_i← certifEstimate_i ∪ (Echo, p_j, r_i,estimate_j);
        If ( | certifEstimate_i | = [(2n + 1)/3]) then
          Send_all (Estimate, p_i, r_i, v_i, estimate_i, certifEstimate_i) ;

    Upon reception of Well Formed (Estimate, p_j, r_i, v_j, estimate_j, certifEstimate_j) ;
    From p_j When (phase_i = 1) :
      If | certifDecision_i | ={} then
        estimate_i ← estimate_j;  certifEstimate_i ← certifEstimate_j;
        if i<>c_i then Send_all (Estimate, p_i, r_i, v_i, estimate_i, certifEstimate_i) ;
    certifDecision_i ← certifDecision_i ∪ (Estimate, p_j, r_i, v_j, estimate_j, certifEstimate_j);
      if ( | certifDecision_i | =[(2n + 1)/3]) then
        Send_all (Decide, p_i, r_j, estimate_i, CertifDecision_i);  return estimate_i;

    Upon (c_i ∈ ◊M_i ∨ Byzantine_i(c_i)) When not coordSuspected_i :
      Send_all (Suspicion, p_i, r_i); coordSuspected_i ← true;

    Upon reception of (Suspicion, p_j, r_i) From p_j When (phase_i = 1) :
      certifSuspection_i ← certifSuspection_i ∪ (Suspicion, p_j, r_i)
      if ( | certifSuspection_i | =[(2n + 1)/3]) then
        phase_i←2;  Send_all (GoPhase2,  p_i, r_j, estimate_i, CertifEstimatei, certifSuspection_i);

    Upon reception of Well Formed (GoPhase2,  p_i, r_j, estimate_i, CertifEstimatei, certifSuspection_i)
    From p_j :
      If (phase_i = 1) then
        certifInit_i ←{}; phase_i ← 2;
        certifSuspection_i ← certifSuspection_j ;
        Send_all (GoPhase2,  p_i, r_j, estimate_i, CertifEstimatei, certifSuspection_i);

      certifInit_i ← certifInit_i ∪ ( p_j, r_j, estimate_j, CertifEstimate_j, certifSuspection_j) ;
      if ( | certifInit_i | =[(2n + 1)/3]) then
        (tmpesimate_i,tmpCertifEstimate_i)←LastCertifiedestimate(r_i,certifInit_i);
        if (tmpesimate_i ≠ null) then
          estmate_i ←tmpesimate_i
          certifEstimate_i ←tmpCertifEstimate_i;
        currentRoundTerminated_i ← true ; r_i ← r_i +1;
  end while
 end loop  coned
```

Figure 3.2 : consensus vectoriel avec un détecteur du mutisme [DS97]

4.1.3 Le détecteur de défaillance ◊W(Byz,A) et ◊ *S(*Byz,*A*)

Dans le modèle de Malkhi et Reiter, et celui de Doudou et Schiper, quelques comportements byzantins qui peuvent être suspectés, ne sont pas détectés par le détecteur de défaillances, mais plutôt sont masqués par le service d'émission fiable ou bien par l'algorithme du consensus.

En vue de libérer l'algorithme du consensus de ces comportements, Un autre protocole a été présenté par KIM POTTER KIHLSTROM1 et al [KMM03], l'objectif de ce protocole est d'essayer de saisir autant d'informations sur les fautes byzantines que possible. Plutôt que de les masquer, ce nouveau détecteur de défaillance permet de détecter et d'exclure les processus byzantins du système.

Une fois qu'un processus défaillant a été enlevé, il ne compte plus dans la condition de résilience.

4.1.3.1 Propriétés et hypothèses

Les auteurs de l'article [KMM03], ont définit les détecteurs de fautes byzantines en termes de fautes détectables, c'est à dire détecter la déviation de l'algorithme où le détecteur est impliqué, ils ont définit d'autres propriétés de complétude liée aux comportements byzantins :

- **Complétude byzantine ultimement forte d'un algorithme A :** il y a un temps à partir duquel chaque processus qui dévie de l'algorithme A est détecté d'une manière permanente par tout les processus corrects

- **Complétude byzantine ultimement faible d'un algorithme A :** il y a un temps à partir duquel chaque processus qui dévie de l'algorithme A est détecté d'une manière permanente par certains processus corrects.

En plus de la propriété de complétude, ils ont utilisé les propriétés d'exactitude définit par Chandra et Toueg [CT96] pour créer deux nouvelles classes de détecteur de fautes byzantines voir Figure 3.3

Complétude	Exactitude	
	ultimement forte	ultimement faible
ultimement byzantine forte	◊ P(Byz,A) Ultimement parfait	◊S (Byz,A) Ultimement fort
ultimement (k+1) byzantine faible	◊ Q(Byz,A)	◊W (Byz,A) Ultimement faible

Figure 3.3 : Classes de détecteurs de défaillances byzantines [KMM03]

Dans les systèmes caractérisés par le comportement byzantin, la transformation d'un détecteur de fautes faible vers le détecteur de défaillances fort ne préserve pas la propriété d'exactitude. Cependant cette transformation est possible dans les systèmes avec pannes franches [CT96]. Pour remédier à ce problème, une nouvelle propriété de complétude a été présentée :

- **Complétude byzantine ultimement (k+1) faible d'un algorithme A :** il y a un temps à partir duquel chaque processus qui dévie de l'algorithme A est suspecté par au moins k+1 processus corrects.

Le protocole présenté par les auteurs tolère (n-1)/3 fautes byzantines dans la résolution du consensus traditionnel, de plus il utilise le modèle partiellement synchrone M3 pour implémenter le détecteur de défaillance ◊W(Byz,A) qui serai transformé vers un détecteur plus fort ◊S(Byz,A) par l'algorithme lui-même.

4.1.3.2 Principe du protocole

Les éléments d'intersection entre ce protocole et celui proposé par Doudou sont l'utilisation des signatures et des certificats. Ce protocole est décomposé en cinq tâches qui s'exécutent en parallèles, dont la première est la tâche principale qui se base sur le principe du coordinateur tournant et se déroule en une ou plusieurs rondes asynchrones, chaque ronde est divisée en trois phases.

Durant la première phase, chaque processus diffuse sa valeur d'estimation calculée pendant les rondes précédentes ou bien sa valeur initiale, le coordinateur de la ronde courante collecte (n - t) messages d'estimation. Il choisit alors une valeur parmi ces valeurs puis il diffuse un message de type « select » qui contient cette valeur, c'est la deuxième phase. Dans la troisième phase un processus attend la réception de (n + t)/2 + 1 messages de type « confirm » ou bien il suspecte le coordinateur en consultant le module de détection et diffuse un message pour passer à la prochaine ronde, et dans le cas contraire il diffuse un message pour indiquer qu'il est prêt à décider sur cette valeur.

La diffusion du message « confirm » aura lieu dans la deuxième tâche, une fois qu'un processus reçoit le message «select » pour la première fois. La tâche de décision qui est la troisième tâche , après qu'un processus reçoit les n-t messages de type prêt il décide la valeur contenue dans ces messages.

Les tâches quatre et cinq transforment le détecteur de fautes D1 $\in$ ◊W(Byz,A) en un autre détecteur D2 type ◊S(Byz,A). Dans la tâche quatre, un processus p envoie périodiquement une liste des processus suspectés obtenue à partir de son module local D1 de détecteur de défaillance. Dans la cinquième tâche, un processus p ajoute un autre processus q à D2 si et seulement si p a reçu k +1 (lui-même y compris) rapports des processus qui suspects q selon le détecteur D1.

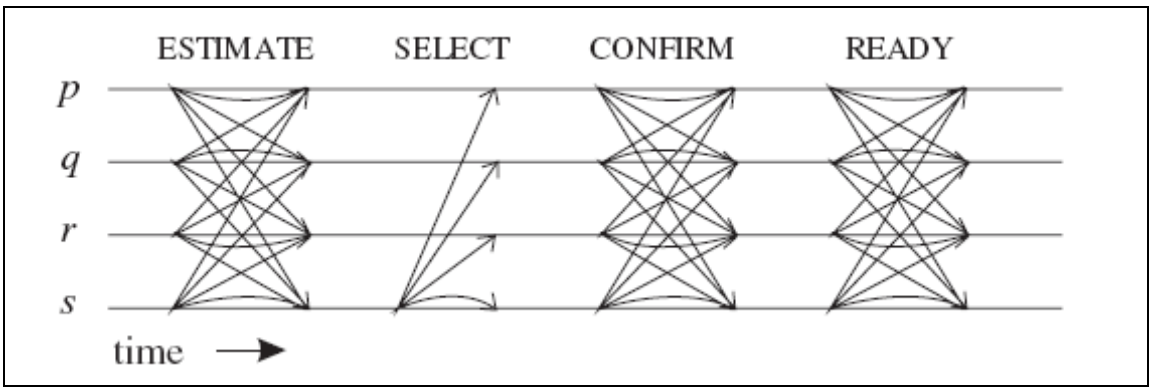

Figure 3.4 : Etapes de communication durant une ronde de [KMM03]

4.1.3.3 Implémentation du détecteur

Dans le même article les auteurs ont présenté deux autres algorithmes dans le quel ils ont détaillé l'implémentation du détecteur de défaillances dans un modèle partiellement synchrone M3. L'implémentation du détecteur se décompose en deux parties, la première sert à détecter les fautes d'omission tandis que la deuxième a comme objectif de suspecter les fautes de commission.

La première se base sur les « timeouts », une activation des « timeouts » pour n'importe quel message attendu, si le « timeout » s'écoule alors il faut ajouter ce processus dans la liste des processus suspectés, et si le message arrive ultérieurement alors il faut éliminer le processus de la liste et incrémenté le « timeout » car nous sommes dans un système asynchrone et nous ne pouvons pas différencier entre un processus byzantin et un autre qui est lent.

Dans la deuxième partie le détecteur vérifie la conformité des messages c'est à dire il vérifie les signatures et les certificats des messages. Il détecte en plus les processus qui diffusent les messages mutants.

$e_p \leftarrow v_p$; $r_p \leftarrow 0$; confirms$_p \leftarrow \emptyset$; suspected$_p \leftarrow \emptyset$; output$(D_2)_p \leftarrow \emptyset$;
for each r **in** S suspecting$_p[r] \leftarrow \emptyset$;
cobegin ** Five concurrent tasks */*
---------- */* Task 1: */*
repeat forever
$r_p \leftarrow r_p + 1$; $c_p \leftarrow (r_p \bmod n) + 1$;
send ⟨⟨ ESTIMATE, p, r_p, e_p, $tsp\rangle_p$, confirms$_p\rangle_p$ to all;---------- */* Phase 1: */*
if $[p = c_p]$ **then**---------- */* Phase 2: */*
wait until [for $n - k$ distinct processes q: p received properly formed and justified
⟨⟨ESTIMATE , q, r_p, e_q , $ts_q \rangle_q$, confirms$_q\rangle_q$ from q];
estimates$_p \leftarrow$ { ⟨ESTIMATE , q, r_p, e_q , $ts_q\rangle_q$: p received properly formed and justified
⟨⟨ESTIMATE , q, r_p, e_q , $ts_q\rangle_q$, confirms$q\rangle$ q from q};
$t_s \leftarrow$largest ts_q : ⟨ESTIMATE, q, r_p, e_q, $ts_q\rangle_q \in$ estimates p;
if [ts = 0 **and** (for at least k + 1 distinct processes q and common value e: ⟨ESTIMATE , q, r_p, e, $ts\rangle_q \in$ estimates$_p$)] **then** $es \leftarrow e$; **else** $es \leftarrow e_q$: ⟨ESTIMATE, q, r_p, e_q , $ts\rangle_q \in$ estimates p;
send ⟨⟨SELECT, p, r_p, e_s, $ts\rangle_p$, estimates$p\rangle_p$ to all;
---------- --/* Phase 3: */
wait until [(for [$(n + k)/2$] + 1 distinct processes q and common values e, select$_p$: p received properly formed and justified ⟨⟨CONFIRM, q, r_p, $e\rangle_q$, select$_q\rangle_q$ from q) **o**r $c_p \in$output$(D_2)_p$];
if [for [$(n + k)/2$] + 1 distinct processes q and common values e, select$_p$: p received properly formed and justified ⟨⟨CONFIRM , q, r_p, $e\rangle_q$, select$_p\rangle_q$] **then**
$ts_p \leftarrow r_p$; $e_p \leftarrow e$;
confirms$_p \leftarrow$ {⟨⟨CONFIRM , q, r_p, $e\rangle_q$: p received properly formed and justified ⟨⟨CONFIRM, q, r_p, $e\rangle_q$, select$_q\rangle_q$ };
send ⟨⟨READY, p, r_p, $e\rangle_p$, confirms$_{p\rangle p}$ to all;
else send ⟨NREADY, p, $r_p\rangle_p$ to all;
---------- */* Task 2: */*
repeat forever
if [(p received properly formed and justified ⟨⟨SELECT, c, r, e, $ts\rangle_c$, estimates$_c\rangle_c$
from $c \equiv (r \bmod n) + 1$) **a**nd (p has not previously sent ⟨⟨CONFIRM, p, r, $-\rangle_p$, $-\rangle_p$)] **then**
select$_p \leftarrow$ { ⟨SELECT, c, r, e, $ts\rangle_c$};
send ⟨⟨CONFIRM, p, r, $e\rangle p$, select$_p\rangle_p$ to all;
/ Task 3: */*
wait until [for [$(n + k)/2$] + 1 distinct processes q and a common r, e: p received
properly formed and justified ⟨⟨READY, q, r, $e\rangle_q$, confirms$_q\rangle_q$ from q]; decide(e);
----------*/* Task 4: */*
repeat forever
suspected$_p \leftarrow D_{1p}$;
send ⟨SUSPECT, p, suspected$_p\rangle_p$ to all;
----------*/* Task 5: */*
when p receives ⟨SUSPECT , q, suspected $_q\rangle_q$ from q
for each r **in** S
if $r \in$ suspected$_q$ **then**
suspecting$_p[r] \leftarrow$ (suspecting$_p[r] \cup \{q\}$);
else
suspecting$_p[r] \leftarrow$ (suspecting$_p[r] - \{q\}$);
if |suspecting$_p[r]| \geq k + 1$ **then**
output$(D_2)_p \leftarrow$(output$(D_2)_p \cup \{r\}$);
else output$(D_2)_p \leftarrow$ (output$(D_2)_p - \{r\}$);
coend

Figure 3.5 : Algorithme du consensus avec un détecteur ◊W(Byz,A1) [KMM03]

4.1.4 Le détecteur de défaillance ◊Pmute

4.1.4.1 Propriétés et Hypothèses

Friedman et al [FMR05] ont utilisé une classe améliorée des détecteurs de défaillances présentée par doudou [DS97] pour résoudre le consensus byzantin. Cette nouvelle classe est caractérisée par les propriétés suivantes:

- **Complétude forte du mutisme** Eventuellement, on suspecte d'une manière permanente chaque processus muet par tout processus correct.

- **Exactitude ultime forte :** Il y a un temps après lequel, aucun processus correct n'est suspecté.

Remarque les détecteurs de défaillances de la classe **◊Pmute** peuvent être implémentés dans un système partiellement synchrone.

Cette solution s'appuie sur des principes de conception simple et est relativement facile à comprendre, leur complexité de message est $O(n^2)$ par ronde, la taille de n'importe quel message est O(1), ce protocole n'utilise pas des mécanismes lourds tels que les certificats ou n'importe quel genre de signatures du niveau application, il suppose qu'un processus ne peut pas impersonner un autre. Cette solution est caractérisée par une seule étape de communication, mais elle ne tolère que n/6 processus byzantins voir figure 3.6

4.1.4.2 Principe du protocole

En plus de la terminaison et de l'accord, il est nécessaire au consensus byzantin d'assurer la validité, à savoir, la valeur décidée doit être v quand tous les processus corrects proposent le v. pour ceci, le protocole est basé sur le principe suivant:

(1) si le nombre d'occurrence de la valeur la plus courante dévie d'un certain seuil, alors cette valeur sera décidée.
(2) autrement, le paradigme de coordination est employé pour forcer par la suite une valeur à être adoptée par suffisamment de processus de sorte que la propriété précédente devienne satisfaite.

Les processus procèdent alors dans des rondes consécutives asynchrones, Chaque ronde se compose de deux phases. La première phase de chaque ronde est un échange global des valeurs où chaque processus diffuse sa valeur d'estimation et attend la réception de n-t messages des autres processus. Une fois qu'une valeur a été reçue plus de n/2 fois alors le processus adapte cette valeur comme la valeur dominante, si ce n'est pas le cas alors il adapte la valeur initiale comme valeur dominante, alors que la deuxième phase est basée sur le paradigme du coordonnateur tournant.

Durant la deuxième phase, si un processus reçoit une valeur dominante plus de (n - 2t - $\#_{\perp}$**)** fois, **a**lors il modifie sa valeur d'estimation à cette valeur dominante, et cette dernière ne serai décidé que si le nombre atteint (n - t). Dans le cas où le seuil de cette valeur est inférieur de (n - 2t - $\#_{\perp}$**)** alors ces processus attendent la valeur du coordinateur**,** cependant, si un processus pi ne reçoit pas la valeur du coordinateur ou le suspect, il utilise sa valeur dominante et courante comme valeur par défaut, sinon il utilise la valeur proposée par le coordinateur pour forcer la décision dans les prochaines rondes.

Function Consensus(v_i)
init: $est_i \leftarrow v_i$; $r_i \leftarrow 0$;
repeat forever
(C01) $r_i \leftarrow r_i + 1$; $V_i \leftarrow [\perp, \ldots, \perp]$; $c \leftarrow ((r_i - 1) \bmod n) + 1$;
———————————- Step 1 of round r ————————————————————
(C02) broadcast VAL(r_i, est_i);
(C03) **wait until (**VAL(r_i,−) or DEC(−) messages have been rec. from all non-suspected proc and from at least (n − f) distinct processes);
(C04) **for each** j: **if** (VAL(r_i, est_j) or DEC(est_j) rec. from pj) **then** $V_i[j] \leftarrow est_j$ **endif**;
(C05)**if** ($\exists v \neq \perp : \#_v(V_i) > n/2$) **then** $dominating_i \leftarrow v$ e**lse** $dominating_i \leftarrow est_i$ **endif**;
———————————- Step 2 of round r ————————————————————
(C06) **if** i = c **then** broadcast COORD(r_i, $dominating_i$) **endif**;
(C07) **if** ($\#dominating_i(V_i) \geq n - 2f - \#_{\perp}(V_i)$) **then**
(C08) $est_i \leftarrow dominating_i$;
(C09) **if** ($\#dominating_i(V_i) \geq n - f$) **then** broadcast DEC(est_i); **return** (est_i) **endif**
(C10) **else**
(C11) **wait until** ((COORD(r_i,−) or DEC(−) received from p_c) ∨ (p_c is suspected)
(C12) **if (** COORD(ri, x) or DEC(x) received from pc)
(C13) **then** $coord\_val_i \leftarrow x$
(C14) **else**
(C15) $coord\_val_i \leftarrow dominating_i$
(C16) **endif**;
(C17) $est_i \leftarrow coord\_val_i$
(C18) **endif**
end repeat

Figure 3.6: Consensus avec *◊Pmute FD et* ($n > 6f$) [FMR05]

4.1.5 Protocoles indéterministes (protocoles probabilistes)

Dans la littérature, les protocoles randomisés sont classés par des auteurs dans la catégorie des oracles alors que d'autres auteurs les classes dans la catégorie des protocoles indéterministes par le faite que la terminaison est probabiliste. Les premiers protocoles randomisés pour résoudre le consensus byzantin dans les systèmes asynchrones ont été proposé par Ben-Or [B83] et Rabin [R83].

Le protocole de Ben-Or's exige n > 5f, deux étapes de communication par ronde, et $O(n^2)$ messages par ronde. Le protocole de Rabin exige n > 10 f. Un autre protocole qui utilise les nombres aléatoires a été introduit par R Friedman et al [FMR05], ce protocole tolère moins de n/5 fautes byzantines, il utilise un générateur partagé et exige une seule étape de communication par ronde, mais la contrainte à respecter est d'empêcher les processus byzantins de commander l'ordre dans lequel les messages seront délivrés aux processus corrects.

4.1.5.1 Principe du protocole

Le principe de cet algorithme est simple, au début de chaque ronde, tout processus correct pi obtient le prochain nombre aléatoire du générateur en commun et le stocke dans une variable locale. Par la suite, il diffuse sa valeur d'estimation et attend la réception de (n – f) valeurs d'estimations ou bien des valeurs de décision. pi établit à partir de ces données un vecteur Vi qui représente sa vue courante des valeurs courantes d'estimation.

Dans la deuxième partie du protocole, nous avons deux cas possibles : Dans le premier cas si un processus n'a pas de valeur dominante dans son vecteur alors il initialise sa valeur d'estimation par la valeur du générateur, dans le cas contraire la valeur d'estimation c'est la valeur dominante, et en ce qui concerne la décision, elle aura lieu si la valeur dominante est égale à la valeur du générateur.

L'algorithme est probabiliste c'est à dire il termine avec une probabilité égale à 1, la propriété d'accord est assurée par l'hypothèse du protocole c'est à dire n > 5f. Un processus décide une valeur v s'il a reçu 3f+1 messages, parmi ces messages, f messages sont en provenance des processus byzantins, tandis que l'autre ensemble qui peut être formé ne contient que 3f messages donc ils ne peuvent pas décider une autre valeur ni de l'accepter. Ils n'acceptent que la valeur du générateur, celle qui a été décidé en haut.

Function Consensus(v_i)
init: $est_i \leftarrow v_i$; $r_i \leftarrow 0$;
repeat forever
(101) $r_i \leftarrow r_i + 1$; $V_i \leftarrow [\perp, \ldots, \perp]$; $s_i \leftarrow$random();
(102) broadcast VAL(ri, est_i);
(103) **wait until** (VAL(r_i,−) or DEC(−) messages have been rec from at least (n − f) distinct proc.);
(104) **for each** j: **if** (VAL(r_i, est_j) or DEC(est_j) received from p_j)
then Vi[j] ← estj **endif**;
(105) **if** ($\exists v : v \neq \perp : \#v(Vi) \geq n - 2f$)
(106) **then** esti ← v;
(107) **if** ($s_i = v$)
(108) **then** broadcast DEC(est_i); **return** (est_i) **endif**
(109) **else** $est_i \leftarrow s_i$
(110) **endif**
end repeat

Figure 3.7 : Consensus randomisé qui tolère moins de n / 5 fautes byzantines [FMR05]

4.1.6 Wormholes

Correia, Neves, poumon et Verissimo ont étudié le concept pour résoudre le consensus en utilisant une base sécurisée de calcul synchrone (Trusted Timely Computing Base TTCB) [CNLV05]. Un TTCB est un lien de communication spécial, également surnommé le trou de ver (wormehole), et qui garantit un minimum de synchronisme dans un environnement byzantin et asynchrone.

L'idée est d'utiliser ce lien comme lien d'urgence pour les aspects critiques des applications (par exemple, la diffusion de la valeur du coordinateur dans le protocole du consensus), où la majeure partie du système emploie les autres liens standards asynchrones.

4.2 Ajout des suppositions temporelles

Il est impossible d'implémenter un détecteur de défaillance dans un système complètement asynchrone, et pour la rendre possible il faut utiliser un système partiellement synchrone [LFA02]. Nous constatons l'utilisation de deux éléments des hypothèses temporelles en plus du détecteur, alors qu'il est possible de résoudre le consensus dans un système partiellement synchrone.

Quant nous disons qu'un système est partiellement synchrone cela veut dire que les n^2 liens de communication qui forment le système sont partiellement synchrones. La question qui se pose est : est il possible d'assurer un consensus byzantin avec une partie des n^2 liens partiellement synchrone ?

La réponse est affirmative. Les premiers ayant présenté une solution sont : Aguilira et al [ADFT06] ils ont proposés une solution avec 2(n-1) liens (un processus correct dont tous les liens entrant et sortant sont partiellement synchrones) au lieu des n^2 liens partiellement synchrones, alors que dans la solution la plus récente de H Moumen et al [MMT07] le nombre est de 4t (précisément un processus correct qui est bisource avec 2t processus)

4.2.1 Consensus byzantin avec un processus bisource

Un processus est dit bisource si tous ses liens entrants et sortants sont partiellement synchrones, cette notion a été utilisée par Aguilira et al [ADFT06] pour résoudre le consensus byzantin, la solution présentée a comme hypothèse l'existence d'un processus correct bisource pour assurer la terminaison, et que le nombre de processus est $n \geq 3t+1$.

Afin de limiter le comportement byzantin, la diffusion fiable a été utilisée, avec deux protocoles :

- Le premier, est un protocole de Diffusion unique cohérente (consistant unique broadcast)
- Le deuxième est appelé l'Emission sûre et prouvable (Provable Reliable Send). Cette primitive peut être employer par un processus p pour envoyer un message m à q afin qu'une troisième partie obtienne une preuve que m est en transit. Elle garantit donc que si p est correct que tous les processus corrects r obtiennent la preuve que m est en transit

4.2.1.1 Principe du protocole

Cet algorithme se déroule dans des rondes asynchrones, chaque ronde est gérée par un coordinateur, ce dernier n'est appelé qu'à la dernière étape de la ronde une fois que les processus ne sont pas d'accords sur une valeur donnée, il intervient pour forcer le consensus sur une valeur, elle est alors la valeur dominante dans les messages demandeurs d'aide du coordinateur. Cependant ce coordinateur peut se comporter d'une manière aléatoire et ne diffuse pas les messages, c'est pour cela que les auteurs de cet article [ADFT06] on définit l'Emission sûre et prouvable pour s'assurer que les (n-t) messages sont en transit vers le coordinateur et à ce moment là, les processus activent les temporisateurs pour éviter le blocage.

Dans les autres étapes, le protocole utilise trois fois la diffusion fiable (CUB), elle est appelée pour la première fois pour certifier les valeurs, une valeur n'est acceptée dans les autres étapes que si le nombre d'occurrences est t+1, dans la deuxième étape la diffusion fiable est utilisée pour échanger les valeurs d'estimation alors que dans la troisième étape elle est appliquée pour proposer une valeur dominante.

Durant la quatrième phase un processus décide une valeur dont il a reçus n-t propositions, et en fonction du nombre d'occurrence de cette valeur le processus accepte ou non la valeur du coordinateur.

```
Code for process p:
Initialization:
Timeout ← 1
function certified(k)
1 return {w : cudelivered (CERTIFY; k;w) from at least f + 1 processes }
To propose(v):
2 k ← 0
3 while true do
4 k← k +1
(* phase 0: certification *)
5 cubcast(CERTIFY; k; v)
6 wait until cudeliver (CERTIFY; k; ∗) from n - f processes
(* phase 1: reporting estimates *)
7 v ← value cudelivered most in (CERTIFY; k; ∗) messages
8 cubcast(REPORT; k; v)
9 wait until cudeliver (REPORT; k; ∗) from n - f processes with ∗ ∈ certified(k)
```

```
(* phase 2: proposing the most common estimate *)
10 w ← value cudelivered most in (REPORT; k; *) messages
11 if all (REPORT; k;_) with * ∈certified(k) are for w
12 then cubcast(PROPOSE; k; w)
13 else cubcast(PROPOSE; k;?)
14 wait until cudeliver (PROPOSE; k; *) from n - f processes with * = w
or (_* = ? and 1 - w ∈ certified(k))
(* phase 3: consulting coordinator *)
15 accept_ coord_ new_ estimate ← true
16 if cudelivered (PROPOSE; k;x) with x ≠ ? from n - f processes then
17 decide x ; 18 v ← x ; 19 accept_ coord_ new_ estimate ← false
20 else if cudelivered (PROPOSE; k;x) with x ≠ ? from n - 2f processes then
21 v ← x
22 accept_ coord_ new_ estimate ← false
23 else if cudelivered (PROPOSE; k;x) with x ≠ ? from n - 3f processes then
24 v ← x
25 psend (HELP-REQ; k; v) to k mod n
26 wait until getproof of (HELP-REQ; k; *) from n - f processes to k mod n
27 start time ← clock()
28 wait until received (HELP-RESP; k; y) from k mod n or clock() - start time > Timeout
29 if received (HELP-RESP; k; y) from k mod n then
30 if accept_ coord_ new_ estimate then v ← y
31 else Timeout ← Timeout + 1
(* coordinator's help *)
upon preceive (HELP-REQ; k; *) from n - f processes do
32 z ← value that occurs most in (HELP-REQ; k; *) messages
33 send (HELP-RESP; k; z) to all
```

Figure 3.8 : Implémentation du consensus binaire avec au moins un processus ◊bisource[ADFT06]

4.2.2 Consensus byzantin avec un processus 2t-◊bisource

Dans l'article [MMT07], les auteurs proposent un modèle de système où le modèle de communication se situe entre le modèle asynchrone et le modèle partiellement synchrone. Le modèle supposé, considère que seulement quelques liens, qui sont inéluctablement synchrones, quand tous les liens sont asynchrones alors le modèle est asynchrone. Tandis que si tous les liens sont inéluctablement synchrones le système est partiellement synchrone, donc le modèle proposé est plus fort que le modèle asynchrone pour lequel il n'existe pas de solution et qui est plus faible que le modèle partiellement synchrone.

Les liens de communication inéluctablement synchrones doivent respecter un certain comportement afin de résoudre le consensus byzantin. Ce comportement est capturé par la notion x-◊bisource. La bisource assumée par [ADFT06] a une valeur maximale (x = n-1). Plus exactement, un éventuel x-bisource est un processus correct où le nombre de voisins privilégiés est x au lieu de n-1. D'après [MMT07] on peut définir un processus x-◊bisource suivant les définitions suivantes

Définition1 Un lien reliant un processus p avec n'importe quel processus q est synchrone à l'instant τ si (1) aucun message envoyé par p à l'instant τ est reçu par q après ($\tau + \delta$) ou (2) le processus q est incorrect.

Définition 2 un processus p est un x-◊bisource à l'instant τ si:

- (1) p est correct,
- (2) Il existe un ensemble **X** de cardinalité x, où : pour tous processus **q** dans **X**, les liens de **q** vers **p** et de **p** vers **q** sont synchrone à l'instant τ. Les processus de l'ensemble **X** sont les voisins privilégiés de **p**

4.2.2.1 Hypothèses du protocole

A l'instar du dernier protocole, le protocole proposé par H Moumen et al [MMT07] utilise les certificats et les signatures pour suspecter les processus byzantins, dont le nombre tolérés est de (n-1)/3, et pour assurer la terminaison du consensus byzantin les auteurs supposent l'existence d'un processus correct 2t-◊bisource.

4.2.2.2 Principe du protocole

Après la phase d'initialisation, le protocole procède dans des rondes consécutives asynchrones. Chaque ronde est coordonnée par un processus prédéterminé. Ainsi, le protocole emploie le paradigme du coordinateur tournant. Chaque ronde se compose de quatre phases de communication.

Chaque processus qui commence une ronde, (le coordonnateur y compris) envoie d'abord sa propre estimation (avec le certificat associé) au coordonnateur (PC) de la ronde et active le temporisateur du coordinateur. Quand le temps du temporisateur expire tout en attendant la réponse du coordinateur, la valeur du temporisateur est incrémentée (la valeur du temporisateur sera incrémenté d'une ronde à une autre jusqu'à ce que le système devienne synchrone) D'ailleurs, ceci empêche pi de se bloquer tout en attendant la réponse d'un coordonnateur défectueux.

Quand le coordonnateur de la ronde courante reçoit pour la première fois un message valide (peut-être de lui-même), contenant une valeur d'estimation, il envoie un message de coordination COORD (est) à tous les processus. C'est le point essentiel de ce protocole, d'après l'hypothèse 2t+1 ◊bisource (2t entrant et 2t sortant plus le coordinateur lui même), lorsque le coordinateur diffuse une valeur v alors t+1 processus corrects rediffusent cette valeur dans la deuxième phase (cette phase est comme un répéteur, faire entendre ceux qui n'ont pas entendu la valeur du coordinateur). À la fin de cette deuxième phase, tous les processus adoptent la valeur du coordinateur (n-t > 2t).

La décision aura lieu dans la quatrième phase une fois qu'un processus collecte n-t valeurs identiques, alors que la troisième phase n'est utile que lorsque le coordinateur est byzantin pour l'empêcher de diffuser deux valeurs différentes c'est à dire après cette phase une seule valeur dans le système est certifiée.

```
Function Consensus(vi)
Init: ri ← 0; Δi[1::n] ← 1;
Task T1: % basic task %
——————————————————————————- init phase ——————————————————————————
(1) send INIT(ri; vi) to all;
(2) wait until INIT(ri; *) received from at least (n - t) distinct processes ;
(3) if ( ∃ v : received at least (n - 2t) times ) then esti ← v else esti ← vi endif;

repeat forever
(4) c ← (ri mod n) + 1;  ri ← ri + 1;
——————————————————————————- round ri ——————————————————————————
(5) send QUERY(ri; esti) to pc; set timer(Δi[c]);
(6 ) wait until COORD(ri; est) received from pc or time-out) store value in auxi;%else ⊥%
(7) if (timer times out)) then Δi[c] ←Δi[c] + 1 else disable timer endif;

(8) send RELAY(ri; auxi) to all;
(9) wait until (RELAY(ri;*) received from at least (n - t) distinct processes) store values in
Vi;
(10) if (Vi – {⊥} = {v}) then auxi ← v else auxi ← ⊥ endif;
(11) send FILT1(ri; auxi) to all;
(12) wait until FILT1(ri;*)received from at least (n - t) distinct processes) store values in Vi
(13) if (Vi = {v}) then auxi ← v else auxi ←⊥ endif;
(14) send FILT2(ri; auxi) to all;
(15) wait until FILT2(ri; _) received from at least (n - t) distinct processes) store values in
Vi;
(16) case (Vi = {v}) then send DEC(v) to all; return(v);
(17) (Vi = {v, ⊥}) then esti ← v;
(18) endcase;
————————————————————————————————————————————————————————————————
end repeat
Task T2: % coordination task %
(19) upon receipt of QUERY(r; est) for the first time for round r: send COORD(r; est) to
all;
Task T3:
(20) upon receipt of DEC(est): send DEC(est) to all; return(est);
```

Figure 3.9 : Protocole du Consensus byzantin (hypothèse 2t-◊bisource)

5 Comparaisons des solutions présentées

La plus part des solutions présentées tolèrent (n-1)/3 fautes à l'exception des protocoles présentés par Friedman et al [FMR05] où la tolérance est de n/5 et n/6. En ce qui concerne les méthodes pour suspecter le comportement byzantin il y a deux choix : soit par l'utilisation de la diffusion fiable ou bien par l'utilisation des certificats signés. Si nous comparons ces deux catégories nous trouverons que le nombre de messages générés par les protocoles basés sur la diffusion fiable est de $o(n^3)$ alors que dans l'autre cas il est de $o(n^2)$.

Un autre critère de comparaison est l'hypothèse temporelle, la seule résolution avec un système complètement asynchrone c'est l'utilisation des solutions indéterministes [FMR05]. En ce qui concerne les solutions déterministes la solution présenté par H Moumen et al [MMT07] est la plus faible c'est à dire qu'elle suppose (2t-◊bisource) 4t liens partiellement synchrones, le reste de la comparaison est dans la figure suivante :

(avec N: nombre de processus, FD: type du détecteur de défaillances, SYNCH : modèles temporels, MSG : nombre de messages, Cetif+Sign : certificats + signatures, PS : partiellement synchrone, t : nombre de processus byzantins)

	PROTECTION	TYPE	N≥	AIDE	FD	SYNCH	MSG
[DS97]	Certif+Sign	vectoriel	3t+1	coordinateur	◊M	PS	$O(n^2)$
[KMM03]	Certif+Sign	Binaire	3t+1	coordinateur	$\Diamond S$(Byz,*A*)	PS (M3)	$O(n^2)$
[FMR05]	----------	Binaire	6t+1	coordinateur	◊P Mute	PS	$O(n^2)$
[FMR05]	----------	Binaire	5t+1	Common coin	------	asynchrone	$O(n^2)$
[ADFT06]	Diffusion fiable	Binaire	3t+1	Coordinateur	------	n-1 ◊bisource	$O(n^3)$
[MMT07]	Certif+Sign	Multivalué	3t+1	Coordinateur	------	2t-◊bisource	$O(n^2)$

Figure 3.10 : Comparaison des différents protocoles présentés

6 Conclusion

Dans ce chapitre, nous avons présenté les protocoles résolvant le consensus byzantin dans un système asynchrone. Ces protocoles sont implémentés selon les modèles temporels et les propriétés du consensus.

Parmi les instances du consensus byzantin, nous retrouvons : le consensus strict, le consensus vectoriel et le consensus probabiliste.

Chaque instance ne peut être implémenté que dans des modèles temporels bien définis. Parmi ces modèles nous avons :

- Le modèle asynchrone pour résoudre le consensus probabiliste.
- Le modèle partiellement asynchrone pour le consensus vectoriel.
- Le modèle inéluctablement synchrone pour résoudre le consensus byzantin avec les propriétés traditionnelles d'accord, validité et terminaison.

Chapitre 4

Implémentation du consensus byzantin

1 Introduction

A l'exception des protocoles randomisés, les protocoles déterministes n'ont pas de solution au problème du consensus byzantin dans un système asynchrone. Les solutions possibles sont basées sur le synchronisme partiel, plus exactement sur le temps physique du système. Ainsi, il est possible d'implémenter un détecteur de fautes crashs dans un système asynchrone. Cette solution est proposée par Mostefaoui, Mourgaya, et Raynal [MMR03] et elle n'est pas fondée sur des hypothèses de synchronie, c'est-à-dire qu'elle n'utilise pas le temps physique dans l'implémentation. Elle se base sur un mécanisme de questions et de réponses (Query-Response mechanism) et sur une hypothèse comportementale [MMR03, MRT06] traduisant le fait que pour une interrogation donnée, les réponses de certains processus arrivent toujours parmi les (*n - t)* premières réponses.

Dans ce chapitre, nous proposons deux protocoles, le premier résout le consensus byzantin dans un système inéluctablement synchrone en se basant sur le principe des canaux gagnants [MMR03, MRT06] . Ce protocole est fondé sur deux idées :

1. Une idée complètement indépendante du temps physique du système, plus exactement, c'est une propriété comportementale du système [MRT06] pour assurer un consensus byzantin avec un nombre minimal de canaux gagnants, nous supposons l'existence d'un processus ◊2tWC (2t winning chanels). Cette hypothèse pour assurer la terminaison.
2. Une deuxième idée qui repose sur le protocole proposé par [MMT07] pour concevoir notre protocole (au lieu de 2t -◊bisource de [MMT07], nous utilisons ◊2tWC)

Le deuxième protocole, est un protocole hybride qui utilise les deux propriétés temporelles et comportementales pour résoudre le consensus byzantin.

2 Les canaux gagnants

L'implémentation des canaux gagnants se base sur le mécanisme QUERY-RESPONSE. Ce dernier peut être facilement mis en application dans un système distribué asynchrone plus spécifiquement, n'importe quel processus pi peut diffuser un message QUERY et attendre les messages correspondants REPONSE des n-t processus (ce sont les réponses gagnantes pour cette question, et les processus correspondants sont les processus gagnants pour cette requête). Les t messages non reçus sont les réponses perdantes de la requête. La notion réponse gagnante est une notion libre du temps (complètement asynchrone) dans le sens que son exécution n'exige pas des temporisateur (timeout), Mostefaui et al [MRT06] ont définis les canaux gagnants comme suit :

Définition 1 (canal inéluctablement gagnant). Pour deux processus pi et pj, le canal dirigé pi $\rightarrow$ pj est inéluctablement gagnant (noté ◊WC) si, après un instant inconnu t, il existe une réponse de pi à chaque requête émise par pj, cette réponse est gagnante (t est fini mais inconnu).

Définition 2 (x-◊WC). S'il existe un processus correct pi, et un ensemble Q formé de x processus pj tel que $\forall pj \in Q$ et $pi \notin Q$, alors le canal $p_i \rightarrow p_j$ est inéluctablement gagnant.

3. Modèle du système et hypothèses

3.1. Le modèle du système

On considère un système distribué composé d'un ensemble de n processus (n >3). Ces processus peuvent se comporter d'une manière aléatoire, le nombre maximale de fautes byzantines tolérés est t=(n-1)/3 donc n >3t . Le système de communication est fiable, c'est à dire, qu'il n'existe aucune défaillance sur les canaux de communication, et que le réseau est complètement connecté. Les processus communiquent entre eux par l'échange de messages. Il n'y a aucune borne sur les délais de transmission des messages et les vitesses relatives des processeurs.

3.2 Modèle d'authentification

Un processus peut se comporter d'une manière aléatoire, et pour se protéger contre ces comportements on suppose l'existence de deux modules. Le premier est un module de certificat alors que le deuxième est un module de signature. Le module de certificat, certifie la valeur envoyée par n'importe quel processus en ajoutant un ensemble qui contient les messages reçus durant les étapes précédentes cela permet à un récepteur de vérifier la consistance des variables internes de l'émetteur.

Le deuxième module assure l'identité de l'émetteur donc de détecter l'altération dans le message (fonction d'hachage ou RSA).

Ces deux modules n'assurent pas la protection de touts les comportements byzantins. Dans un système asynchrone un processus attend n-t messages, il est possible qu'il reçoive n messages donc il peut créer deux valeurs certifiées. Afin de ne pas compliquer les protocoles présentés dans ce travail, nous supposons que ces deux modules sont implémentés dans une couche supérieure.

3.3 Hypothèses

Les propriétés du consensus présentées dans ce mémoire sont les propriétés traditionnelles : l'accord, la validité et la terminaison. Les deux premières sont assurées par l'algorithme lui-même, tandis que la terminaison est assurée par une hypothèse temporelle. Comme première hypothèse, nous supposons qu'il existe un processus correct p_x 2t◊WC. La deuxième hypothèse est que le nombre de processus est n, tel que n > 3t (t est le nombre de processus byzantin).

Function Consensus(vi)
Init: $r_i \leftarrow 0$; last_round_coord $\leftarrow 0$, $count_i \leftarrow [0,...,0]$, $RECEIVED_I \leftarrow \{i\}$;
Task T1: % basic task %
———————————————- init phase ———————————————
(1) send INIT(r_i; v_i) to all;
(2) **wait until** INIT(r_i; *) received from at least (n - t) distinct processes ;
(3) **if** (∃ v : received at least (n - 2t) times) **then** $est_i \leftarrow$ v **else** $est_i \leftarrow v_i$ **endif**;

repeat forever
(4) $c \leftarrow (r_i \bmod n) + 1$; $r_i \leftarrow r_i + 1$; $aux_i \leftarrow \perp$;
———————————————- round ri ———————————————
(5) send query(r_i , est_i) to all;
(6) **Wait until** (RESPONSE (r_i , *) received from (n-f) processes);
(7) Let $RECEIVED_I$= the set received at line 6 ;
(8) **For_each** j ∈Π - $RECEIVED_I$ **do** $count_i[j] \leftarrow count_i[j] + 1$ **enddo**
(9) **if** (∃ response(r_i , v) from pc at line 6) **then** $aux_i \leftarrow$ v **endif**

(10) send RELAY(ri; auxi) to all;
(11) **wait until (**RELAY(r_i;*) received from at least (n - t) distinct processes) **store values** in Vi;
(12) **if** (Vi – {⊥} = {v}) **then** $aux_i \leftarrow$ v **else** $aux_i \leftarrow \perp$ **endif**;

(13) send FILT1(r_i; aux_i) to all;
(14) **wait until** FILT1(r_i;*)received from at least (n - t) distinct processes) **store values** in Vi
(15) **if** (Vi = {v}) **then** $aux_i \leftarrow$ v **else** $aux_i \leftarrow \perp$ **endif**;

(16) send FILT2(r_i; aux_i) to all;
(17) **wait until** FILT2(r_i; _) received from at least (n - t) distinct processes) **store values** in Vi;
(18) **case** (Vi = {v}) **then** send DEC(v) to all; **return**(v);
(19) (Vi = {v, ⊥}) **then** $est_i \leftarrow$ v;
(20) **endcase**;
———————————————————————————————————————

end repeat

Task T2 %Server task%

(21) **Upon** recept of QUERY (r, est) from p_j :
(22) **If** (i = (r mod n) and r > last_round_coord and r ≥ r_i) **then**
(23) send RESPONSE (r_i , est) to all;
(24) last_round_coord ← r
(25) **else** Send response (r_i , -) to p_j ;
(26) **endif;**

Task T3 %Decision task%
(27) **upon** receipt of DEC(est): send DEC(est) to all; **return**(est);

Figure 4.1 : Consensus byzantine 2t winning et n>3t

4 Principe du protocole

Le protocole présenté dans la figure 4.1 utilise les certificats et suppose l'existence d'un processus correct 2t-◊WC. Le protocole débute par une étape d'initialisation où tout les processus échangent leurs valeurs pour certifier celle qui est dominante en vue de l'utiliser dans les prochaines étapes.

Après une phase d'initialisation, le protocole peut être décomposé en trois tâches qui s'exécutent en parallèles, la première tâche du protocole procède à des rondes consécutives asynchrones.

Chaque ronde est gérée par un coordinateur prédéterminer (c= r_i mod n, avec r_i le numéro de la ronde courante). Durant la première phase d'une ronde r (lignes 5-9), chaque processus qui débute une ronde, fait appel au mécanisme requête/ réponse (query /response) d'un coté pour détecter les canaux gagnants (Winnings channels) et d'un autre coté pour attendre la valeur du coordinateur.

Chaque processus diffuse un message Query étiqueté par sa valeur d'estimation et le numéro de la ronde courante, et attend la réception de (n-t) réponse, il est possible que la réponse du coordinateur soit parmi ces réponses (au lieu d'attendre d'une façon directe la valeur du coordinateur, cette solution exploite le mécanisme QUERY / RESPONSE pour cacher la notion de temps). Si cela est le cas alors, il mis à jour la variable AUX_I par la valeur reçue dans la réponse du coordinateur.

Dans le cas contraire, lorsque la réponse du coordinateur n'appartient pas à cet ensemble alors il opte $\perp$ comme valeur pour AUX_I. Les n-t réponses sont utilisées pour pondérer les processus c'est à dire incrémenter le compteur des processus dont la réponse n'a pas été reçue.

Les canaux gagnants sont ceux qui après un instant inconnu t le compteur est le plus petit et reste inchangé.

Quand le coordonnateur de la ronde courante reçoit pour la première fois à la ligne 21 un message valide QUERY (peut-être de lui-même) contenant une valeur d'estimation EST, il diffuse un message RESPONSE étiqueté par *EST* et le numéro de la ronde concernée.

Le message RESPONSE est envoyé à partir d'une autre tâche (tâche 2 lignes 21-26) qui s'exécute en parallèle avec la première, et dont le but est de permettre aux coordinateur de gérer sa ronde même s'il est coincé dans les rondes précédentes, (les processus détectent le message de coordination par l'identité de l'émetteur du message et plus exactement par la signature).

Si le coordonnateur courant est un 2t-◊WC, il a au moins 2t voisins favorisés parmi lesquels au moins t sont des processus corrects, En conséquence, au moins (t + 1) processus corrects (les t voisins corrects et le coordonnateur lui-même) ont obtenus la valeur v du coordinateur et ainsi AUX_I=v ($v \neq \perp$). Si le coordonnateur courant est byzantin, il peut être muet et n'envoie rien à certains processus, en même temps il peut envoyer différentes valeurs certifiées à différents processus (dans ce cas, nécessairement aucune de ces valeurs n'a été décidée dans une ronde précédente). Si le coordonnateur courant n'est pas un 2t-◊WC ou s'il est Byzantin, les trois phases suivantes permettent à des processus corrects de se comporter d'une manière cohérente. Aucun processus ne décide ou si certains d'entre eux décident une valeur v, alors la seule valeur certifiée pour la prochaine ronde sera v et cela empêche les processus byzantins de présenter d'autres valeurs.

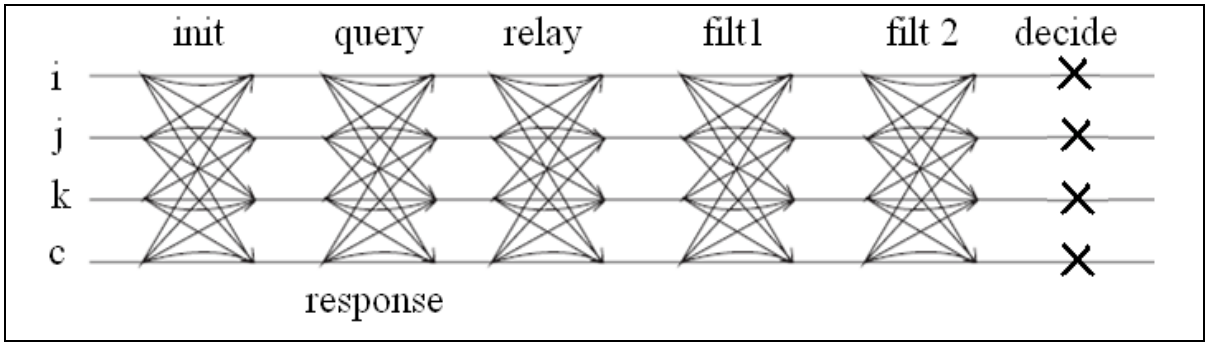

Figure 4.2 : Etapes de communications du protocole proposé

La deuxième phase d'une ronde r (lignes 10-12), Cette phase vise à prolonger la portée des ◊2t-WC, c'est à dire si le coordinateur courant est 2t-◊WC donc au moins t+1 processus corrects ont la variable AUX_I=v, et à la fin de cette étape tout les processus corrects optent pour v comme valeur à AUX_I, cela peut être réaliser par un relais de la valeur reçus du coordinateur (v ou $\perp$) , chaque processus collecte n-t messages valides de type RELAY

Si le coordonnateur est un 2t-◊WC alors quelque soit le processus correct p_j, il recevra au moins une fois la valeur du coordinateur cela résulte du fait que t est le nombre des processus byzantins et en même temps c'est le nombre des processus corrects mais qui n'ont pas reçus la valeur du coordinateur. Si nous additionnons ces deux nombres en obtient 2t ce qui veut dire qu'un processus correct ne peut collecter que 2t messages $\perp$ et comme n-t >2t alors au moins un message $v \neq \perp$ a été reçus.

La troisième phase (lignes 13-15) est une étape de filtrage, elle n'est utile que si le coordinateur est byzantin pour l'empêcher de diffuser différents messages de coordinations pour la même ronde. A la fin de cette étape une seule valeur est certifiée.

Après la troisième phase, une seule valeur certifiée qui peut être décidé par n'importe quel processus dans la quatrième phase, cette dernière a pour but d'assurer que la propriété d'accord ne sera jamais violée.

Un processus correct décide une valeur v s'il a accumulé n-t messages FILT2 contenant la valeur v. Le seuil n-t garantit la propriété d'accord car parmi ces n-t messages, t+1 sont en provenance des processus corrects. Les autres processus ont donc deux choix possible soit de décider v ou bien d'accepter v comme valeur d'estimation pour la prochaine ronde (si un processus décide v, alors v est la seule valeur certifiée dans cette ronde ou dans les prochaines rondes).

Avant qu'un processus décide sur une valeur, il diffuse un message de type DECIDE pour empêcher le blocage des processus corrects. Donc un processus peut décider dans deux cas possibles, soit dans la quatrième phase ou bien dans la troisième tâche une fois qu'il ait reçu un message valide à la ligne 27 de type DECIDE

5 Preuve du protocole

Lemme 1 : L'intersection entre l'ensemble des messages rassemblés V_i et V_j par deux processus corrects p_i et p_j durant une étape de communication donnée n'est pas vide, c'est à dire $Vi \cap Vj \neq \varnothing$

Preuve : la preuve est par contradiction, on suppose que $Vi \cap Vj = \varnothing$. Après une étape d'échange de messages un processus p_i collecte $(n - t)$ messages, parmi ces messages, f messages sont en provenance des processus byzantins ($f \leq t$). Cela implique que le nombre de messages reçus des processus corrects est $(n - t - f)$.

Alors $\forall j$ p_j collecte $(n - t - f)$ messages des $(n - t)$ processus corrects, (f messages des processus corrects est le complément de $n - t - f$).

Si $Vi \cap Vj = \varnothing$, alors $f \geq n - t - f$ ($f \leq t$,si on prend $t=f$) donc $t \geq n/3$ ce qui est contradictoire avec l'hypothèse $t < n/3$

Lemme 2 : Dans une ronde r, après l'échange de messages dans les lignes 13-14, au maximum une valeur différente de $\perp$ peut être certifié dans cette ronde r.

Preuve : Considérant une exécution où un processus pi rassemble seulement $(n - t)$ messages avec v comme valeur (les messages rassemblés constituent le certificat de v).

Par Lemme 1, aucun autre processus p_j ne peut certifier un ensemble de $(n - t)$ valeurs $w \neq v$ car les deux ensembles doivent s'intersecté et par conséquent, aucun certificat ne peut être exhibé pour une autre valeur.

Le seule cas possible est que $v = W$.

Lemme 3 un canal de p_j à p_i est 1-◊WC, si et seulement s'il existe un temps inconnu t après lequel la valeur $COUNT_I[j]$ reste inchangée.

Preuve : Le mécanisme QUERY/ RESPONSE est utilisé pour implémenter les canaux gagnants, au début de chaque ronde, tout les processus diffuse QUERY et attendent la réception des n-t RESPONSE. Une fois que ces messages ont été obtenus, les éléments du complément de l'ensemble formé des n-t sont incrémenté un par un dans la case appropriée de $COUNT_I$. Le mécanisme se répète à chaque ronde et le compteur $COUNT_I[j]$ augmente jusqu' à un instant t (ronde r) où la valeur devient constante (égale à z).

c'est-à-dire que $\forall r_i > r$ $COUNT_I[j]=z$, le processus p_i reçoit toujours la réponse du processus p_j, Ce qui assure que le lien entre p_j à p_i est 1-◊WC.

Lemme 4 : si le coordinateur est un processus correct 2t-◊WC, alors 2t+1 processus vont recevoir la valeur du coordinateur dans la ligne 6.

Preuve : Après un instant inconnu t le système devient 2t-◊WC, et d'après les propriétés des canaux gagnants (réalisable à partir du lemme 3), les 2t voisins favorisés du processus corrects 2t-◊WC recevront toujours la réponse du coordinateur (2t-◊WC) parmi les n-t réponse attendus (bien sûr cette propriété est assurée après que le système devienne stable). Si un processus parmi les 2t+1 voisins favoris (2t voisins plus le coordinateur lui même) diffuse une QUERY alors 2t+1 processus recevront la valeur du coordinateur parmi les n-t attendus, ce qui assure le lemme3

Corollaire 1 : si un processus décide une valeur certifiée v pendant une ronde, alors seulement la valeur v peut être décidé dans la même ronde ou dans les prochaines rondes (aucune autre valeur que v ne peut être certifiée).

Preuve : un processus décide une valeur v si le nombre d'occurrence de cette valeur est n-t, on appliquant le lemme2, v est la seule valeur certifiée dans le système, et d'après le lemme1, l'intersection entre les différents ensembles n'est pas vide, donc l'intersection est égale à v. Cela veut dire que si un processus décide, il décide v. s'il ne décide pas, il adopte v comme valeur d'estimation pour les prochaines rondes.

Théorème 1 (Accord) Deux processus corrects ne décident pas différemment.
Preuve : les deux cas possibles pour qu'un processus décide une valeur sont :
Soit qu'il décide dans la ligne 27 après la réception d'un message de décision certifié.
Ou bien dans la ligne 18, alors l'accord est assuré par le corollaire1.

Théorème 2 (validité) si un processus décide une valeur v, alors v a été proposée par au moins un processus ;
Preuve : la preuve est par contradiction, on suppose qu'un processus a décidé une valeur v qui n'a pas été proposée par aucun processus, c'est à dire qu'aucun processus n'a diffusé

QUERY avec v comme valeur dans la ligne 5, de même que la valeur possible dans la réponse du coordinateur peut être v′ avec (v≠v′) dans ce cas le reste des processus décident v′, donc deux pour processus ont décidés pour deux valeurs différentes, contradiction avec le théorème 1.

Théorème 3 (Terminaison): S'il existe un processus correct 2t-◊WC, alors tous les processus corrects décident inéluctablement.

Preuve : Si un processus décide à la ligne 18 alors il diffuse un DECIDE, et chaque processus qui reçoit ce message décide à la ligne 28.

On suppose qu'aucun processus n'a décidé, et la preuve et par contradiction.

D'après l'hypothèse de la solution, après un instant t le processus correct p_i devient 2t-◊WC et si on applique le lemme 4 on obtient 2t+1 processus qui ont reçus la valeur du coordinateur.

Dans le pire des cas, il y a t processus byzantin parmi les 2t +1 voisins favorisés par p_i. Un processus byzantin ne peut relier dans la ligne 11 que la valeur du coordinateur ou bien ⊥ (ce sont les deux valeurs certifiées seulement)

Ceci nous laisse conclure que la valeur v envoyé par p_i est transmise par au moins t +1 voisins corrects privilégiés de p_i (ligne 10). Puisque chaque processus rassemble au moins (n-t) message de RELAY, nous pouvons conclure que tous les processus obtiendront au moins un message RELAIY contenant la valeur v du p_i (à cet instant les processus byzantins ont un seule choix c'est le mutisme).

Durant la troisième phase tous les processus diffusent v (la valeur proposée par p_i), et comme v est la seule valeur certifiée dans le système, tous les processus diffusent une autre fois v dans la quatrième phase a cet instant tous les processus décide v à la ligne 19, ce qui prouve le théorème.

Théorème 4 : le protocole présenté dans la figure 4.1 résout le consensus byzantin dans un système asynchrone avec la présence d'un processus correct 2t-◊WC

Preuve : la preuve résulte des trois théorèmes 1,2 et 3.

6 Implémentation hybride du consensus byzantin

Si nous faisons une comparaison entre le protocole proposé et celui de Moumen et al [MMT07], nous constatons que la différence résulte dans l'attente des messages de coordination.

Dans le deuxième protocole l'attente est implémenté d'une manière explicite par l'utilisation des timeouts c'est à dire le protocole dépend du temps physique du système alors que le premier protocole est indépendant du temps physique par le fait que l'attente est assuré d'une manière implicite par l'exploitation du mécanisme QUERY-RESPONSE.

Et pour profiter au mieux de ces deux approches, nous introduisant une autre approche intéressante qui consiste en la conception du consensus byzantin dans un système qui satisfait chacune de ces hypothèses 2t-◊WC et 2t-bisource. C'est à dire donner plus de chance aux processus pour assurer le synchronisme.

Cette combinaison est définie comme une hypothèse dénotée 2t-[◊WC $\vee$ bisource].

Définition 3 (2t-[◊WC $\vee$ ◊bisource]). S'il existe un processus correct p_i, et un ensemble Q formé de 2t processus p_j tel que $\forall p_j \in Q$ et $p_i \notin Q$, le canal $p_i \rightarrow p_j$ est inéluctablement gagnant ou bien il est ◊bisource.

6.1 **Modèle du système et hypothèses**

Ce protocole hybride est implémenté dans le même modèle présenté dans la section 3, la terminaison est assurée par l'hypothèse qu'un processus correct est 2t [◊WC $\vee$ ◊bisource] et que le nombre de processus byzantins est au plus t avec n>3t.

6.2 **Principe du protocole**

La figure 17 représente une solution au problème du consensus byzantin dans un système asynchrone affaibli par l'hypothèse 2t [◊WC ∨ ◊bisource], l'algorithme de cette solution est semblable à celui présenté dans la section 4.

Après une phase d'initialisation, les processus font appels au mécanisme requête/ réponse (query /response) pour guetter le message du coordinateur, nous avons deux cas possibles :

Le premier cas est basé sur les propriétés comportementales des messages alors que le deuxième est fondé sur les propriétés temporelles. Les propriétés comportementales des messages sont satisfaites si, le message du coordinateur est parmi les n-t messages reçus dans la ligne 6.

Dans le deuxième cas, si la condition précédente n'est pas satisfaite, l'algorithme fait appel aux propriétés temporelles, par l'activation d'un temporisateur dont le but est de donner plus de chance aux processus pour recevoir le message de coordination (lignes 11 – 14). Le reste de l'algorithme est identique à la première proposition.

6.3 Preuve du protocole

Théorème 5 (Terminaison): S'il existe un processus correct 2t-[◊WC ∨ ◊bisource], alors tous les processus corrects décident inéluctablement.

Preuve : A la ligne 23 du protocole, le premier processus décide une valeur, et avant qu'il ne termine son exécution, il diffuse un message DECIDE pour éviter le blocage des autres processus, alors le reste des processus décident soit dans la ligne 23 ou bien à la réception du message DECIDE à la ligne 32.

La preuve est par l'absurde, on suppose que tout les processus corrects n'ont pas décidé. D'après l'hypothèse de la solution, après un instant inconnu q un processus correct p_i devient 2t-[◊WC ∨ ◊bisource], alors f processus sont ◊bisource avec p_i ($f \leq n - t$) et p_i a $(n - t - f)$ canaux gagnants.

Les 2t liens favorisés par p_i ne sont pas obligatoirement tous du même type, un processus soit qu'il est ◊bisource avec p_i ou bien que le canal qui le relie avec le processus pi est inéluctablement gagnant.

Après plusieurs rondes consécutives le processus p_i devient le coordinateur de la ronde courante r, chaque processus qui commence cette ronde diffuse QUERY (ligne 5), une fois que le coordinateur a reçu ce message alors il le rediffuse vers tout les processus, et comme il a 2t voisins favorisés parmi eux, t+1 processus corrects (t processus corrects plus le coordinateur lui même) ont reçu la valeur du coordinateur à la ligne 6 (propriétés comportementales) ou 12 (propriétés temporelles).

Chacun de ces t+1 processus corrects rediffuse la valeur du coordinateur à la ligne 15 pour faire entendre ceux qui n'ont pas reçu la valeur du coordinateur. A la fin de cette étape chaque processus correct (même les processus byzantins) a reçu au moins une seule fois la valeur du coordinateur (n-t >2t). Le seul comportement byzantin possible est d'omettre l'émission des messages et cela ne conduit pas au blocage car un processus correct n'attend que n-t messages (n-t soient le nombre des processus corrects).

Ensuite, tout processus correct diffuse FILT1 et FILT2 respectivement dans les lignes 18 et 21 avec la même valeur v, cela permet à ces processus de décider v à la ligne 23, et ce qui prouve le théorème

Function Consensus(vi)
Init: $r_i \leftarrow 0$; last_round_coord $\leftarrow 0$, $count_i \leftarrow [0,...,0]$, $RECEIVED_I \leftarrow \{i\}$; $\Delta_i[1::n] \leftarrow 1$;
Task T1: % basic task %
———————————————————- init phase ———————————————————
(1) send INIT(r_i; v_i) to all;
(2) **wait until** INIT(r_i; *) received from at least (n - t) distinct processes ;
(3) **if** (∃ v : received at least (n - 2t) times) **then** $est_i \leftarrow$ v **else** $est_i \leftarrow v_i$ **endif**;

repeat forever
(4) $c \leftarrow (r_i \bmod n) + 1$; $r_i \leftarrow r_i + 1$; $aux_i \leftarrow \perp$;
———————————————————- round ri ———————————————————
(5) send query(r_i , est_i) to all;
(6) **Wait until** (RESPONSE (r_i , *) received from (n-f) processes);
(7) Let $RECEIVED_I$= the set received at line 6 ;
(8) **For_each** j ∈Π - $RECEIVED_I$ **do** $count_i[j] \leftarrow count_i[j] + 1$ **enddo**
(9) **if** (∃ response(r_i , v) from pc at line 6) **then** $aux_i \leftarrow$ v;
(11) **else** set_timer(Δi [pc]);
(12) **Wait until** (RESPONSE (ri , *) received from pc or time-out;
(13) **if** (timer time-out) **then** Δi[pc] ←Δi[pc] + 1 **else** disable_timer ; auxi ← v ; **endif**
(14) **endif**

(15) send RELAY(ri; auxi) to all;
(16) **wait until (**RELAY(r_i;*) received from at least (n - t) distinct processes) **store values** in Vi;
(17) **if** (Vi – {⊥} = {v}) **then** $aux_i \leftarrow$ v **else** $aux_i \leftarrow \perp$ **endif**;

(18) send FILT1(r_i; aux_i) to all;
(19) **wait until** FILT1(r_i;*)received from at least (n - t) distinct processes) **store values** in Vi
(20) **if** (Vi = {v}) **then** $aux_i \leftarrow$ v **else** $aux_i \leftarrow \perp$ **endif**;

(21) send FILT2(r_i; aux_i) to all;
(22) **wait until** FILT2(r_i; _) received from at least (n - t) distinct processes) **store values** in Vi;
(23) **case** (Vi = {v}) **then** send DEC(v) to all; **return**(v);
(24) (Vi = {v, ⊥}) **then** $est_i \leftarrow$ v;
(25) **endcase**;

---

end repeat

Task T2 %Server task%

(26) **Upon** recept of QUERY (r, est) from p_j :
(27) **If** (i = (r mod n) and r > last_round_coord and r ≥ r_i) **then**
(28) send RESPONSE (r_i , est) to all;
(29) last_round_coord ← r
(30) **else** Send response (r_i , -) to p_j ;
(31) **endif;**

Task T3 %Decision task%
(32) **upon** receipt of DEC(est): send DEC(est) to all; **return**(est);

Figure 4.3 : Consensus byzantine avec 2t-[◊WC ∨ ◊bisource] et n>3t

7 Conclusion

Dans ce chapitre, nous avons proposé deux protocoles pour implémenter le consensus byzantin d'une manière déterministe avec des hypothèses temporelles les plus faibles. La première solution est composé de cinq étapes de communication, le nombre de messages est de $o(n^2)$, elle se base sur une hypothèse faible c'est à dire 2t-◊WC. Le même nombre de canaux est supposé inéluctablement synchrone dans le modèle de [MMT07].

Cependant d'après l'article de [MRT06] les deux modèles, canaux gagnants et inéluctablement synchrone sont deux modèles incomparables, nous avons utilisé ces deux modèles pour résoudre le consensus byzantin dans un modèle hybride, et qui a été présenté dans la deuxième solution, avec la même complexité mais avec une nouvelle hypothèse temporelle 2t-[◊WC, ◊bisource].

Conclusion Générale et Perspectives

1 Conclusion générale

En conclusion, il est évident que le consensus byzantin est un paradigme fondamental pour la tolérance aux fautes dans les calculs distribués, de plus toute solution à ce problème est valable pour n'importe quel type de fautes. Toutefois, dans un système asynchrone le consensus byzantin n'a pas de solution déterministe en présence d'un seul processus crashé.

Les différentes approches possibles pour contourner cette impossibilité, ont été présentées dans ce mémoire. Ainsi que deux protocoles ont été proposés; ils sont basés sur une approche élégante pour contourner l'impossibilité de FLP. Cette approche est fondée sur des hypothèses de synchronisme les plus faibles possible.

Le premier protocole présenté dans notre travail est le premier en son genre qui résout le consensus byzantin dans un système inéluctablement synchrone en négligeant le temps physique du système, en d'autre termes, cette solution est libre du temps (pas de timeout), ce protocole se base sur l'hypothèse qu'un processus correct est 2t gagnant.

Le deuxième protocole présenté est un protocole hybride, il utilise deux modèles : un modèle à base de temps et un deuxième modèle indépendant du temps, pour résoudre le consensus byzantin avec l'hypothèse qu'un processus correct est 2t [◊WC, ◊bisource].

2 Perspectives

2.1 Condition nécessaire pour l'implémentation du consensus byzantin dans des systèmes dynamiques :

Les solutions pour résoudre le consensus byzantin ne sont applicables que dans des systèmes bien définis, plus exactement dans des systèmes statiques où l'identité des processus est connue au préalable, Un problème reste posé concernant les conditions nécessaires ainsi que les hypothèses de synchronisme pour résoudre le consensus byzantin dans un système dynamique.

2.1 Accord byzantin avec économie d'énergie dans les réseaux de capteurs:

La tolérance aux défaillances est l'objet d'intenses recherches, d'autant plus que les réseaux de mobiles ou de capteurs nécessitent des algorithmes robustes pour continuer de fonctionner malgré la panne de certains éléments ou bien la présence d'intrus. La caractéristique principale de ces modèles est que l'énergie est limitée. En effet, les algorithmes qui résolvent par exemple le consensus byzantin sont excessivement consommateurs de ressources (processeur et réseau), alors le problème qui se pose est comment assurer un consensus byzantin en économisant l'énergie ?

Bibliographie

[AAH00] Y. Amir, G. Ateniese, D. Hasse, Y. Kim, C. Nita-Rotaru, T. Schlossnagle, J. Schultz, J. Stanton, and G. Tsudik, "Secure Group Communication in Asynchronous Networks with Failures: Integration and Experiments". In Proc. of the 20th ICDCS, pp 330–343, 2000

[ADFT04] Aguilera M.K., Delporte-Gallet C., Fauconnier H. and Toueg S, "Communication Efficient Leader Election and Consensus with Limited Link Synchrony", In Proc, 23th ACM Symposium on Principles of Distributed Computing (PODC'04), ACM Press, pp. 328-337, 2004.

[ADFT06] Aguilera M.K., Delporte-Gallet C., Fauconnier H. and Toueg S, "Consensus with byzantine failures and little system synchrony". In Proc, International Conference on Dependable Systems and Networks (DSN'06), pp. 147-155, 2006,

[B83] M. Ben-Or, "Another advantage of free choice: Completely asynchronous agreement protocols". In Proc, of the 2nd ACM Symposium on Principles of Distributed Computing, pp 27–30, 1983.

[BERN01] Bernadette Charron-Bost. "Agreement problems in fault-tolerant distributed systems, Lecture Notes In Computer Science"; In Proc. of the 28th Conference on Current Trends in Theory and Practice of Informatics, volume 2234, pp: 10 – 32, 2001

[BHG87] P A Bernstin, V.Hadzilacos and N.Googdman, "Concureancy control and recovery in database system", Ed. Addison-Wesely, 1987

[BHRT03] R. Baldoni, J. Helary, M. Raynal, and L. Tanguy. "Consensus in Byzantine asynchronous systems". In Proc, Journal of Discrete Algorithms, volume 1, n°2, pp 185–210, 2003.

[BT85] G. Bracha and S. Toueg. "Asynchronous consensus and broadcast protocols", In Proc, Journal of ACM, volume 32, n°4, pp 824–840, 1985.

Bibliographie

[C90] S. Chaudhuri, “Agreement is harder than consensus: Set consensus problems in totally asynchronous systems”, In Proc. 9th ACM Symposium on Principles of Distributed Computing, pp 311-324, 1990.

[CF99] F. Cristian and C. Fetzer. “The timed asynchronous distributed system model”. In Proc, IEEE Transactions on parallel and Distributed Systems, pp 642-657, 1999.

[CNLV05] Correia M., Neves N.F., Lung L.C. and Verıssimo P, “Low Complexity Byzantine-Resilient Consensus”. Distributed Computing, Volume 17, n°3, pp 237- 249, 2005.

[CNV06] M. Correia, N. F. Neves, and P. Verıssimo, “From consensus to atomic broadcast: Time-free Byzantine resistant protocols without signatures”, Computer Journal, volume 41, n°1, pp 82–96, 2006.

[CS00] B.Charron-Bost and A.Schiper. “Uniform consensus is harder than consensus (extended abstract)”. Technical Report DSC/200/028, EPFL, 2000.

[CT96] T.Chandra and S.Toueg. “Unreliable failure detectors for reliable distributed systems”. In Proc, Journal of the ACM, volume 43, n°2, pp 225–26, 1996.

[CT03] Bernadette Charron-Bost. “Comparing the atomic commitement and consensus problems”, In proc, of Springer Book Future Directions in Distributed Computing Book Series Lecture Notes in Computer Science Volume, 2584, pp 29-34, 2003

[DLS88] C. Dwork, N. Lynch, and L. Stockmeyer, “Consensus in the presence of partial synchrony”, In Proc, Journal of the ACM, volume 35, n°2, pp 288–323, 1988.

[DS97] A. Doudou and A. Schiper, “Muteness detectors for consensus with Byzantine processes”. Technical Report 97/30, EPFL, 1997.

[FLP85] Fischer M.J., Lynch N. and Paterson M.S, “Impossibility of Distributed Consensus with One Faulty Process”. In Proc, Journal of the ACM, volume 32, n°2, pp 374-382, 1985.

Bibliographie

[FMR05] Friedman R., Mostefaoui A. and Raynal M., “Simple and efficient oracle-based consensus protocols for asynchronous byzantine systems”. Proc in, IEEE Transactions on Dependable and Secure Computing, volume 2, n°1, pp 46-56, 2005.

[FJR06] A. FERNANDEZ , E. JIMENEZ and M. RAYNAL, “Eventual Leader Election with Weak Assumptions on Initial Knowledge, Communication Reliability, and Synchrony” .In Proc, IEEE Conference on Dependable Systems and Networks (DSN'06), IEEE Computer Society Press, pp 166-178, 2006

[HRCWS03] :HariGovind V. Ramasamy , M. Cukier , and William H. Sanders . “Formal specification and Verification of a Group Membership Protocol for an Intrusion-Tolerant group Communication System”, In Proc, of the Foundations of Intrusion Tolerant Systems (OASIS’03) IEEE, pp 9, 2003

[HT93]V. Hadzilacos and S. Toueg. “Reliable broadcast and related problems”. In Proc, Journal Of ACM, pp 97-145, 1993.

[G95] R.Guerraoui “Revisiting the relationship between non-blocking atomic commitment and consensus”. In Proc, 9th Int. Workshop on Distributed Algorithms (WDAG-9), Springer Verlag, LNCS 972, pp 87-100, 1995

[G02] Guerraoui R, “Non-Blocking Atomic Commit in Asynchronous Distributed Systems with Failure Detectors”, In Proc, Springer Distributed Computing, volume 15, n°1, pp 17-25, 2002

[KMM01] K.P. Kihlstrom, L.E. Moser, and P.M. Melliar-Smith. “The Secure Ring Group Communication System”. In Proc, ACM Transactions on Information and System Security, volume 4, n°4, pp 371–406, 2001.

[KMM03] K. P. Kihlstrom, L. E. Moser, and P. M. Melliar-Smith, “Byzantine fault detectors for solving consensus”, In Proc, The Computer Journal, volume 46, n°1, pp 16–35, 2003.

Bibliographie

[LFA02] M. Larrea, A. Fernandez, and S. Arévalo. “On the Impossibility of Implementing Perpetual Failure Detectors in Partially Synchronous Systems”, In Proc, of the 10th Euromicro Workshop on Parallel, Distributed and Network-based Processing IEEE, pp 99-105, 2002

[LSP82] L.Lamport, R. Shostak, and M.Pease. “The Byzantine generals problem”, In Proc, ACM Transactions on Programming Languages and Systems, volume 4, n°3, pp 382–401, 1982.

[MMR03] Mostefaoui A., Mourgaya E., and Raynal M., “Asynchronous Implementation of Failure Detectors”, In Proc, Int. IEEE Conference on Dependable Systems and Networks (DSN’03), pp 351-360, 2003

[MMT07] H Moumen, A MOSTEFAOUI and G TREDAN : Byzantine Consensus with Few Synchronous Links, OPODIS 2007

[MR97] D.Malkhi and M.Reiter.“Unreliable intrusion detection in distributed computations”. In Proc, of the 10th Computer Security Foundations Workshop, pp 116–124, 1997.

[MRT06] A. Mostefaoui, M. Raynal, and C. Travers, “Time-Free and Timer-Based Assumptions Can Be Combined to Obtain Eventual Leadership”, In Proc, IEEE transactions on parallel and distributed systems, volume 17, n°7, 2006

[MRTPA05] A. MOSTEFAOUI , M. RAYNAL ,C. TRAVERS , S. PATTERSON and D. AGRAWAL. “From Static Distributed Systems to Dynamic Systems”,In Proc, of the 2005 24th IEEE Symposium on Reliable Distributed Systems (SRDS’05). 2005

[PSL80] M. Pease, R. Shostak, and L. Lamport. “Reaching agreement in the presence of faults”. In Proc, Journal of ACM, volume 27, n°2, pp 228-234, 1980.

[PSL82] M.Pease, R. Shostak, and L.Lamport. “The Byzantine generals problem”. In Proc, ACM Transactions on Programming Languages and Systems , volume 4, n°3, pp 382–401, 1982.

Bibliographie

[R83] M. O. Rabin. “Randomized Byzantine generals”. In Proc, of the 24th Annual IEEE Symposium on Foundations of Computer Science, pp 403–409, 1983.

[RR03] P.Raipin Parevédy and M. Raynal. “Uniform agreement despite process omission failures”. In Proc, IEEE IPDPS Workshop on fault-tolerant Parallel and distributed Systems (FTPDS’03). IEEE, 2003.

[RR04] P.Raipin Parevédy and M. Raynal. “Optimal early stopping uniform consensus in synchronous system with process omission failures”. In Proc. of Sixteenth ACM symposium on Parallelism in Algorithms and Architectures (SPAA ’04), 2004.

[SBCM93] Samaras, G.; Britton, K.; Citron et A.; Mohan, C.. "Two-Phase Commit Optimization and Tradeoffs in the Commercial Environment," In Proc, of the Ninth International Conference on Data Engineering. Vienna, Austria, pp 520-529, 1993.

[TOU84] Toueg S., “Randomized Byzantine Agreement”. In Proc, 3th ACM Symposium on Principles of Distributed Computing (PODC’84), pp 163-178, 1984.