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

Mémoire de fin de cycle

En vue de l'obtention du diplôme MASTER
en Informatique

Option :

Réseaux et Systèmes Distribués

Thème

Le problème du k-Accord Byzantin dans les systèmes distribués asynchrones

Réalisé par :

M^r *BELILITA* Hocine
M^{lle} *BELAKBIR* Rekia

Devant le jury composé de :

Président :	M^r *BAADACHE* Atmane	Enseignant université A. MIRA de Bejaia
Rapporteur :	M^r *MOUMEN* Hamouma	Enseignant université A. MIRA de Bejaia
Examinateur :	M^r *SAADI* Mustfa	Enseignante université A. MIRA de Bejaia
Examinateur :	M^{lle} *BOUAKKAZ* Feriel	Doctorante à l'université A. MIRA de Bejaia

Année universitaire 2012/2013

Remerciements

Louange A Dieu, le miséricordieux, sans Lui rien de tout cela n'aurait pu être.

Nous tenons en premier lieu à exprimer notre profonde reconnaissance à M^r H.Moumen enseignant à l'université A. Mira de Béjaia, pour son encadrement, ses qualités tant scientifiques qu'humaines. Ce travail doit beaucoup à sa disponibilité permanente, sa rigueur scientifique et sa patience.

Nous tenons également à remercier M^r A.Baadache, d'avoir accepté de présider le jury de notre mémoire.

Nous remercions M^r M. Saadi et M^{elle} F.Bouakkaz d'avoir accepté de faire partie du jury et consacré leur temps à la lecture et à la correction de ce mémoire.

Nos remerciements les plus vifs vont tout particulièrement à nos parents, en qui nous avons puisé tout le courage, la volonté et la confiance, nous leur serons éternellement reconnaissants.

Enfin, merci à tous ceux qui ont contribué de près ou de loin à la réalisation de ce travail.

Dédicaces

Nous dédions ce modeste travail :

***A** nos chers parents.*

***A** nos frères et soeurs.*

***A** tous nos amis.*

Hocine, Rekia

Table des matières

Table des figures

INTRODUCTION GÉNÉRALE

Un système distribué typique peut être structuré comme un ensemble de processus, s'exécutant dans des postes différents, fonctionnant selon des conditions fonctionnelles spécifiques. Pendant que les systèmes deviennent plus distribués, ils deviennent également plus complexes et doivent traiter de nouveaux genres de problèmes tels que la défaillance des processus et des canaux de communication. Ainsi, gérer les systèmes distribués est une tâche difficile puisqu'on doit traiter la transmission distante et les divers types de défaillances qui peuvent résulter de la distribution.

L'avantage principal d'un système distribué est sa tolérance aux défaillances. Dans un système distribué, les services peuvent être répliqués sur plusieurs ordinateurs, ainsi la panne d'un ordinateur n'affecte pas le fonctionnement du système. Cet avantage a une importance fondamentale pour développer des systèmes qui fournit des services fiables.

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

Beaucoup de problèmes d'accord sont liés au problème de consensus. Le consensus est un paradigme fondamental pour les calculs distribués tolérants aux fautes. Informellement, le consensus permet aux processus de prendre une décision commune, qui dépend de leurs valeurs initialement proposées, en dépit des défaillances. Une façon pour affaiblir le problème du consensus est le k-accord (appelé aussi le k-consensus) qui a été introduit par Chaudhuri en 1993. Le problème du K-accord est une généralisation naturelle du problème du consensus ordinaire (qui corresponds au cas $k = 1$). Mais maintenant, au lieu d'exiger que tous les processus décident exactement sur la même valeur, nous insistons seulement que ceux-ci limitent leurs décisions sur un petit nombre,k, de valeurs distinctes.

La motivation originale pour ce problème était purement mathématique- il est intéressant d'essayer de déterminer comment les résultats quant au consensus ordinaire changent quand les exigences du problème sont changées de cette simple façon. Mais il est possible d'imaginer des situations pratiques dans lesquelles un tel algorithme peut être utile. Par exemple, considérons le problème d'allocation de ressources partageables, tel que les fréquences de broadcast dans un réseau de communication. Il est peut être désirable pour un nombre de processus de se mettre en accord sur un nombre de fréquences à utiliser pour le broadcast d'une grande quantité de données (videotape : bande vidéo).Puisque la communication est en broadcast, n'importe quel nombre de processus peuvent recevoir les données en utilisant la même fréquence. Pour minimiser la charge totale de communication, il est préférable de maintenir le nombre k de fréquences qui sont utilisés petit.

Le problème du K-accord est sujet à des recherches intensives, mais dans le contexte de défaillances par crash. Comme le k-accord est une généralisation du consensus ordinaire, des questions très intéressantes viennent à l'esprit : est ce que plus de choix permet plus de défaillances dans le modèle de défaillances le plus sévère qui est celui des Byzantines ? Comment vont être les résultats du contexte des crashs dans le contexte des Byzantins.

Dans ce mémoire, nous allons essayer à répondre à ces questions pour la première fois dans la littérature.

Ce mémoire est organisé comme suit : le premier chapitre discute les différents modèles de système distribué qui ont été considérés dans la littérature et les différents problèmes d'accord et quelques résultats de recherche au sujet de ces problèmes. Le deuxième chapitre présente quelques résultats clés pour le problème du K-accord. Le troisième chapitre présente des protocoles pour résoudre le K-accord Byzantin. Finalement, le quatrième présente un protocole pour le k-accord Byzantin tolérant aux intrusions.

Chapitre 1

Généralités sur les systèmes distribués

1.1 Introduction

Depuis leur apparition, les systèmes répartis ont toujours été sujets à divers problèmes. Les problèmes les plus intéressants dans ces systèmes exigent aux processus de coordonner leurs actions dans le même sens. Il peut aussi être aussi important pour les protocoles résolvant ces problèmes de tolérer les défaillances de processus.

Dans ce premier chapitre, nous présentons quelques notions sur les systèmes distribués. Ensuite, nous nous intéressons aux défaillances dans ces systèmes notamment les défaillances byzantines pour finir avec les problèmes d'accord. Dans ces derniers, nous nous basons sur le problème du consensus qui est l'origine du problème fondamental dans notre mémoire.

1.2 Les systèmes distribués

Il n'émerge pas, actuellement, une définition unique pour un système distribué. Il peut être défini comme un ensemble de nœuds communiquant via un réseau de communication. Chaque nœud exécute un ou plusieurs processus. Ces derniers s'exécutent d'une façon concurrente et coopèrent pour la réalisation de tâches communes. L'échange d'informatios entre les processus permet à chacun d'entre eux de décider à propos de l'état du système.

1.2.1 Modes de communication

On distingue généralement deux modes de communication : par *mémoire partagée* et par *échange de messages*.

1.2.1.1 Communication par mémoire partagée

Dans le modèle de communication par mémoire partagée [1], les processus communiquent à l'aide d'un emplacement mémoire partagé, accédé par chacun d'eux, appelé " mémoire partagée ". Cela permet à chaque processus communiquant de lire l'état de tous les autres processus. Ce mode est caractérisé par deux types de primitives qui permettent de gérer l'accès à la mémoire partagée : *Lire ( )* pour la lecture de l'information stockée dans la mémoire partagée et *Ecrire (donnée)* qui permet l'écriture de la donnée dans la mémoire partagée.

1.2.1.2 Communication par échange de messages

Il est le modèle couramment utilisé dans les systèmes distribués. Dans ce modèle[12], les processus communiquent par l'envoie et la réception de messages via des canaux de communication. Ce modèle se caractérise par l'absence d'horloge globale et de mémoire commune. Généralement, il utilise deux primitives : *Envoyer (*msg*,* p_i *)* qui permet d'envoyer le message msg au processus p_i et *Recevoir (*msg*,* p_i*)* qui permet de recevoir un message msg transmis par p_i .

1.3 Modèles temporels de communication par échange de messages

C'est une modélisation où intervient directement le temps. On distingue généralement trois modèles : synchrones, asynchrones et partiellement synchrone.

1.3.1 Le modèle synchrone

Le système synchrone est un système caractérisé par des hypothèses temporelles sur la transmission de messages. L'approche synchrone consiste à supposer que [24] :

- Il existe une borne supérieure connue comme délai de transmission d'un message. Ce délai comprend le temps nécessaire de l'émission, la transmission et la réception du message.
- Chaque processus possède une horloge logique et la dérive de cette horloge par rapport au temps réel a une borne supérieure connue.
- Il existe une borne inférieure et une borne supérieure connues au temps nécessaire à un processus pour exécuter une instruction de son programme.

La définition de ces bornes permet de définir un délai maximum au bout duquel un message doit être acquitté par le destinataire. Si l'acquittement ne parvient pas à l'expéditeur dans ce délai, une défaillance du destinataire ou du réseau est survenue. Ce délai est appelé délai de garde (*timeout*).

1.3.2 Le modèle asynchrone

Dans un système asynchrone [31], il n'ya pas d'hypothèses temporelles sur les temps d'exécution et de délivrance des messages, ce qu'il le rend le modèle le plus réaliste. Un message envoyé par un nœud non-défaillant à un autre nœud, à travers un canal non-défaillant, sera éventuellement reçu et traité, sans pouvoir garantir de borne temporelle [30].Un système asynchrone modélise en particulier un système dont la charge n'est pas prédictible (charge CPU, charge du réseau). C'est le cas de la plupart des systèmes réels utilisés actuellement.

Le fait que le choix de ce modèle ne permet de faire aucune hypothèse temporelle, rend la résolution de certains problèmes très difficile. En effet, un système asynchrone est un système où il n'est pas possible de faire la différence entre un délai de la transmission d'un message et la défaillance de l'expéditeur de ce message.

1.3.3 Le modèle partiellement synchrone

Le modèle partiellement synchrone ([14], [15],[18]) est un modèle intermédiaire entre le synchrone et l'asynchrone. Ce modèle affaiblit le modèle asynchrone par l'ajout de propriétés temporelles. La satisfaction de ces propriétés permet aux algorithmes basés sur ce modèle d'atteindre la terminaison.

D'une manière plus générale, le système partiellement synchrone se comporte d'abord d'une manière asynchrone, ensuite, il se stabilise et commence à se comporter d'une manière synchrone. Plusieurs modèles partiellement synchrones on été définis selon plusieurs critères [15].

Entre le modèle asynchrone, où aucune hypothèse temporelle n'est faite, et le modèle synchrone,

où tous les délais sont parfaitement connus, se trouve le modèle partiellement synchrone plus réaliste.

La figure 1.1 illustre les principales différences entre les trois modèles temporels.

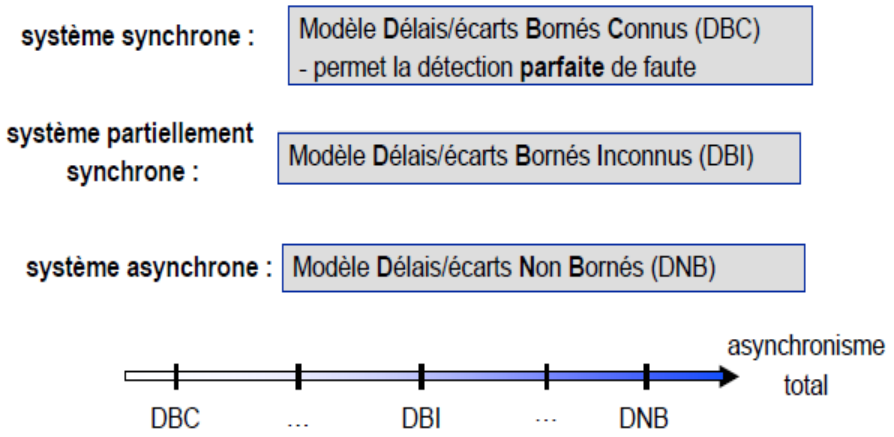

FIGURE 1.1 – Modèles temporels

1.4 Défaillances des processus et de canaux de communication

1.4.1 Processus

Un système distribué est modélisé comme un ensemble de processus qui communiquent en échangeant des messages par l'intermédiaire de canaux de communication .Dans notre travail, l'unité de défaillance est le processus. Un processus peut être correct ou incorrect. Il est dit correct s'il n'est pas défaillant. Un processus défaillant est dit incorrect ou fautif.

Dans [33], les défaillances sont classées en quatre catégories : défaillances par crash, défaillances par omission, défaillances temporelles et défaillances byzantines.

1.4.1.1 Défaillance par arrêt définitif (Crash fault)

Cette défaillance entraine l'arrêt définitif du processus. Avant la défaillance par crash, le processus a un comportement normal et à partir de celle-ci le processus cesse définitivement toute activité.

1.4.1.2 Défaillance par omission (Omission fault)

Un processus est sujet à omission sur émission ou sur réception de messages, voir les deux. Cette défaillance entraîne une cessation momentanée de l'activité du composant. Dans le cas d'un processus, certains messages peuvent, par exemple, ne pas être envoyés. Le processus reprend ensuite son activité.

1.4.1.3 Défaillance temporelle (Timing fault)

Les instants de délivrance du service par les processus ne sont pas conformes à la spécification du système. Cette défaillance suppose donc que les hypothèses temporelles réalisées sur la durée d'une tâche ou sur le délai de transmission d'un message ne sont pas respectées. Cela peut être dû au ralentissement ou à l'accélération de la durée d'une tâche ou de la transmission d'un message.

1.4.1.4 Défaillance arbitraire (Byzantine fault)

Affecté d'une telle défaillance, un processus peut avoir un comportement arbitraire pouvant aller de la panne franche aux comportements malveillants. Cette défaillance qui regroupe l'intégralité des comportements possibles permet de modéliser l'intrusion dans le système.

Un processus byzantin peut présenter différents comportements qui ne sont pas conformes à la spécification de l'algorithme qu'il exécute. Les défaillances byzantines sont généralement réparties en deux catégories : fautes détectables et non détectables.

- *Fautes byzantines détectables :* La faute détectable d'un byzantin est une faute qui est observable par ls autres processus corrects du système, le processus byzantin dans ce cas présente une déviation évidente dans son comportement externe.
  Il éxiste deux types de fautes byzantines détectables : les fautes d'omission et fautes de commission. Dans ces dernières, un processus byzantin envoie des messages qui ne doivent pas être envoyés.
- *Fautes byzantines non détectables :* une faute byzantine non détectable est une faute que les autres processus ne peuvent pas observer. Dans ce cas, un processus byzantin peut à titre d'éxemple envoyer un message comme un processus correct alors que son état interne utilise un autre message.

Dans notre mémoire nous nous intéressons à ce cas le plus général des défaillances. Le processus défaillant dans ce cas est dit byzantin. Ce modèle de défaillance des processus est caractérisé par

l'absence de toute hypothèse sur le mode de défaillance. En effet, tout ce que dit la spécification du byzantin est que le comportement est arbitraire. Cela veut dire " faillir de n'importe quelle manière ". La figure 1.2 montre que les déffaillances byzantines incluent tous les types de défaillances.

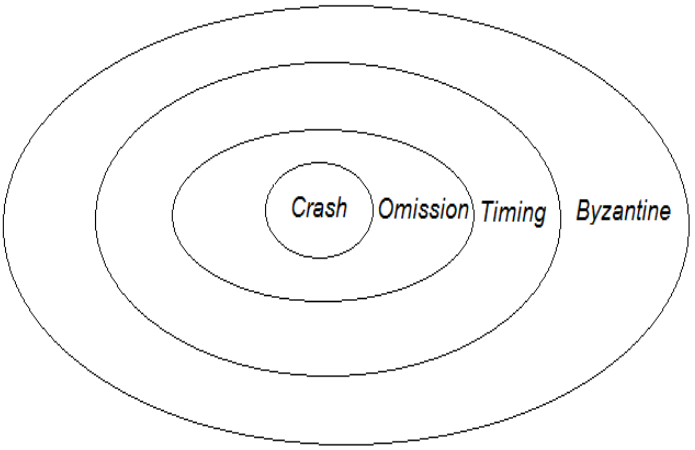

FIGURE 1.2 – Classses des défaillances dans un système.

1.4.2 Canaux de communication

1.4.2.1 Modèles de canaux de communication

Il existe plusieurs modélisations pour les canaux de communication : non fiable, équitable, quasi-fiable et fiable. Dans ces modèles il n'y a ni de création, ni de duplication de messages. Cependant, la caractérisation des canaux de communication en terme de perte de messages diffère entre ces modèles ([9],[24]) :

- ***Canaux non fiables (lossy channels)*** : Dans ce type de canaux, on suppose toujours la possibilité de la perte de messages ; tout message envoyé peut être perdu.
- ***Canaux équitables (fair-lossy channels)*** : Si un processus correct p envoie un message m une infinité de fois à un processus correct q alors le processus q reçoit une infinité de fois le message m.
- ***Canaux quasi-fiables (quasi-reliable channels)*** : Si un processus correct p envoie un message m à un processus correct q alors q reçoit finalement le message m.
- ***" Canaux fiables (reliable channels)*** : Si un processus p envoie un message m à un processus correct q alors q reçoit finalement le message m.

1.5 Problèmes d'accord

De nombreuses applications dans les systèmes distribués requièrent aux processus, participant à une tache commune, de s'entendre sur une décision. Il s'agit, à chaque fois, d'essayer d'établir un accord entre ces processus. D'où, ces processus doivent tous participer à l'exécution d'un protocole d'accord. Dans ce qui suit, nous allons présenter des problèmes d'accord très répondus dans les applications réparties.

1.5.1 La diffusion atomique (atomic broadcast)

La diffusion permet à un processus d'émettre un message à tous les membres du groupe. La diffusion d'un message m est réalisée par deux opérations de communication :

- ***Diffuse(***m***)*** : Cette opération permet d'émettre le message m à tous les processus du groupe.
- ***Délivre(***m***)*** : Cette opération permet la livraison du message m envoyé.

Avant de présenter la diffusion atomique, il est nécessaire de présenter la diffusion fiable [24].

1.5.1.1 La diffusion fiable (reliable broadcast)

La diffusion fiable garantit qu'un message émis à destination du groupe est, soit délivré par l'ensemble des processus corrects, soit par aucun d'entre eux. Les protocoles de diffusion fiable reposent sur des mécanismes de réémission des messages. Lorsqu'un processus reçoit un message pour la première fois, il le diffuse à tous les membres du groupe avant de le délivrer. Ces protocoles respectent les quatre propriétés suivantes :

- ***Validité*** : si un processus exécute *Délivre(*m*)*, alors un processus a exécuté *Diffuse(*m*)*.
- ***Terminaison*** : si un processus correct exécute *Diffuse(*m*)*, alors tous les processus corrects exécuteront *Délivre(*m*)*.
- ***Accord*** : si un processus correct exécute *Délivre(*m*)*, alors tous les processus corrects exécutent *Délivre(*m*)*.
- ***Integrité*** : un processus n'exécute *Délivre(*m*)* qu'une fois au plus.

1.5.1.2 La diffusion atomique

La diffusion atomique étend la diffusion fiable par l'ajout de la propriété de l'ordre total. La diffusion atomique garantit que tous les processus corrects délivrent le même ensemble de messages dans le même ordre. Elle doit respecter les propriétés suivantes :

- ***Validité*** : si un processus exécute *Délivre(m)*, alors un processus a exécuté *Diffuse(m)*.
- ***Terminaison*** : si un processus correct exécute *Diffuse(m)*, alors tous les processus corrects exécuteront *Délivre(m)*.
- ***Accord*** : si un processus correct exécute *Délivre(m)*, alors tous les processus corrects exécutent *Délivre(m)*.
- ***Integrité*** : un processus exécute *Délivre(m)* au plus une seule fois.
- ***Ordre total*** : si deux processus p_i et p_j délivrent deux messages m et m', alors p_i exécute *Délivre(m)* avant d'exécuter *Délivre(m')* si et seulement si p_j exécute *Délivre(m)* avant d'exécuter *Délivre(m')* .

1.5.2 La validation atomique

Le but de la validation atomique est d'assurer la propriété d'atomicité d'une transaction distribuée même en présence de pannes. L'atomicité, appelée aussi propriété de tout ou rien, garantit la validation de la transaction ou l'abandon de cette dernière. Dans le cas ou la transaction est validée, toutes les mises à jour faites par la transaction sur l'ensemble des sites sont rendues permanentes. Dans le cas où la transaction est abandonnée, ces mises à jour sont toutes annulées. La validation atomique garantit donc une terminaison uniforme de l'ensemble des branches qui composent une transaction distribuée. Chaque site accédé par une branche de transaction distribuée est appelé participant. Les sites participants se mettent d'accord sur l'issue d'une transaction en exécutant un protocole de validation atomique. Dans ce protocole, chacun révèle sa capacité à valider en émettant un vote. Si les participants votent " oui ", les mises à jour effectuées par la transaction deviennent permanentes même suite à un problème interne.

La validation atomique présente un problème d'accord entre les participants dans une transaction sur l'issue de cette dernière. Formellement, ce problème est défini par les quatre propriétés suivantes, qui doivent être satisfaites par tout protocole de validation atomique :

- ***Validité*** : si un participant décide de valider, alors tous les participants doivent avoir voté *oui*.
- ***Non-Trivialité*** : si tous les participants votent *oui* et aucune panne ne se produit, alors tous les participants décident de valider.
- ***Accord uniforme*** : tous les participants qui décident de valider ou d'abandonner, prennent la

même décision.
- ***Integrité*** : chaque participant décide au plus une fois.

1.5.3 La gestion de groupe

Le service de la gestion de groupe permet de fournir aux processus du groupe à un instant donné une liste des processus corrects qui appartiennent au groupe. Cette dernière doit tenir compte des événements qui ont apporté des modifications à la composition du groupe. Ces événements peuvent être des opérations volontaires telles que l'ajout des membres ou leur suppression, mais aussi des faits involontaires comme les défaillances des processus ou des canaux de communication. L'ensemble des processus qui constituent le groupe à un instant donné est appelé *la vue courante du groupe*. Chaque processus peut appeler trois primitives *joindre()*, *installer(vue)* et *quitter()* , qui permettent respectivement de joindre le groupe, installer la vue et quitter le groupe. Un processus notifie les autres membres du groupe des changements effectués en installant une nouvelle vue.

Dans [26], le problème de la gestion du groupe est spécifié par quatre propriétés :

- ***Validité*** : Si un processus correct installe une vue V, alors tous les membres corrects de V installeront V.
- ***Inclusion automatique*** : Si un processus installe une vue, alors il est un membre de cette vue.
- ***Accord*** : Si V_i est une vue installée par deux processus corrects p_i et p_j, alors cette vue contient les mêmes membres pour les deux processus.
- ***Integrité*** : Si V_1 et V_2 sont deux vues consécutives et p un processus qui est contenu dans V_1 tandis qu'il est exclu dans V_2, alors ce processus est suspecté par au moins un processus correct.

D'autres services de la gestion du groupe existent encore : le service de la synchronisation des vues et le service communication.

Un autre problème d'accord fondamental, que nous présentons dans la section suivante et qui est l'origine de notre problème étudié, est celui du Consensus (voir section 1.6).

1.6 Le Concensus

Tous les problèmes d'accord dans les systèmes répartis peuvent être réduits à un problème d'accord qui est au centre de tous ces problèmes, celui du Consensus [22]. Le but du Consensus est

d'établir un accord général entre les processus sur une certaine valeur même en présence de pannes. Dans le problème du consensus, chaque processus propose une valeur et doit décider une des valeurs initiales de manière unanime et irrévocable [20].

Plus formellement, un algorithme de consensus est un algorithme de décision pour lequel on suppose que : initialement, chaque processus p_i dispose d'une valeur initiale v_i et de deux primitives *Propose(v_i)* et *Décide(v)*. Tous les processus corrects doivent se mettre d'accord sur une valeur commune v parmi toutes celles qui ont été proposées. Les décisions des processus sont irrévocables et doivent assurer les propriétés suivantes :

- ***Accord*** : Deux processus corrects ne décident pas de valeurs différentes.
- ***Validité*** : Les valeurs décidées doivent avoir été proposées.
- ***Terminaison*** : Tout processus correct doit décider en un temps fini.
- ***Integrité*** : Tout processus décide au plus une fois.

Si les processus ne peuvent initialement proposer que 0 ou 1, on parle du ***consensus binaire***. Si par contre, l'ensemble des valeurs proposables contient plus de deux valeurs, alors on parle du ***consensus multivalué***.

1.6.1 Impossibilité du Consensus

Fischer, Lynch et Paterson ont prouvé dans [21], qu'il était impossible de trouver une solution entièrement déterministe dans le cas où, même un seul processus pouvait être défaillant. Ce résultat d'impossibilité provient du fait qu'il n'est pas possible de distinguer de manière absolument fiable un processus très lent, d'un processus défaillant. Dans ces conditions, toute solution entièrement déterministe peut être poussée à la faute, en l'obligeant à prendre une décision sans l'accord d'un processus très lent, qui sera considéré comme défaillant. Sans détection fiable des défaillances, il est alors impossible aux processus participant au Consensus de partager une vision cohérente du système, et donc de garantir les propriétés de ce problème.

1.6.2 Solutions pour contourner le résultat d'impossibilité FLP

Afin de contourner ce résultat d'impossibilité et de rendre le Consensus adaptable aux systèmes asynchrones, deux approches sont distinguées :

- Une approche qui consiste à augmenter le système asynchrone par l'ajout d'hypothèses telles que la synchronie partielle [18] ou la présence d'oracles capables de donner des informations

supplémentaires sur le système [14].
– Une autre approche qui consiste à affaiblir certaines propriétés du consensus pour contourner le résultat d'impossibilité. Cette approche était adoptée par [9], qui a proposé de relâcher la propriété de terminaison du consensus en faisant appel à un générateur aléatoire qui permettra de décider avec une probabilité qui tend vers 1. Dans le même sens, [8] et [2] ont proposé de relâcher la propriété d'accord du consensus pour décider plus d'une valeur.

1.6.3 Les variantes du concensus

Plusieurs variantes sont issues du problème original " le Consensus ". Ces variantes sont obtenues par la substitution de certaines propriétés du Consensus par d'autres.

1.6.3.1 Le Concensus probabiliste

Le Consensus probabiliste [4] relâche la propriété de la terminaison du consensus et la remplace par une autre propriété de terminaison propre au consensus probabiliste :

– ***Terminaison*** : Tout processus correct participant au Consensus décide une valeur avec une probabilité qui tend vers 1.

Cette solution fait appel à un générateur aléatoire dans le cas où les valeurs initiales proposées par les processus sont différentes. Ce générateur permettra de décider avec une probabilité qui tend vers 1.

1.6.3.2 Le Concensus uniforme

Dans le Consensus, La propriété de l'accord ne s'applique qu'aux processus corrects. D'où, il est possible qu'un processus décide une valeur, puis devienne défaillant, laissant alors aux autres processus la possibilité de décider une autre valeur. Le Consensus uniforme permet d'éviter ce cas de figure puisqu'il possède, en plus des propriétés de Terminaison, Validité et Intégrité, la propriété d'accord suivante :

– ***Accord uniforme*** : Deux processus (corrects ou non) ne décident pas de valeurs différentes.

1.6.3.3 Le Concensus strict

Malkhi et Reiter [27] considèrent une nouvelle variante du consensus, qui est le consensus strict. Dans ce dernier, c'est la propriété de la validité du consensus qui est substituée par la nouvelle propriété de validité suivante :

- ***Validité stricte*** : si tous les processus corrects proposent une valeur v, alors seule la valeur v peut être décidée par un processus correct.

Dans le modèle de défaillances byzantines, c'est la validité stricte qui est considérée à la place de la validité traditionnelle, car cette dernière présente un inconvénient de ne pas exclure le cas où la valeur décidée est une valeur initiale d'un processus byzantin. Le fait que tous les processus corrects doivent avoir la même valeur initiale pour opter à une décision, rend le consensus strict trop restrictif pour la résolution d'autres problèmes d'accord.

1.6.3.4 Le Concensus vectoriel

Le consensus vectoriel a été introduit par Doudou et al [19] , le but était de relâcher la restriction imposée par le consensus strict. Le consensus vectoriel permet à un ensemble de processus corrects, chacun ayant une valeur initiale, de décider sur un vecteur qui contient au moins $f + 1$ valeurs initiales de processus corrects, en dépit de présence de processus byzantins. Dans cette variante du consensus, c'est aussi la propriété de validité traditionnelle qui est substituée, mais cette fois, par la propriété de la validité suivante :

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

1.6.3.5 Le k-Accord (k-set-agreement)

Le problème du k-accord a été introduit par S. CHAUDHURI [8] . Dans ce problème, c'est la propriété de l'accord du consensus qui est relâchée en autorisant la décision de plus d'une valeur ([8],[2]) . Les propriétés de la validité, l'intégrité et la terminaison du consensus doivent être vérifiées, en plus de la propriété d'accord propre au k-accord suivante :

- ***k-Accord*** : Au plus k valeurs différentes sont décidées dans l'ensemble du système.

Ce problème dépond du nombre k de valeurs qui peuvent être décidées et du nombre maximal t des défaillances des processus :

- **($k = 1$)**, ce problème est équivalent au Consensus.
- **($k \leq t$)**, il a été montré dans ([5],[23],[32]) que ce problème du k-accord est impossible à résoudre de manière déterministe dans un milieu asynchrone.
- **($k > t$)**, ce problème est simple à résoudre, il suffit alors que k processus diffusent leurs valeurs qu'ils proposent et qu'un processus décide la première valeur qu'il reçoit.
- **($k = n - 1$)**, le problème est appelé accord ensembliste (n est le nombre de processus).

1.7 Conclusion

Dans ce chapitre, nous avons présenté des notions, que nous trouvons, de base sur les systèmes distribués soumis aux défaillances et aux problèmes d'accord. Dans ces derniers, nous avons cité quelques problèmes fondamentaux qui peuvent être réduits au problème du consensus dont le problème étudié dans notre mémoire est une variante. Nous nous intéressons dans ce mémoire au problème du k-Accord avec la présence de processus byzantins.

Chapitre 2

Le problème du k-Accord dans le cas de déffaillances par crash

2.1 Introduction

Le problème du k-Accord est une simple généralisation du problème ordinaire du consensus. Dans ce dernier, la contrainte de décider sur une seule et même valeur est imposée. Dans le cas du k-Accord, cette condition n'est plus exigée, il suffit aux processus de limiter leurs décisions à un nombre k de valeurs distinctes.

La motivation originale de ce problème était purement mathématique ; il était intéressent d'essayer de comprendre comment les résultats obtenus dans le consensus changent quand les pré requis de ce problème changent. Malheureusement, même ce problème présente une autre impossibilité dans les systèmes asynchrones. Dans ce chapitre, nous allons présenter ce problème et les différentes approches utilisées pour contourner cette impossibilité.

2.2 Le k-Accord

Le k-Accord a été introduit par S. Chaudhuri [8] qui, en considérant uniquement le modèle de défaillance par crash, étudiait la manière dont le nombre (k) de choix permis aux processus est lié au nombre maximal des processus qui peuvent être défaillants par crash (t). Il s'agit d'une version généralisée du concensus dans laquelle les processus proposent chacun une valeur et décident chacun une valeur proposée par l'un d'eux, la contrainte étant que le cardinal de l'ensemble des valeurs

décidées soit inférieur ou égal à k. Formellement, l'opération propose(v) est disponible et invoquée sur chaque site, et retourne, à la terminaison de l'algorithme, une valeur que nous appelons valeur décidée.

Le problème peut être défini comme suit : Chacun des N processus définissant le système démarre avec sa propre valeur (appelée "valeur proposée "). Chaque processus correct doit décider d'une valeur (***terminaison***), de telle sorte que la valeur décidée soit une valeur proposée (***validité***) et aux plus k valeurs différentes sont décidées (***accord***).

2.3 Un résultat clé dans les systèmes synchrones

2.3.1 Le modèle synchrone

Dans ce modèle de calcul, chaque exécution est composée d'une séquence de rondes. Celles-ci sont identifiées par des entiers successifs 1, 2, etc. Pour les processus, le numéro de la ronde courante apparait comme une variable globale dont le progrès global entraîne leurs progrès local.

Durant une ronde, un processus broadcaste un message, reçoit des messages, et enfin exécute un calcul local. La propriété fondamentale de synchronie qu'un système synchrone fournit aux processus et la suivante : un message envoyé durant une ronde r est reçu par le processus destinataire durant la même ronde r. Durant une ronde, si un processus se crashe, lors de l'envoi d'un message, un sous-ensemble arbitraire (non connu à l'avance) de processus reçoivent ce message.

2.3.2 Un résultat principal

Le problème du k-Accord peut être toujours résolu dans un système synchrone. Le résultat principal est pour le nombre minimal de rondes (R_t) qui sont nécessaires pour les processus non-défaillants afin de décider dans le pire des scénarios (ce scenario se produit quand exactement k processus se crashent dans chaque ronde).

Il a été démontré dans [10] que $(t/k + 1)$ est le nombre de rondes nécessaire pour permettre aux processus corrects de décider. Un très simple algorithme qui respecte cette borne inferieure est décrit dans la figure 2.1.

Function k-set_agreement(v_i)
$est_i \leftarrow v_i$

(1) **when** $r = 1, 2, ..., (\frac{t}{k}) + 1$ **do** ;
(2) **begin_round**
(3) *send* PROPOSE(est_i) to all ;
(4) $est_i \leftarrow$ min($\{est_j$ values received during the current round $r\}$) ;
(5) **end_round** ;
(6) return(est_i)

FIGURE 2.1 – Un simple k-set protocole synchrone pour le k-Accord.

2.4 Des résultats clés dans les systèmes asynchrones

2.4.1 Un résultat d'impossibilité

Dans les systèmes synchrones où les processus sont sujets aux défaillances par crash, le problème du k-Accord peut toujours être résolu, quelque soit la valeur de t par rapport à k.

Ce problème peut aussi trivialement être résolu dans les systèmes asynchrones soumis même à une seule panne lorsque k > t . Une étape de communication dans un protocole est comme suit : (1) k processus sont arbitrairement choisis avant l'exécution,(2) chacun de ces processus envoie sa valeur à tous les processus, (3) un processus décide la première valeur qu'il reçoit.

Cependant, il a été montré qu'il n'y a pas de solution dans ces systèmes dès que ($k \leq t$)([5],[23],[32]). Une nouvelle impossibilité est alors présenté dans cette variante du concensus, ce qui donne naissance à d'autres recherches pour surmonter cette impossibilité.

2.4.2 Contourner l'impossibilité

Afin de contourner l'impossibilité de résoudre le problème du k-Accord dans les systèmes asynchrones sujets aux défaillances. Plusieurs approches ont été proposées, nous distinguons quatre approches principales :

- Approche par randomisation.
- Approche par détecteur de défaillances.
- Approche par contraintes par condition.
- Approche par ajout d'hypothèses temporelles.

2.4.2.1 La randomisation

La randomisation consiste à augmenter les systèmes asynchrones par des oracles aléatoires. Un oracle est un ensemble de modules, chacun attaché à un processus. Chaque valeur x ($n > x > 1$) a une probabilité de $1/n$ d'être retournée quand un processus invoque la primitive random. Dans le contexte de systèmes distribués asynchrones augmentés par des oracles aléatoires, le problème du k-Accord est formellement défini par les propriétés suivantes :

- *Terminaison* : Avec une probabilité de 1, chaque processus correct décide finalement une valeur.
- *Validité* : si un processus correct décide v, alors v a été proposée par quelque processus.
- *Accord* : Au plus k valeurs différentes sont décidées par des processus.

La randomisation a été combinée avec la diffusion fiable dans [28] pour surmonter l'impossibilité du k-Accord. Les processus utilisent la diffusion fiable pour diffuser les valeurs qu'ils proposent. La randomisation est utilisée pour assurer la terminaison avec une probabilité de 1. Des tours asynchrones sont utilisés pour faire converger les processus, et une valeur par défaut est utilisée lors de chaque tour pour restreindre l'ensemble des valeurs proposées à un ensemble d'aux plus k valeurs.

2.4.2.2 Ajout d'hypothèses temporelles

Le modèle asynchrone peut être augmenté en lui ajoutant des hypothèses temporelles. On obtient de nouveaux modèles temporels : partiellement synchrones [18] ou asynchrones temporisés [11]. Dans les deux cas, le problème du k-Accord peut être résolu. Les modèles partiellement synchrones supposent de nouvelles bornes sur les délais de communication et la vitesse des processus qui sont satisfaites à partir d'un certain temps appelé " temps global de stabilisation ". Les modèles asynchrones temporisés sont caractérisés par une alternance entre des périodes où le système se comporte de manière totalement asynchrone et d'autres périodes où le système se comporte d'une manière synchrone.

2.4.2.3 Détecteurs de défaillances

Les oracles sont des mécanismes mis à disposition des processus et qui leur donnent de l'information sur le système pour leur permettre de résoudre un problème donné. Chandra et Toueg [14] ont augmenté le modèle asynchrone, pour surmonter le résultat d'impossibilité, en introduisant la notion de détecteur de défaillances qui sont des oracles chargés de détecter les défaillances d'une manière suffisamment fiable pour permettre de trouver une solution au problème du Consensus.

Ils ont proposé des détecteurs de défaillances où chaque site possède son propre détecteur de

défaillance. Ce dernier est non fiable, c'est-à-dire qu'un processus peut être considéré défaillant alors qu'il fonctionne encore. Des processus peuvent être ajoutés ou supprimés de la liste des processus suspectés d'être en panne au fur et à mesure du temps. Cette liste peut être différente sur chacun des sites. Chandra et Toueg proposent une catégorisation des détecteurs de défaillances qui permet d'analyser les problèmes qui peuvent être résolus ou non en fonction du type de détecteur disponible et du nombre maximum de défaillances possibles.

Chandra et Toueg définissent huit classes de détecteurs de défaillances en caractérisant chacune d'entre elles par une propriété de complétude et une propriété de précision :

La propriété de complétude : Elle définit des contraintes concernant la détection des processus réellement arrêtés. Elle définit deux propriétés :

- ***Complétude forte*** : il existe un instant à partir duquel tout processus défaillant est définitivement suspecté par tout processus correct.
- ***Complétude faible*** : il existe un instant à partir duquel tout processus défaillant est définitivement suspecté par au moins un processus correct.

La propriété d'exactitude : Elle vise à limiter les suspicions erronées que peut commettre un détecteur de défaillances. Elle définit quatre propriétés :

- ***Précision forte*** : aucun processus correct n'est jamais suspecté.
- ***Précision faible*** : il existe au moins un processus correct qui n'est jamais suspecté.
- ***Précision ultime forte*** : il existe un instant à partir duquel tout processus correct n'est suspecté par aucun processus correct.
- ***Précision ultime faible*** : il existe un instant à partir duquel au moins un processus correct n'est suspecté par aucun processus correct.

Dans le modèle asynchrone à échange de messages, la recherche du détecteur minimal pour le k-accord lorsque $1 < k < n$ reste un problème ouvert.

La figure 2.2 illustre les huit classes de détecteurs de défaillances :

Chandra et Toueg s'intéressent à un modèle dans lequel tous les processus sont reliés entre eux deux à deux par un réseau fiable et ne prennent en compte que les pannes franches. Malkhi et Reiter ont été les premiers à étendre la notion de détecteurs de défaillances au modèle de défaillances byzantines [27].

	Justesse forte	Justesse faible	Justesse ultimement forte	Justesse ultimement faible
Complétude forte	Parfait P	Fort S	Ultimement parfait $\diamond$P	Ultimement Fort$\diamond$S
Complétude faible	Q	Faible W	$\diamond$Q	Ultimement faible $\diamond$W

FIGURE 2.2 – Les classes de détecteurs de défaillances.

2.4.2.4 Approche basée sur les conditions

L'approche basée sur les conditions consiste à chercher certaines combinaisons des valeurs d'entrée d'un problème distribué donné. Cette approche a pour but d'étudier les conditions qui limitent les entrées du k-Accord qui font le problème dans un système asynchrone où t processus peuvent se bloquer.

Il est souvent le cas dans la pratique que certaines combinaisons des valeurs d'entrée du processus occurrent plus fréquemment que d'autres. Plus précisément, un vecteur d'entrée contient les valeurs proposées par les processus dans une exécution. Une condition C est un ensemble de vecteurs d'entrée, chacun représentant une combinaison commune des entrées à un problème. Si un protocole résout le k-Accord pour C, puis à chaque fois que le vecteur d'entrée appartient à C, tous les processus corrects décident. La solution doit être indulgente dans le sens que si les processus corrects décident alors que le vecteur d'entrée n'appartient à la condition, ils ne décident pas plus de k valeurs.

2.5 Conclusion

Le problème du k-Accord généralise le problème du consensus en ce sens qu'il ne limite pas l'ensemble des valeurs de décision à une seule mais à k valeurs, il est de ce fait moins dur à résoudre. Dans un système asynchrone composé de n processus et où au plus t d'entre eux peuvent être défaillants, le problème du k-Accord peut être facilement résolu si $k > t$. Il a par ailleurs été démontré qu'aucune solution déterministe n'existe lorsque $k \leq t$.

Dans ce chapitre nous avons présenté le problème du k-Accord ainsi que les différentes approches utilisées pour contourner l'impossibilité du k-Accord.

Chapitre 3

Le k-Accord Byzantin avec $(k > t)$

3.1 Introduction

La solvabilité du problème du k-Accord sujet à des défaillances par crash dans les systèmes asynchrones à échange de messages, dans le cas où $(k > t)$, est sujette de plusieurs travaux puisque ce problème admet des solutions triviales, comme nous l'avons vu dans le chapitre 2. Cependant, on ne trouve pas de travaux qui résolvent le même problème dans les systèmes asynchrones sujet à des défaillances Byzantines dans le même cas. Pour résoudre ce problème, la condition de validité a un impact profond sur le moment où le problème est solvable.

Le but de ce chapitre est de répondre à la première question que nous avons posée dans l'introduction. Nous allons discuter sur la validité dans le problème du k-Accord, présenter le problème du k-Accord Byzantin, pour finir par proposer deux protocoles résolvant ce problème dans le cas où $(k > t)$, le premier utilise la diffusion fiable et le deuxième utilise l'authentification.

3.2 Modèle du système

Nous considérons un système constitué d'un ensemble fini de $n > 1$ processus notés de p_1 à p_n qui communiquent par échange de messages, dans un réseau complètement connecté , via des canaux fiables. Le système est asynchrone, ce qui signifie qu'il n'y a pas de limites sur les délais de traitement de messages ou de communication entre processus. Jusqu'à t processus peuvent présenter un comportement byzantin. Un processus byzantin est un processus qui se comporte arbitrairement : il peut par exemple tomber en panne, ne pas parvenir à envoyer ou recevoir des messages, envoyer

des messages arbitraires ou commencer dans un état arbitraire.

En outre, les processus byzantins peuvent s'entendre à avoir un comportement malveillant dans le système. On suppose qu'ils ne contrôlent pas le réseau. Cela signifie qu'ils ne peuvent pas corrompre les messages envoyés par des processus non-byzantins. Un processus qui présente un comportement byzantin est appelé défaillant. Sinon, il est correct ou non-défaillant.

3.3 Sur la propriété de la validité

Comme le problème du k-Accord est une simple généralisation du problème du consensus où la propriété de l'accord est substituée par la propriété du k-accord, et puisque ce qui nous intéresse dans cette section est la propriété de validité, nous considérons ici le problème du consensus. Ce problème est formellement défini en termes des propriétés de validité, accord et terminaison. Les deux premières propriétés sont des propriétés de sureté, à savoir, les propriétés qui disent que certaines "mauvaises choses" ne peuvent pas se produire, tandis que la troisième est une propriété de vivacité ; c'est une propriété qui définit "les bonnes choses" qui doivent se produire [3].

Au moment que propriétés de l'accord et la terminaison sont demeurées généralement les mêmes pour le consensus multivalué, certains articles utilisent la propriété suivante de validité ([3], [18],[25]).

– **Validité 1** : Si tous les processus corrects proposent la même valeur v, alors tout correct processus qui décide, décide v.

D'autres utilisent la suivante ([16],[17], [6]) :

– **Validité 2** : Si un processus correct décide v, alors v a été proposée par un certain processus.

Les deux propriétés sont un peu faibles. Validité 1 ne dit rien à propos de ce qui est décidé lorsque les processus corrects ne proposent pas tous la même valeur v, tandis que validité 2 ne dit rien sur ce qui est la valeur décidée (par exemple, est-elle la valeur proposée par les processus corrects s'ils proposent tous la même valeur, ou bien, une valeur proposée par un processus défaillant ?). Une définition qui donne plus de détails sur ce qui est décidé a également été proposée [13]. La définition comporte trois propriétés de validité :

– **Validité 1** : Si tous les processus corrects proposent la même valeur v, alors tout correct processus qui décide, décide v.
– **Validité 2A** : Si un processus correct décide v, alors v a été proposée par un certain processus

ou $v = \perp$.
- **Validité 3** : Si une valeur v est proposée uniquement par des processus défaillants, alors aucun processus correct ne décide v.

Les deux premières propriétés sont essentiellement les propriétés de validité déjà introduites, à l'exception que Validité 2A permet la décision protective d'une valeur $\perp$ n'appartenant pas à V (l'ensemble des valeurs proposées). La troisième propriété est inspirée de la définition originale dans le contexte de la métaphore des "généraux byzantins" utilisée dans le papier classique par Lamport et al [26] qui considère un système synchrone. La définition était : (1) Tous les généraux fidèles décident sur le même plan d'action, (2) Un petit nombre de traîtres ne peuvent pas provoquer les généraux fidèles à adopter un mauvais plan.

Parmis toutes ces propriétés de validité, quelle est la propriété qui peut etre adoptée pour le k-Accod Byzantin, ou bien est-il necessiare de définir une nouvelle propriété de validité ?

3.4 Le k-Accord Byzantin avec k > t

Avec la présence des comportements byzantins qui ne spécifient aucune contrainte sur la manière dont un processus peut devenir défaillant, le but est d'éviter autant que possible que la valeur finale décidée provienne d'une proposition byzantine. Une autre définition pour la validité doit être alors utilisée. On appelle alors k-accord Byzantin le problème du k-accord (avec la nouvelle définition de la validité) en présence de processus byzantins.

Formellement, le k-Accord Byzantin est défini par les proporiétés suivantes :
- ***Accord :*** Au plus k valeurs sont décidées.
- ***Terminaison :*** Tout processus correct décide une valeur.
- ***Validité :*** La valeur décidée est une valeur proposée.

3.5 Proposition d'un protocole résolvant le problème du k-Accord Byzantin sans signature

Dans cette section, nous proposons un protocole (figure 3.2) qui résout le problème du k-Accord Byzantin quand $k > t$. Le protocole proposé utilise des primitives de la diffusion fiable. Pour se faire, nous allons d'abord présenter un simple algorithme de diffusion fiable.

operation R_broadcast(v_i)
(1) broadcast INIT(v_i, i) ;

RB-delivery task from p_j :
(2) **wait until** (INIT(v, j) delivered from p_j or
ECHO(v, j) delivered from $((n + t)/2)$ different processes or
READY(v, j) delivered from $((n - 2t)$ different processes$)$;
(3) broadcast ECHO(v, j) ;
(4) **wait until** (ECHO(v, j) delivered from $((n + t)/2)$ different processes or
READY(v, j) delivered from $((n - 2t)$ different processes$)$;
(5) broadcast READY(v, j) ;
(6) **wait until** (READY(v, j) delivered from $((n - t)$ different processes$)$;
(7) R_deliver(v) at p_i as the value R_broadcast by p_j

FIGURE 3.1 – Un simple algorithme de diffusion fiable

3.5.1 Un simple algorithme de diffusion fiable (Reliable-Broadcast)

La figure 3.1 présente un simple algorithme implémentant la diffusion fiable dans un système asynchrone sujet à des défaillances Byzantines dans lequel $t < n$ / 3. Cet algorithme, inspiré de [7], utilise trois types de messages (INIT(v_i, i), ECHO(v_i, i) et READY(v_i, i)) et deux opérations R_broadcast() et R_deliver().

Ce protocole de diffusion fiable assure les propriétés suivantes :

- *Accord* : il n'existe pas deux processus corrects qui R_délivrent différents messages à partir de n'importe quel p_j.
- *Terminaison* : si l'expéditeur est correct, tous les processus corrects inéluctablément R_délivrent son message.
- *Uniformité* : si un processus correct R-délivre un message de p_j (possible qu'il est défaillant), tous les corrects R_délivrent un message de p_j.

Quand un processus p_i invoque l'opération R_broadcast(v_i), il diffuse le message INIT(v_i, i). Quand p_i délivre un message INIT(v, j) de p_j ou ECHO(v, j) de $((n+t)/2)$ processus ou READY(v, j) de $(n - 2t)$ processus (lignes 1, 2), il diffuse ECHO(v, j) (ligne 3). Quand p_i a délivré un message ECHO(v, j) délivré de $((n+t)/2))$ processus distincts ou READY(v, j) de $(n-2t)$ processus distincts (Ligne 4), il diffuse READY(v, j) (ligne 5), puis p_i attend jusqu'à ce qu'il délivre le même message READY(v, j) de $(n - t)$ processus distincts. Lorsque cela se produit, il R_délivre la valeur v comme la valeur R_broadcast par p_j (lignes 6,7).

3.5.1.1 Preuve de correction du protocole

Lemme 1 (Non-Duplicité) *pas de deux processus corrects qui R-délivrent différents messages à partir de n'importe quel processus* p_j.

Preuve Considérons une exécution où un processus p_j invoque l'opération R_broadcast(). Soient p_i et p_k deux processus corrects qui R_délivrent de p_j les valeurs v et v', respectivement. Il résulte de l'algorithme que p_i a délivré $(n-t)$ messages READY(v, j) et p_k a délivré $(n-t)$ messages READY(v', j) (ligne 6). Comme $(n-t) > (n-2t) > t)$, il s'ensuit qu'il existe au moins un processus correct qui a envoyé READY(v, j) à p_i et READY(v', j) à p_k (ligne5). Comme il est correct, ce processus a envoyé la même valeur $v = v'$ à la fois à p_i et p_k, ce qui prouve le lemme. $\square_{Lemme\ 1}$

Lemme 2 (Terminaison) *Si l'expéditeur est correct, tous les processus corrects inéluctablément R_délivrent son message.*

Preuve Considérons une exécution où un processus p_i invoque l'opération R_broadcast(v_i). Si p_i est correct, tous les processus corrects finalement délivrent un message INIT(v, i) de p_i à la ligne 2. Par conséquent, ils diffusent le message ECHO(v, i). Après cela, tout les processus corrects finalement délivrent un message ECHO(v, i) d'au moins $(n-t)$ processus distincts et ils diffusent tous le message READY(v, i). Par conséquent, tout processus correct délivre finalement READY(v, i) d'au moins $(n-t)$ des processus distincts et R_délivre v. $\square_{Lemme\ 2}$

Lemme 3 (Uniformité) *Si un processus correct R_délivre un message de* p_j *(possiblement défaillant), alors tout les processus corrects R_délivrent un message de* p_j.

Preuve Preuve Soit p_i un processus correct qui R_délivre v de p_j. Il résulte de la ligne 6 que p_i a délivré un message READY(v, j) de $(n-t)$ processus distincts. Comme $n > 3t$, au moins $(n-2t) \geq t+1$ de ces processus sont corrects, ce qui signifie que tout processus correct a délivré le message READY(v, j) d'au moins $(n-2t)$ processus distincts. Il en résulte alors de la ligne 4 que tout processus correct a délivré un message READY(v, j) d'au moins $(n-2t)$ processus distincts et diffuse READY(v, j) à la ligne 5. Enfin, comme tout correct processus diffuse le message READY(v, j), il s'ensuit que tout processus correct délivre le message READY(v, j) de $(n-t)$ processus distincts et par conséquent R_délivre v comme la valeur R_Broadcast par p_j. $\square_{Lemme\ 3}$

Theorème 1 *L'algorithme de la figure 3.1 implémente la diffusion fiable dans un système distribué asynchrone sujet à des défaillances Byzantines, où* $n \geq 3t+1$.

Preuve La preuve découle directement de 1, 2et 3. $\square_{Theorme\ 1}$

3.5.2 Le protocole résolvant le problème du k-Accord Byzantin sans signature

Dans cette section, nous présentons une solution pour le problème de ce chapitre qui est la résolution du k-Accord Byzantin dans le cas où $(k > t)$. Pour cela nous allons utiliser les primitives de l'algorithme de la diffusion fiable présenté dans la section précédante. Les processus utilisent la diffusion fiable pour diffuser les valeurs qu'ils proposent. Notre protocole est un protocole du k-Accord Byzantin sans signature, ce qui veut dire que les messages qui transitent entre les processus ne sont pas signés.

3.5.2.1 Description du protocole

Le protocole est décrit sur la figure 3.2.

Chaque processus p_i de l'ensemble A_i commence sa participation dans le protocole en invoquant la fonction k_Accord_Byzantin(v_i) qui retourne une valeur décidée. Un processus p_i obtient sa valeur de décision v quand il invoque *return* (Ligne 04). L'exécution de cette invocation termine la participation de p_i dans le protocole.

Chaque processus gère une variable locale (est_i) qui représente l'estimation courante de sa valeur de décision. Initialement, est_i est fixée à v_i (la valeur proposée par p_i). Pour éviter le blocage d'un processus pour toujours (ie, attendre une valeur d'un processus qui a déjà été décidée), un processus qui décide utilise une diffusion fiable (lignes 03 et 04) pour diffuser sa valeur de décision.

Chaque processus p_i de l'nsemble A_i propose une valeur pour qu'elle soit décidée, il R_Broadcast un message PROPOSE(p_i, est_i) contenant son identité ainsi que l'estimation courante de sa valeur de décision (ligne 03).

Le processus qui a proposé une valur v, attend que cette dérnière soit R_délivrée par un processus de l'ensemble A_i (ligne 03) pour décider v (ligne 04).

Ce protocole n'est pas tolérant aux intrusions , mais il garantit le fait que les processus byzantins ne peuvent pas envoyer des valeurs différentes à des processus corrects ou bien envoyer à quelques processus uniquement et laisser les autres, grace aux primitives de la diffusuion fiable. Dans cette

Function k_Accord_Byzantin(v_i)
$est_i \leftarrow v_i$

(1) **let** A_i = a set of processes with $A_i = \{p_1, p_2, .., p_k\}$;
(2) **if** $p_i \in A_i$ **then** *R_broadcast* PROPOSE(p_i, est_i) ; ;
(3) **wait until** the first PROPOSE(p_k, v) message *R_delivered* from p_k with $(p_k \in A_i)$
(4) return(v)

FIGURE 3.2 – Un protocol pour le k-Accord Byzantin avec $(k > t)$

dérnière, on ne peut pas avoir deux processus qui R_délivrent deux valeurs différentes de n'importe quel processus p_i de A_i, et aussi, si un seul processus R_délivre une valeur v, alors v sera R_délivée par tous les processus corrects de l'ensemble A_i,

Ce protocole garantit les trois proporiété du k-Accord Byzantin : *accord*, *terminaison* et *validité*.

3.5.2.2 Preuve de correction du protocole

Lemme 4 (Accord) *Au plus k valeurs sont décidées.*

Preuve Considérons une exécution où les k processus de l'ensemble A_i invoque l'opération R-broadcastPROPOSE(p_i, est_i) (ligne 02) donc, nous avons k valeurs proposées. La ligne 03 montre que le message de proposition est soit R_délivré par tous les processus corrects, par conséquent, la valeur contenue dans ce message est décidée (ligne 04), soit ce message n'est R_délivré par aucun processus correct et la valeur contenue dans ce message n'est pas décidée, ce qui montre qu'on ne pourra jamais décider plus de k valeurs. D'où, au plus k valeurs sont décidées. , ce qui prouve le lemme. $\square_{Lemme\ 4}$

Lemme 5 (Terminaison) *Chaque processus correct doit décider d'une valeur.*

Preuve De la ligne 02, chaque processus de l'ensemble A_i doit participer dans le protocole en proposant une valeur. Soit p_i un processus de l'ensemble A_i qui invoque l'opération *R_broadcast* PROPOSE(p_i, est_i) (ligne 02). Il résulte de la ligne 03 que si un processus correct R_délivre le message PROPOSE(p_i, est_i) alors tous les processus corrects R_délivrent ce messages (propriété de la diffusion fiable) (ligne 03) pour que la valeur soit décidée.D'où, chaque processus correct doit décider d'une valeur. $\square_{Lemme\ 5}$

Lemme 6 (Validité) *La valeur décidée est une valeur proposée.*

Preuve Soit p_i un processus correct qui R_délivre un message PROPOSE(p_i, est_i) de p_j. Ce message doit forcément être R_ broadcasté (ligne 02 et 03). Puisque l'abstraction de la diffusion fiable oblige tous les processus à R_délivrer le message de p_j R_délivré par p_i, alors tous les processus corrects R_délivrent le message PROPOSE(p_i, est_i) de p_j ,il résulte des lignes 03 et 04 que cette valeur est décidée .D'où, la valeur décidée est une valeur proposée. $\square_{Lemme\ 6}$

Theorème 2 *L'algorithme de la 3.2 implémente le k-Accord Byzantin dans un système distribué asynchrone sujet à des défaillances byzantines, où $k > t$.*

Preuve La preuve découle directement du lemme 4, le lemme 5 et le lemme 6. $\square_{Theorme\ 2}$

3.6 Proposition d'un protocole du k-Accord Byzantin avec authentification

Dans cette section, nous proposons un autre protocole résolvant le k-Accord Byzantin quand $(k > t)$, mais cette fois en utilisant l'authentification.

3.6.1 Authentification et signature des messages

Notre modèle suppose l'existence de moyens d'authentification. Un processus Byzantin peut disséminer une valeur fausse (i.e. différente de celle qu'il aurait envoyée s'il était correct). Afin de prévenir une telle dissémination, le protocole utilise des certificats : ce sont des signatures au niveau applicatif (de type cryptographie à clé publique, comme les signatures RSA). Une implémentation simple consiste à inclure un ensemble de messages signés comme certificats. Prenons le cas d'un processus p souhaitant relayer la valeur v reçue d'un processus q. Le processus q signe son message et l'envoie à p. Le processus p peut prétendre ne pas avoir reçu la valeur de q (ce n'est pas vérifiable) mais s'il relaie une valeur, c'est nécessairement la valeur signée par q. Ceci suppose que nos processus byzantins ne sont pas capables de falsifier les signatures. Supposons maintenant que p doit relayer à tous les processus la valeur qu'il a reçue majoritairement (parmi toutes les valeurs reçues). Le certificat que p joint consiste en l'ensemble des messages signés qu'il a reçu (de manière à ce que chaque processus puisse vérifier que p a bien envoyé la valeur majoritaire).

3.6.2 Description du protocole

Le protocole est décrit dans la figure 3.3 . Chaque processus p_i de l'ensemble A_i commence sa participation au protocole en appelant la fonction du k-Accord Byzantin Authentifié(v_i) qui retourne une valeur décidée v (ligne6).

Cette fois, notre protocole utilise l'authentification. Nous supposons donc que tous les messages de propositions qui transitent entre les processus sont des messages signés. Cela veut dire que les processus byzantins ne peuvent pas modifier ces messages.

Un processus p_i obtient sa valeur de décision v quand il invoque *return* (Ligne 6) qui termine la participation de p_i dans le protocole.

Pendant l'exécution du protocole, le processus p_i utilise la variable est_i pour stocker la valeur qu'il pense décider. Ainsi, au commencement du protocole, chaque p_i processus correct aura sa variable est_i initialisée à la valeur qu'il propose v_i ; à la fin de l'exécution du protocole, la variable est_i correspondra à la valeur décidée.

Dans ce protocole, les processus utilisent une diffusion simple pour diffuser leurs valeurs (ligne2). Chaque processus p_i de l'ensemble A_i propose une valeur pour la décider. Il R_Broadcast donc un message PROPOSE(p_i, est_i) ,(ligne2), contenant son identité ainsi que l'estimation courante de sa valeur de décision.

Quand la valeur initiale de p_j est reçue par p_i, elle est enregistrée dans la variable $aux_i[j]$. Initialement, $aux_i[j]$ est initialisée à $\perp$ (ligne3).

Après l'enregistrement de la valeur initiale de p_j dans $aux_i[j]$, le processus p_i broadcast un message RELAY$(p_i, aux_i[j]$ (lignes 3 et 4). Cette étape permet de diffuser davantage la valeur proposée par un processus. Si un processus a reçu cette valeur, il la fait suivre à tous les autres processus. De cette façon, tous les processus échangent les valeurs de propositions qu'ils ont reçues. Un processus attend $n - t$ messages RELAY$(*, *)$ contenant la même valeur v (ligne 5) pour décider v (ligne6).

Grace au principe d'authentification, les processus byzantins ne peuvent pas falsifier les messages qu'ils envoient aux autres processus, car cela peut facilement être détecté, à cause des messages RELAY qui doivent contenir la même signature pour une valeur proposée de n'importe quel processus.

Function k_Accord_Byzantin_Authentifié(v_i)
$est_i \leftarrow v_i$, $aux_i[1..k] \leftarrow \perp$

(1) **let** A_i = a set of processes with $A_i = \{p_1, p_2, .., p_k\}$;
(2) **if** $p_i \in A_i$ **then** *broadcast* PROPOSE(p_i, est_i) ;
(3) **upon** *receipt* of PROPOSE(p_j, est)**then** $aux_i[j] \leftarrow est$;
(4) *broadcast* RELAY$(p_i, aux_i[j])$;
(5) **wait until** at least $(n - t)$ RELAY$(*, *)$ messages, carrying the same value, received from distinct processes ;
(6) return(v)

FIGURE 3.3 – Un protocole du k-Accord Byzantin avec authentification où $(k > t)$

3.6.2.1 Démonstration du protocole

Lemme 7 (Accord) *Au plus k valeurs sont décidées.*

Preuve Considérons une exécution où les k processus de l'ensemble A_i invoquent l'opération *broadcast* PROPOSE(p_i, est_i) (ligne 02) donc, nous avons k valeurs proposées. Lors de la réception d'un message PROPOSE(p_i, est_i), les processus p_j de l'ensemble A_i échangent des messages RELAY(lignes 03 et 04) contenat leurs identités ainsi que la variable $aux_i[j]$. Si p_i obtient $(n - t)$ messages RELAY contenant la même valeur signée , alors cette valeur est décidée (lignes 05 et 06), sinon, la valeur ne pourra pas être décidée. ce qui montre qu'on ne pourra jamais décider plus de k valeurs. D'où, au plus k valeurs sont décidées. , ce qui prouve le lemme. $\square_{Lemme\ 7}$

Lemme 8 (Terminaison) *Chaque processus correct doit décider d'une valeur.*

Preuve De la ligne 02, chaque processus de l'ensemble A_i doit participer dans le protocole en proposant une valeur. Soit p_j un processus de l'ensemble A_i qui invoque l'opération *R_broadcast* PROPOSE(p_j, est_j) (ligne 02). Tous les processus p_j recevant le message de proposition doivent échanger des messages RELAY$(p_j, aux_i[j])$. La décision n'est obtenue qu'après avoir reçu les messages RELAY contenant la même valeur signée de $(n-t)$ processus qui représentent tous les processus crrects. D'où, chaque processus correct doit décider d'une valeur.

$\square_{Lemme\ 8}$

Lemme 9 (Validité) *La valeur décidée est une valeur proposée.*

Preuve Considérons une exécution où une valeur v est décidée. Il résulte des ligns 05 et 06 que v est contenue dans $(n-t)$ messages RELAY provenant de $(n-t)$ processus disticts . Cette valeur est alors une valeur contenue dans un message de proposition d'un processus de l'ensemble A_i (lignes 02 et 03), puisque quand le message de proposition contenant la valeur initiale de p_j est reçu par p_i, cette valeur est enregistrée dans la variable $aux_i[j]$ (ligne 03). D'où, la valeur décidée est une valeur proposée. $\square_{Lemme\ 9}$

Theorème 3 *L'algorithme de la 3.3 implémente le k-Accord Byzantin dans un système distribué asynchrone sujet à des défaillances byzantines, où* $k > t$.

Preuve La preuve découle directement du lemme 4, le lemme 5 et le lemme 6. $\square_{Theorme\ 3}$

3.7 Conclusion

Dans ce chapitre, nous avons traité le problème du k-Accord Byzantin avec $k > t$. Nous avons gardé la même propriété de validité que celle du k-Accord dans le cas de déffaillances par crash pour montrer qu'avec la validité la plus générale, plus de choix ne permet pas plus de fautes dans le modèle des défaillances Byzantines, car t est contraint par la condition : $n > 3t$. Nous avons présenté deux protocoles résolvant ce problème : Le premier se base sur la diffusion fiable et le deuxième protocole est un protocole se basant sur l'authentification de messages.

Les deux protocoles ne sont pas tolérants aux intrusions, mais ils garantissent le fait que les processus byzantins ne pourront jamais envoyer des valeurs différentes aux autres processus ou falsifier les messages provenant des autres procssus, et chaque tentative d'un byzantin dont l'objectif est la déviation du protocole sera facilement détectée.

Chapitre 4

Le problème du k-Accord Byzantin Tolèrant aux Intrusions

4.1 Introduction

Dans le k-accord Byzantin asynchrone, un processus a un comportement byzantin quand il se comporte arbitrairement. Ce mauvais comportement peut être intentionnel (comportement malveillant par exemple) ou tout simplement le résultat d'un défaut qui a modifié l'état local d'un processus.

Nous nous intéressons dans ce chapitre à résoudre le problème du k-accord Byzantin dans les systèmes distribués asynchrones. Nous essayons à étudier ce problème quand la validité est forte c'est-à-dire la valeur proposée par un byzantin seulement ne doit pas être décidée.

4.2 Modèle du système

Le modèle du système est constitué d'un ensemble $\prod$ de n processus $(n > 1)$ complètement connectés. On pose $\prod = \{p_1, .., p_n\}$. Jusqu'à t processus peuvent exhiber un comportement byzantin. Un tel processus peut se comporter de manière arbitraire, indépendamment du code et du comportement prévu initialement par le concepteur du système. Il s'agit à ce titre du pire modèle de défaillance : un byzantin peut tomber en panne et s'arrêter, commencer dans un état arbitraire, envoyer des valeurs différentes à différents nœuds, etc. Un processus ayant un comportement byzantin est dit défaillant.

Le réseau de communication est fiable dans le sens où un message envoyé par un processus correct à un autre processus correct est reçu une unique fois en un temps fini. Les messages ne sont pas altérés et le récepteur connait l'identité de l'émetteur : nous utilisons la primitive Validated-Broadcast.

4.2.0.2 La primitive Validated-Broadcast (VB)

4.2.0.3 La spécification de VB

Validated Broadcast (VB) est une nouvelle primitive définie dans [29]. Plus précisement , Validated Broadcast est une diffusion fiable all-to-all (tous vers tous) avec la notion de validation de messages, c'est-à-dire, un message pour qu'il soit valide, il doit être VB-délivré par les processus, par ailleurs, la valeur par défaut $\perp$ est VB-délivrée au lieu du message envoyé.

Comme il est un broadcast all-to-all, VB exige que tous les processus corrects invoquent VB_broadcast(), l'idée est qu'une valeur v est valide s'il existe au moins un processus correct qu'il a broadcast.

Formellement, VB est défini par les propriétés suivantes :

- ***VB-Non-duplicité :*** deux processus corrects ne peuvent pas VB-délivrer des messages distincts de p_i .
- ***VB-Terminaison :*** si l'émetteur est correct, il VB-broadcast un message m, tous les processus corrects inéluctablement VB-délivrent le même message m' où m' = m ou $\perp$.
- ***VB-Uniformité :*** si un processus correct VB-délivre un message de p_i (possible qu'il est défaillant), tous les processus corrects inéluctablement VB-délivrent un message de p_i.
- ***VB-Validité :*** si $\perp$ est VB-délivrée, il existe au moins un processus correct qui n'a pas validé le message VB-broadcasté par p_i.

4.2.0.4 Un algorithme implémentant VB basé sur Reliable-Broadcast

L'algorithme de la figure 4.1 implémente le broadcast validé all-to-all . Rappelons que all-to-all (tous vers tous) signifie ici que tous les processus corrects sont supposés avoir invoqué VB-broadcast(). Ceci signifie que ces processus vont invoquer VB-deliver() pour délivrer au moins $(n-t)$ messages.

Cette implémentation utilise deux invocations consécutives de RB-broadcast. Son coût est de

operation VB_broadcast(v_i)
(1) RB_broadcast INIT(v) ;
(2) **let** rec_i = multiset of values RB_delivered to p_i
(3) **wait until** ($|rec_i| \geq n - t$) ;
(4) **if** ($\#_v(rec_i) \geq n - 2t$)) **then** $aux_i \leftarrow yes$ **else** $aux_i \leftarrow no$ **endif** ;
(5) RB_broadcast VALID(aux_i) ;

for $1 \leq j \leq n$ **VB-delivery** $T_i[j]$:
(6) **wait until** $\big($VALID(x) INIT(v) are RB_delivered from $p_j\big)$;
(7) **if**($x = yes$) **then wait until**($\#_v(rec_i) \geq (n - 2t)$) ; $d = v$
(8) **else wait until**($\#_{v' \neq v}(rec_i) \geq t + 1$)) ; $d = \perp$;
(9) **endif** ;
(10) VB_deliver(d) at p_i as the value VB_broadcast by pj .

FIGURE 4.1 – A Reliable-Broadcast-based VB-Broadcast Algorithm

$2 \times 2 = 4$ étapes de communication et $O(n^3)$ messages de taille $O(log_2 n)$ bits. L'implémentation d'une instance VB-Braodcast est constituée de deux parties.

- La première partie est composée de deux RB_broadcast() consécutives. Plus précisement, un processus p_i invoque tout d'abord RB_broadcast() INIT(v_i) et attend jusqu'à ce qu'il RB_deliver(),des messages d'au moins $n - t$ processus (lignes 1-3). Les valeurs RB_deliverée() sont stockées dans un ensemble $reci_i$.

Ensuite, si la valeur v_i a été RB_deliverée à partir d'au moins $n - 2t \geq t + 1$ processus (Ce qui signifie qu'il a été RB_broadcasté par au moins un processus correct), p_i valide en assignant yes à $auxi$. Sinon p_i met aux_i à no (ligne 4) (dans ce cas, il ne valide pas v_i). Ensuite,p_i invoque un deuxième RB_broadcast() (ligne 5)pour disséminer aux_i à tous les processus.

- La deuxième partie est constituée de n tâches. Les tâches $T_i[j]$ commencent par attendre à la fois la valeur v VB_broadcasté() par p_j et le booléen x VB_broadcasté() par p_j pour dire si la valeur v a été validée ou non.

Notons que la valeur V peut être soit livrée à la ligne 3 ou la ligne 6.

Si $x = yes$, comme p_j peut être Byzantin, v n'a pas nécessairement été validée. Par conséquent, p_i a pour tâche de la vérifier.

À la fin, p_i attend jusqu'à ce que le prédicat ($\#_v(rec_i) \geq (n - 2t)$) devient vrai (ligne 7). Quand ce prédicat devient vrai, nous avons $v(rec_i) \geq t+1$ et par conséquent, v est VB-délivré de p_i comme

étant la valeur VB-broadcastée par p_j.

Autrement, si $x = non$, p_i usqu'à ce que rec_i contient plus de t valeurs différentes de v . Lorsque ceci se produit p_i VB-délivre $\perp$ comme la VB-broadcastée par p_j.

4.2.1 Preuve de correction

Lemme 10 (VB-Non-duplicité) *deux processus corrects ne peuvent pas VB-délivrer des messages distincts de p_i .*

Preuve cette propriété est une conséquence directe de l'abstraction RB_broadcast utilisée pour envoyer un message VALID() à la ligne 5.

$\square_{Lemme\ 10}$

Lemme 11 (VB-Terminaison) *si l'émetteur est correct, il VB-broadcast un message m, tous les processus corrects inéluctablement VB-délivrent le même message m' où m' = m ou $\perp$.*

Preuve Comme il existe au moins $n - t$ processus corrects, et chaque processus correct VB-Broadcast une valeur,nous avons inéluctablement $|reci_j| \geq n - t$ au niveau de n'importe quel processus correct p_j . Par conséquant, RB-broadcast message VALID() à la ligne 5. Considérons maintenant deux cas :
- Soit p_i est un processus correct qui RB-broadcast VALID(yes). Il suit de la ligne 7 que (a) $d = v_i$, et (b) rec_i contient au moins $(n - 2t)$ copies de $v = v_i$, c'est-à-dire, p_i a RB-délivré $(n - 2t)$ messages INIT(v). Dû à RB-uniformité de RB-broadcast, n'importe quel processus p_i inéluctablement RB-delivre ces $(n - 2t)$ messages et VALID(yes), à partir duquel il suit que p_j VB-délivre $v = v_i$ à la ligne 10.
- Soit p_i est un processus correct qui RB-broadcast VALID(no). Il suit à partir de la propriété de RB-terminaison de RB-broadcast que n'importe quel processus correct RB-délivre no à partir de p_i, puis parmi les $(n-t)$ valeurs de rec_i moins que $(n-2t)$ valeurs qui sont similaires à v_i. En plus, dû à RB-uniformité de RB-broadcast, n'importe quel processus p_i inéluctablement RB-délivre $(t + 1)$ valeurs différentes de v_i et, par conséquant, il VB-délivre $\perp$ à la ligne 10.

$\square_{Lemme\ 11}$

Lemme 12 (VB-Uniformité) *si un processus correct VB-délivre un message de p_i (possible qu'il est défaillant), tous les processus corrects inéluctablement VB-délivrent un message de p_i. même message m' où m' = m ou $\perp$.*

Preuve La preuve de cette propriété est très similaire à celle de de la propriété précédente.

$\square_{Lemme\ 12}$

Lemme 13 (VB-Validité) *si $\perp$ est VB-délivrée, il existe au moins un processus correct qui n'a pas validé le message VB-broadcasté par p_i.*

Preuve

Si une valeur $v \neq \perp$ est VB-délivrée par un processus correct $p - i$ comme la valeur VB-broadcastée par p_j, cette valeur apparaît au moins $(n-2t) \geq t+1$ fois dans rec_i ce qui signifie qu'au moins un processus correct a VB-broadcasté v.Similairement, si $\perp$ est VB-délivrée par au moins $t+1$ processus corrects ont VB-broadcasté des valeurs qui se diffèrent de celle VB-broadcastée par p_j.

$\square_{Lemme\ 13}$

Theorème 4 *L'algorithme de la figure 4.1 implémente l'abstraction Validated-Broadcast (VB) dans un système distribué asynchrone avec des défaillances Byzantine où $t < n/3$.*

Preuve A partir du lemme 10 , du lemme 11, du lemme 12 et du lemme 13, nous déduisons directement le théorème. $\square_{Theorme\ 4}$

4.3 Le K-Accord Byzantin Tolérant aux Intrusions et le Modèle Enrichi

4.3.1 Le K-Accord Byzantin Tolérant aux Intrusions

Dans le k-Accord, si tous les processus corrects ne proposent pas la même valeur, n'importe quelle valeur peut être décidée. Comme indiqué dans l'introduction, nous sommes intéréssés par une

version forte du problème du k-Accord dans lequel une valeur proposée seulement par un processus défaillant ne doit être jamais décidée.

Formellement, le k-Accord Byzantin tolérant aux intrusions est défini par les proporiétés suivantes :

- ***Accord :*** Au plus k valeurs différentes sont décidées.
- ***Terminaison :*** Chaque processus correct doit décider une valeur.
- ***Obligation (Validité) :*** Si tous les processus corrects proposent la même valeur v, alors v doit être décidée.
- ***Non-Intrusion(Validité) :*** La veleur décidée est une valeur proposée par un processus correct ou $\perp$.

Du fait qu'aucune valeur proposée seulement par des processus défaillants ne peut être décidée, cette instance du problème du k-Accord obtient son nom (à savoir tolérant aux intrusions).

4.3.2 le Modèle Enrichi pour le k-Accord Byzantin tolérant aux intrusions

4.3.2.1 Une puissance additionnelle est nécessaire

Il est bien connu que le problème du k-Accord ne peut pas être résolu dans un système distribué asynchrone en présence de défaillance si $k \leq t$. Dans notre cas, la puissance ajoutée au système est le consensus binaire qui possède une caractéristique très intéressante.

4.3.2.2 Le Consensus Binaire

Le consensus est binaire quand seulement deux valeurs (exemple : 0 et 1) peuvent être proposées. Quand plus de deux valeurs peuvent être proposées, le consensus est multivalué. Il est très intéressant, le fait que seulement deux valeurs peuvent être proposées pour un protocole de Consensus Binaire Byzantin. Ceci lui fournit une propriété intéressante, à savoir, si tous les processus corrects proposent la même valeur $b \in \{0, 1\}$, il résulte à partir de la propriété d'obligation qu'ils décident a b , quelque soit la valeur (b ou $b = 1 - b$) proposée par les processus défaillants. Par conséquant, nous avons la propriété suivante :

- **propriété 1 :** n'importe quel protocole du consensus binaire qui satisfit la propriété d'obligation, il satisfit également la propriété de non-intrusion. Par conséquant, $\perp$ n'est jamais décidée.

4.4 Un résultat d'impossibilité

Dans cette section, nous allons démontrer que le nombre k de valeurs possibles qui peuvent être décidées, dans un protocole du k-Accord tolérant aux intrusions, est lié au nombre maximal de processus qui peuvent être défaillants t et le nombre total des processus du système n.

D'après les propriétés du k-Accord Byzantin tolérant aux intrusions, il est possible de décider $\perp$, mais il est possible aussi de décider une valeur non-$\perp$ sachant la cardinalité de l'ensemble des valeurs décidées par les processus corrects du système est entre 1 et k. Pour cela, nous allons montrer qu'il existe une borne supérieure sur k qui est liée à n et t, contrairement au cas du système où seulement des crashs sont présents.

Theorème 5 *Dans un protocole du k-Accord Byzantin tolérant aux intrusions, Il est impossible que tous les processus corrects décident sur un ensemble de valeurs dont la cardinalité est k si $k > (n-t)/(t+1)$.*

Preuve

Considérons un ensemble de n processus qui exécutent un algorithme du k-Accord Byzantin. Les processus doivent décider au plus k valeurs. Pour qu'une valeur soit décidée, elle doit être proposée au moins par un processus correct ; ce qui signifie qu'elle doit être proposée par $t+1$ processus (au moins un correct). Supposons que k valeurs sont décidées.

La preuve est par contradiction, c'est-à-dire il est possible de décider sur un nombre de valeurs $k > (n-t)/(t+1)$.

Supposons le meilleur des cas dans lequel chaque valeur non$\perp$ est proposée exactement par (t+1) processus (au moins un correct propose cette valeur conformément à la propriété de non-intrusion) et les t processus Byzantins ne proposent rien (se crashent). Notons que dans le pire des cas, la valeur décidée est $\perp$, car chaque processus correct propose une valeur différente des autres.

Par conséquant, pour décider sur k valeurs différentes dans le meilleur des cas, le système doit être composé de k groupes et chaque groupe est formé par $(t+1)$ processus, comme indiqué dans la figure 4.4. Donc, nous avons un système composé de $(n = k(t+1))$ processus corrects. En ajoutant les t processus Byzantins, alors nous avons un système composé de $n = k(t+1)+t$ processus. Par conséquant, $k = (n-t)/(t+1)$. Une contradiction qui prouve le théorème.

$\square_{Theorme\ 5}$

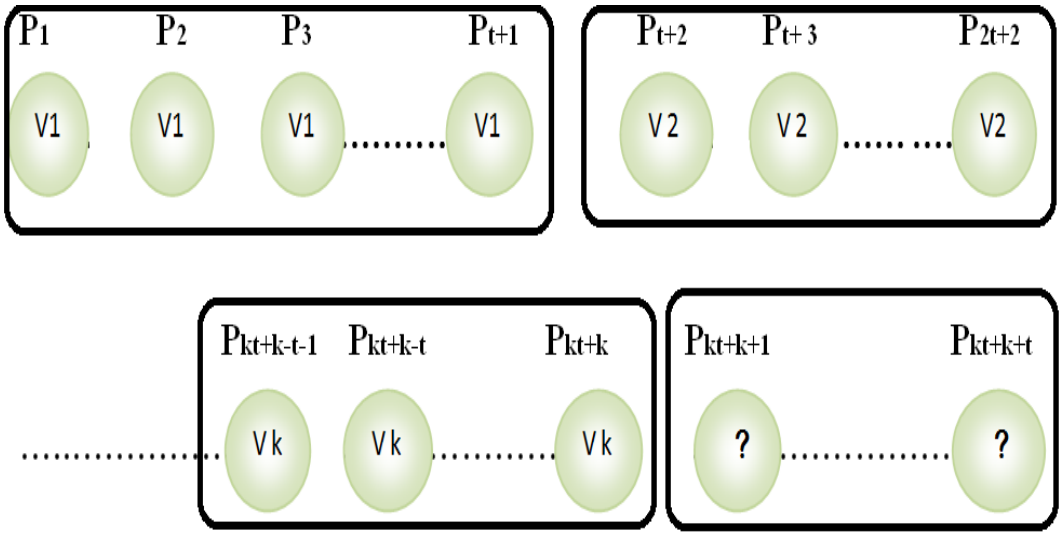

FIGURE 4.2 – Schéma de processus dans une execution d'un protocole du k-Acoord Byzantin

4.5 Un protocole tolérant aux intrusions pour le k-Accord Byzantin

4.5.1 Description du protocole

Cette section présente un protocole (4.3) tolérant aux intrusions pour le k-Accord Byzantin basé sur le broadcast validé. Ce protocole exige que $(n \geq 3t + 1)$. Il exige également au moins une ronde mais, comme il utilise le broadcast validé, cette ronde exige quatre étapes de communication.

Dans la première ronde du protocole (4.3), après avoir VB-broadcast sa valeur, un processus p_i attend la réception des messages EST() de la part de $(n - t)$ processus et dépose les valeurs qui leurs correspondent dans l'ensemble rec_i.

Après, p_i vérifie (en plus de $\perp$) s'il a VB-délivré au moins une valeur non-$\perp$ et que cette valeur a été VB-broadcastée par au moins $(t + 1)$ processus (ligne 6). S'il existe une telle valeur, p_i propose pour l'instance du consensus binaire qui utilise la valeur 1, par ailleurs il propose 0 (ligne 06).

Si cette première instance du consensus binaire retourne 0, alors p_i décide $\perp$ (ligne 9).Différemment , si 1 est retourné , p_i attend jusqu'à ce que au moins VB-délivre (t+1) EST() messages et il stocke les valeurs proposées par au moins un processus correct dans l'ensemble $valid1()$ (lignes 10 et 11).

Par la suite, p_i invoque une deuxième instance du consensus binaire pour assurer la propriété

operation propose(v_i)
Init : $r_i \leftarrow 0$;$est_i \leftarrow v_i$
repeat forever
(1) $r_i \leftarrow r_i + 1$;
——————————————— round r_i ———————————————
(2) VB_broadcast EST(r_i, v_i) ;
(3) **wait until** $\big($EST$(r_i, -)$ VB_delivered from $(n-t)$ processes $\big)$;
(4) **let** rec_i = multiset of the values v such that EST(r_i, v) VB_delivered to p_i
(5) **if** $\big(\exists v \neq \perp : \#_v(rec_i) \geq t+1)\big) \wedge \big($ (rec_i) contains at least one non-$\perp$ value $\big)$
(6) **then** $bp_i \leftarrow 1$ **else** $bp_i \leftarrow 0$
(7) **endif** ;
(8) $b\_dec_i \leftarrow$ propose(r_i, bp_i)
(9) **if** $(b\_dec_i = 0)$ **then** return$(\perp)$
(10) **else wait until** $\big(\exists v \neq \perp$ such that EST(r_i, v) VB_delivered from $(t+1)$ processes $\big)$;
(11) **let** $valid1(rec_i) \equiv \{x | \#_x(rec_i) \geq t+1\}$;
(12) **if** $\big(|valid1(rec_i)| \leq k\big)$ **then** $bp1_i \leftarrow 1$ **else** $bp1_i \leftarrow 0$
(13) $b\_dec1_i \leftarrow$ propose$(r_i, bp1_i)$
(14) **if** $(b\_dec1_i = 1)$ **then** return(any value $v \in valid1(rec_i)$)
(15) **else wait until** $(|valid1(rec_i)| > k)$; $est_i \leftarrow min\{v : v \in valid1(rec_i)\}$;
———————————————————————————————————————
end repeat

FIGURE 4.3 – An Intrusion-Tolerant Byzantine Set Agreement Protocol In a Signature-Free System

d'accord. Si $|valid1(rec_i)| \leq k$, il propose 1, par ailleurs il propose 0 (ligne **??**). Si cette deuxième instance du consensus binaire retourne 1, alors p_i décide n'importe quelle valeur dans $valid1(rec_i)$ (ligne 14). Différemment, il choisit la valeur min de l'ensemble $valid1(rec_i)$ afin de commencer une nouvelle ronde (ligne 15). Le choix de la valeur min a pour objectif d'assurer la propriété d'accord en permettant aux processus de se converger vers au plus k valeurs différentes si le nombre de valeurs valides est supérieur à k .

4.5.2 Preuve de Correction du Protocole

Lemme 14 (Terminaison) *Chaque processus correct doit décider une valeur*

Preuve Si la première instance du consensus binaire b_dec_i retourne 0, la terminaison est triviale. Maintenant, considérons qu'elle retourne 1. Dû à la propriété 1 du consensus binaire, il existe un processus correct p_i tel que $bp_i = 1$, ce qui implique que, à la ligne 3 , p_i a reçu au moins $t+1$ EST() messages pour la même valeur non-$\perp$. Dû aux propriétés de VB-non-duplicité et de VB-non-uniformité du broadcast validé (VB), n'importe quel processus correct, inéluctablement, VB-délivre

ces $t+1$ messages. Par conséquant, aucun processus correct p_i qui se bloque indéfiniment à la ligne 10. Par la suite, Si la deuxième instance du consensus binaire b_dec1_i retourne 1, la terminaison est triviale et un processus correct p_i décide n'importe quelle valeur dans l'ensemble $valid1(rec_i)$ car le nombre de valeurs valides (proposées au moins par un correct) est inferieur à k.

Examinons maintenant le cas où elle retourne 0. Dû à la propriété 1 du consensus binaire ; il existe un processus correct p_i tel que $bp_i = 0$, ce qui implique que, à la ligne 11 , p_i a stocké au moins $k+1$ valeurs valides dans $valid1(rec_i)$. Dû aux propriétés de VB-non-duplicité et de VB-non-uniformité du broadcast validé (VB), n'importe quel processus correct, inéluctablement, VB-délivre au moins $k+1$ messages. Par conséquence, aucun processus correct p_i qui se bloque indéfiniment à la ligne 15, lequel conclut la preuve de la propriété de terminaison.

$\square_{Lemme\ 14}$

Lemme 15 (Accord) *Au plus k valeurs différentes sont décidées.*

Preuve

La preuve est très similaire à celle du lemme précédent. Si la première instance du consensus binaire b_dec_i retourne 0, l'accord est trivial, mais si elle retourne 1, une deuxième instance du consensus binaire b_dec1_i sera invoquée pour vérifier si le nombre de valeurs valides est inferieur ou égal à k. Dans ce cas, n'importe quelle valeur sera décidée.

Différemment, tous les processus corrects choisissent la valeur min de l'ensemble $valid1(rec_i)$ et passent à la prochaine ronde pour se converger vers au plus k valeurs valides pour décider, ce qui complète la preuve de la propriété d'accord. $\square_{Lemme\ 15}$

Lemme 16 (Obligation) *Si tous les processus corrects proposent la même valeur v, alors v doit être décidée.*

Preuve Si tous les processus corrects proposent la même valeur v, il suit à partir de la propriété de VB-validation que v est nécessairement validée, et à partir des autres propriétés qu'ils vont tous VB-delivrer au moins $(n-2t)$ $\text{EST}(v)$ messages. De plus, comme $n-2t > t$, v est unique. Dû à la propriété de VB-validation, une valeur VB-broadcastée seulement par les processus défaillants ne peut pas être validée et par conséquant, aucun processus correct peut la VB-délivrer. Ceci signifie que seulement v, $\perp$ ou rien peut être VB-délivré de la part du processus défaillant. Dû la propriété

d'Obligation du consensus binaire, tous les processus décident 1 dans les deux instances du consensus binaire. Par conséquant, tous les corrects décident la même valeur v dans la première ronde.

$\square_{Lemme\ 16}$

Lemme 17 (Non-intrusion) *La valeur décidée est une valeur proposée par un processus correct ou* $\perp$.

Preuve Si une valeur w est proposée seulement par des processus défaillants, dû la propriété de VB-validation, aucun processus correct peut la VB-délivrer, si la première instance du consensus binaire retourne 0, w est non décidée, si elle retourne 1, nous avons vu dans la preuve de la propriété d'accord que les processus corrects décident une valeur proposée par au moins $t + 1)$ processus, à partir de ça nous concluons que w ne peut pas être décidée. $\square_{Lemme\ 17}$

Theorème 6 *Le protocole de la figure 4.3 résout le k-Accord Byzantin dans un système distribué asynchrone sujet à des défaillances Byzantines, où* $n \geq 3t + 1$*, enrichi par un consensus binaire.*

Preuve A partir du lemme 14 , du lemme 15, du lemme 16 et du lemme 17, nous déduisons directement le théorème. $\square_{Theorme\ 6}$

4.6 Conclusion

Dans ce chapitre, nous avons présenté une famille de protocoles pour résoudre le problème du k-accord. Le protocole du k-accord proposé est construit en se basant sur l'abstraction VB. Par ailleurs, tous les algorithmes présentés dans ce chapitre sont de type signature-free.

CONCLUSION ET PERSPECTIVES

4.7 Conclusion

D'une manière générale, l'objectif principal de ce mémoire résidait dans l'étude et la mise en œuvre des systèmes distribués. Un premier objectif concerne une introduction et une synthèse sur les problèmes d'accord et plus spécifiquement le k-accord. Nous avons présenté pour répondre à cet objectif les résultats les plus significatifs. Un deuxième objectif est une réponse à la première question posée dans l'introduction générale. Cette réponse est négative dans le sens où plus de choix ne permet pas plus de fautes dans le contexte des Byzantins. Un dernier objectif est de proposer un protocole pour le k-accord Byzantin tolérant aux intrusions qui est le premier dans la littérature.

Nous signalons aux lecteurs de ce mémoire que cette étude est la première dans la littérature quant à ce sujet. Pour cela, nous étions perdus à maintes reprises, mais nous avons pu finalement répondre à nos objectifs.

4.8 Perspectives

Les perspectives à ce mémoire sont diverses, nous en citons ci-après deux entre elles.
- Redéfinir la propriété de validité pour la k-accord byzantin, nous pensons que la validité doit avoir un lien avec le nombre de valeurs qui peuvent être décidées k.

- Proposer un protocole pour le k-accord qui se base sur la randomisation.

4.9 Problèmes ouverts

Le K-accord synchrone : Est-il possible de trouver un protocole pour la k-accord Byzantin avec $(t/k) + 1)$ est le nombre minimal de rondes décider dans le cas des systèmes distribués synchrones ?

La topologie mathématique algébrique et le K-accord Est-il possible de modéliser le K-accord Byzantin avec des notions de la topologie algébrique comme le triangle de Bermuda qui a été utilisé pour modéliser le k-accord avec crashs ?

Bibliographie

[1] AbrahamsonK., On achieving consensus using shared memory. *In 7th ACM Symposium on Principles of Distributed Computing*, pages 291-302, 1988.

[2] Anceaume E., Hurfin M., and Raïpin-Parvédy Ph., An efficient solution to the k-set agreement problem. *In Proc. Of the 4th European.Dependable Computing Conference (EDCC-4)*, number 2485 in LNCS, pages 62-78, Toulouse, France, octobre 2002.

[3] Alpern B., and Schneider F., Defining liveness. *Technical report, Department of Computer Science*, Cornell University, 1984.

[4] Ben-Or M., Another advantage of free choice : Completely asynchronous agreement protocols (extended abstract). *In PODC*, pages 27-30, 1983.

[5] Borowsky E., and Gafni E., Generalized FLP Impossibility Results for t-Resilient Asynchronous Computations. *Proc. 25th ACM Symposium on Theory of Computation (STOC'93)*, San Diego (CA), pp. 91-100, 1993.

[6] Baldoni R., Helary J.M., Raynal M., and Tanguy L., Consensus in Byzantine asynchronous systems. *Journal of Discrete Algorithms*, 1(2) :185-210, 2003.

[7] Bracha G., and Toueg S., Asynchronous Consensus and Broadcast Protocols. *Journal of the ACM*, 32(4) :824840, Oct. 1985.

[8] Chaudhuri S., More Choices allow more faults, set consensus problems in totally asynchronous systems. *Inf Comput*, 105, 132-158, 1993.

[9] Cristian F., Aghili H., Strong R., and Dolev D. ; Atomic broadcast, "from simple messagediffusion to byzantine agreement". *In Proceedings of the 15th International Symposium on Fault-Tolerant Computing Systems (FTCS-15)*, 1985.

[10] Chaudhuri, S., Herlihy, M., Lynch, N., Tuttle, M. : Tight Bounds for k-Set Agreement. J. ACM 47(5), 912-943 (2000)

[11] Cristian F., and Fetzer C., The timed asynchronous distributed system model.*IEEE Transactions on Parallel and Distributed Systems*,10(6) :642-657, 1999.

[12] Chang J., and Maxemchuck N., Reliable broadcast protocols. *In ACM Transactions on Computer systems*, pages 251-273, August 1984.

[13] Correia M., Neves N.F., and Verissimo P., From consensus to atomic broadcast : Time free Byzantine-resistant protocols without signatures. *Computer Journal*, 41(1) :82-96, 2006.

[14] Chandra T.D. and Toueg S., Unreliable Failure Detectors for Reliable Distributed Systems. *Journal of the ACM*, 43(2) :225-267, 1996.

[15] Dolev D., Dwork C., and Stockmeyer L., On the minimal synchronism needed for distributed consensus. *In ACM Symposium on Principles of Distributed Computing*, pages 77-97, January 1987.

[16] Doudou A., Garbinato B., and Guerraoui R., Encapsulating failure detection : From crash-stop to Byzantine failures. *In International Conference on Reliable Software Technologies*, pages 24-50, 2002.

[17] Doudou A., Garbinato B., Guerraoui, R., and Schiper A., Muteness failure detectors : Specification and implementation. *In Proceedings of the Third European Dependable Computing Conference*, pages 71-87, 1999.

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

[19] Doudou A., and Schiper A., Muteness detectors for consensus with Byzantines processes. *Technical Report 97/30, EPFL*, 1997.

[20] Fischer M.J., The consensus Problem in Unreliable Distributed System. *Proceedings of the International Conference on Foundations of Computing Theory*, Sweden, 1983.

[21] Fischer M.J., Lynch N., and Paterson M.S., Impossibility of Distributed Consensus with One Faulty Process. *Journal of the ACM*, 32(2) :374-382, 1985.

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

[23] Herlihy M.P., and Shavit N., The Topological Structure of Asynchronous Computability. *Journal of the ACM* , 46(6) :858-923, 1999.

[24] Hadzilacos V., and Toueg S., Distributeds systems , chapter 5 : Fault Tolerant Broadcast and Related Problems , *Addison-Wesley*, pages 97-145, 1993.

[25] Kihlstrom K.P., Moser, L.E., and Melliar-Smith P.M., Byzantine fault detectors for solving consensus. *The Computer Journal*, 46(1) :16-35, 2003.

[26] Lamport L., The part-time parliament.*ACM Transactions Computer Systems*, 16(2) :133-169, 1998.

[27] Malkhi D., and Reiter M., Unreliable Intrusion Detection in Ditributed Computations. *In Proc. 10 th Computer Security Foundations Workshop (CSFW97)*, June 1997.

[28] Mostefaoui A., and Raynal M., Randomized k-Set Agreement.*In ACM Symposium on Principles of Distributed Computing*, 2001.

[29] Mostefaoui A. ; and Raynal M. ; Signature-Free Broadcast-Based Intrusion Tolerance : Never Decide a Byzantine Value.*In ACM Symposium on Principles of Distributed Computing*, 2011.

[30] Powell D., Distributed Fault Tolerance : A Short Tutorial, *in IFIP International Workshop on Dependable Computing and its Applications*, (Johannesbourg, Afrique du Sud), pp 1-12, 1998.

[31] Pease M., Shostak R., and Lamport L., The Byzantin generals problem. *In Proc, ACM Transactions on Programming Languages and Systems* , volume 4, n˚3, pp 382-401, 1989.

[32] Saks M. and Zaharoglou F., Wait-Free k-Set Agreement is Impossible : The Topology of Public Knowledge. *SIAM Journal on Computing*, 29(5) :1449-1483, 2000.

[33] Urban P., and Schiper A., Fault tolerance in Distributed Systems , February 2000.

Résumé

Ce rapport aborde le problème du k-Accord dans les systèmes distribués asynchrones sujet à des défaillances Byzantines. Ce travail progresse l'état de l'art en étudiant ce problème pour la première fois. Le rapport présente plusieurs contributions. Il montre que, dans le problème du k-Accord Byzantin, plus de choix ne permet pas plus de défaillances, contrairement aux systèmes distribués avec défaillances par crash.

Fournir aux processus de l'application des fortes garanties d'accord, en dépit de défaillances est un problème fondamental pour l'informatique distribuée tolérante aux fautes. Les processus corrects ne doivent pas être "pollués" par le comportement erroné des processus défaillants.Pour répondre à cette exigence, le rapport présente un protocole du k-Accord dans lequel aucune mauvaise valeur n'est jamais décidée par les processus corrects. Ces processus décident toujours une valeur qu'ils ont proposée ou une valeur par défaut bot. Ce protocole est appelé protocole du k-Accord tolérant aux intrusions. A notre connaissance, il s'agit du premier protocole du k-Accord tolérant aux intrusions dans la littérature.

Mots clés **:**k-Accord Byzantin,tolérance aux intrusions, systèmes distribués asynchrones, Tolérance aux fautes, protocole.

Abstract

This report tackles the k_Set_Agreement problem in asynchronous distributed systems prone to Byzantine failures. This work advances the state of art by studying this problem for the first time. The report has several contributions. It shows that, in the Byzantine k_Set_Agreement problem, more choices not allow to more faults, unlike of distributed system with crash failures.

Provide application processes with strong agreement guarantees despite failures is a fundamental problem of fault-tolerant distributed computing. Correct processes have not to be "polluted" by the erroneous behavior of faulty processes.To respond to this requirement, the report presents a k_Set_Agreement protocol in which no bad value is ever decided by correct processes. These processes always decide a value they have proposed or a default value $\perp$. This protocol is called Intrusion-free k_Set_Agreement protocol. In our knowledge, this is the first Intrusion-Tolerant Byzantine k_Set_Agreement protocol in the literature.

Keywords **:**Byzantine k_Set_Agreement, ,intrusion-Tolerance, asynchronous distributed system, Faults-Tolerance, Protocol.