République Algérienne Démocratique et Populaire
Ministère de l'Enseignement Supérieur et de la Recherche Scientifique

Université A/MIRA, Béjaia
Faculté des Sciences Exactes
Département d'Informatique

Mémoire de fin de Cycle Master

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

Option

Réseaux et Systèmes distribués

Thème

Protocole de consensus indulgent optimal avec une complexité de communication efficace

Présenté par : $M.$ **Ahmed MAZARI**

Soutenu le 22/06/2015 devant le jury composé de :

Président	$M.$	**Hachem SLIMANI**	**M.C.A**	**Université A/Mira, Béjaia**
Rapporteur	$M.$	**Hamouma MOUMEN**	**M.A.A**	**Université A/Mira, Béjaia**
Examinateur	$M.$	**Abderrahmane SIDER**	**M.C.B**	**Université A/Mira, Béjaia**
Examinatrice	$Mlle$	**Lynda BEN BOUDAOUD**	**Doctorante LMD**	**Université A/Mira, Béjaia**

Promotion 2014/2015

Remerciements

J'exprime mes remerciements à mon encadreur Monsieur **Moumen HAMOUMA** pour ses orientations et ses conseils sans lesquels ce travail n'aurait pas vu le jour,qu'il trouve ici l'expression de ma gratitude.

Je tiens également à remercier Monsieur **Hachem SLIMANI** pour son engagement pédagogique incontestable qui m'a beaucoup aidé à bien assimiler et à mieux appliquer la théorie des graphes dans mes recherches.

J'adresse mes vifs et sincères remerciements plus particulièrement à Monsieur **Amayas Toufik MOSTEFAOUI** pour son précieux suivi permanent pendant mes cinq années à l'université sur tout les plans et pour son partage de connaissances. C'est l'enseignant qui a réussi à m'inspirer, à me faire aimer les sciences et la recherche scientifique. J'ai grandement apprécié son soutien et son attention envers moi.

Je remercie tout particuliérement les membres de jury qui ont accepté de juger mon travail ainsi que tous les enseignants qui ont contribué à ma formation.

Enfin, je remercie aussi ma famille : **ma mère Salima,mon pére Abdelkrim et ma soeur Fadhila**, mes amis, mes camarades et collègues qui m'ont soutenu et tous ceux qui ont contribué de près et de loin à la réalisation de ce travail.

Table des matières

I Définitions, modèles de défaillances et analyse de la complexité 9

1 Concepts et modèles de systèmes distribués 10
1.1 Modèles de systèmes distribués . . . 10
1.1.1 Temps . . . 10
1.1.2 Système distribué synchrone . . . 10
1.1.3 Système distribué asynchrone . . . 11
1.1.4 Système distribué partiellement synchrone . . . 11
1.2 Types de communication et d'exécution . . . 12
1.3 Types de défaillances . . . 13
1.3.1 Pannes franches (crash) . . . 13
1.3.2 Pannes par omission . . . 13
1.3.3 Pannes de temporisation . . . 13
1.3.4 Pannes byzantines . . . 13
1.4 Modèles de défaillances . . . 14
1.5 Algorithme distribué . . . 14
1.6 Exécution d'un algorithme distribué . . . 15
1.7 Problème dans un système distribué . . . 15
1.8 Modes d'accès à la mémoire . . . 15
1.8.1 Mémoire partagée . . . 16
1.8.2 Mémoire distribuée . . . 16
1.8.3 Mémoire distribuée partagée . . . 16
1.9 Conclusion . . . 16

2 Analyse de la complexité algorithmique 17
2.1 Introduction . . . 17
2.2 Notations asymptotiques . . . 17
2.2.1 Complexité en espace d'exécution . . . 18
2.2.2 Complexité en temps d'exécution . . . 18
2.2.3 Complexité en nombre de messages échangés . . . 18
2.2.4 Complexité sur un système synchrone . . . 18
2.2.5 Complexité sur un système asynchrone . . . 19
2.3 Déterminisme et non-déterminisme . . . 19
2.4 Modèles de complexité pour les systèmes distribués . . . 19

II Oracles et les protocoles de consensus indulgents 20

3 Problème de consensus et étude de détecteurs de défaillances 21
3.1 Définition . . . 21
3.2 Types de consensus . . . 21
3.2.1 Consensus binaire . . . 21
3.2.2 K-accord . . . 21
3.3 Impossibilité de FLP . . . 22
3.4 Oracles . . . 23
3.4.1 Métriques d'optimalité . . . 23
3.4.2 Oracle leader . . . 24
3.4.3 Oracle probabiliste . . . 24
3.4.4 Détecteurs de défaillances . . . 24
3.4.5 Popriétés d'un détecteur de défaillances . . . 25
3.4.5.1 Complétude . . . 25
3.4.5.1.a Complétude forte . . . 25
3.4.5.1.b Complétude faible . . . 25
3.4.5.2 Précision . . . 26
3.4.5.2.a Précision forte . . . 26
3.4.5.2.b Précision faible . . . 26
3.4.5.2.c Précision ultime forte . . . 26
3.4.5.2.d Précision ultime faible . . . 26
3.4.6 Classes des détecteurs de défaillances . . . 26
3.4.6.1 Détecteurs de défaillances fiables . . . 27
3.4.6.2 Détecteurs de défaillances non fiables . . . 27
3.4.6.2.a Propriétés des détecteurs de défaillances non fiables . . . 28
3.4.7 Réductibilité . . . 28
3.5 Détecteur de défaillances le plus faible $\diamond S$ pour résoudre le consensus . . . 29
3.6 Conclusion . . . 30

4 Travaux et résultats antérieurs du consensus indlulgent 32
4.1 Introduction et historique de l'indulgence . . . 32
4.2 Définition de l'algorithme indulgent . . . 33
4.3 Modèles de communication pour résoudre le consensus indulgent . . . 34
4.4 Protocole de consensus indulgent de Dutta et Guerraoui . . . 35
4.5 Protocoles de consensus indulgents de Gilbert et al. . . . 35

5 Protocoles pour le consensus indulgent avec communication optimale 36
5.1 Protocole de consensus indulgent en présence de défaillances . . . 36
5.2 Protocole de consensus indulgent en absence de défaillances . . . 38
5.3 Démonstration et preuve de validité des protocoles . . . 39

6 Problèmes ouverts et travaux de recherche en cours **42**
6.1 Algorithme : early-deciding 42
6.2 Algorithme : early-stopping 42
6.3 Problèmes ouverts 43
6.4 Recherches en cours 43

Liste des tableaux

1.1 Types de communication avec leurs paramètres de complexité 12

3.1 Classes des détecteurs de défaillances de $Chandra$ et $Toueg$ [10] 27

5.1 Comparison des protocoles de consensus indulgents 39

Table des figures

3.1 Réductibilité des classes de détecteurs de défaillances 29

5.1 Protocole de consensus indulgent en présence de défaillances 37
5.2 Protocole de consensus indulgent en absence de défaillances 38

Introduction générale

Le problème du consensus est un problème fondamental dans le domaine des systèmes distribués. Résoudre le problème du consensus permet de garantir un service tolérant aux fautes.
Le problème du consensus est introduit par Peas et al.[15]. Ensuite,Fisher et al.[12] ont prouvé formellement qu'il est impossible de résoudre le consensus dans un système distribué purement asynchrone même avec la présence d'une seule défaillance.
Ensuite, Chandra et Toueg [10] ont introduit, la notion de détecteurs de défaillances pour circonvenir les résultats d'impossibilité [12] en augmentant la synchronie des systèmes asynchrones. Les systèmes distribués partiellement synchrones ont été introduits par Dwork et al.[11] où le consensus peut étre résolu.Dans [11], Dwork et al. ont décrit quatres modéles de fautes (la résilience maximale)possible dans le modèle de communication partiellement synchorne. Gafni [2] a étudié le modèle basé sur les tours où il a illustré comment les détecteurs de défaillances peuvent être utilisés pour relier les modèles synchrones et asynchrones.
Guerraoui [3] est le premier à avoir introduit l'algorithme indulgent pouvant tolérer (l'algorithme indulgent est tolérant envers son détecteur de défaillances) une longue période d'asynchronisme artbitraire. Il a caractérisé l'algorithme indulgent et a clarifié la nature des détecteurs de défaillances non fiables.Trois classes de détecteurs de défaillances non fiables ont été introduites : la non fiabilité complète, la non fiabilité forte, la non fiabilité faible. L'article met en relation ces classes avec les classes définies dans [10]. De plus, la sensibilité aux défaillances locales, la sensibilité aux défaillances globales et la propriété de l'équivalence correcte ont été définies pour montrer que les propriéts divergentes n'admettent pas de solutions indulgentes (le consensus est divergent dans n'importe quel environnement de plus de $(n/2)$ processus défaillants).
Guerraoui et Raynal [1] ont défini quatres métriques de performance concernant l'optimalité : efficacité d'oracle, dégradation zéro, décision en une seule étape et l'efficacité de configuration. Ils ont montré qu'aucun leader ni détecteur de défaillances (algorithme de consensus) ne peut être simultanément zéro dégradant et de configuration efficace. Par contre, les algorithmes qui sont oracles efficaces sont intrinsèquement zéro dégradant et certains détecteurs de défaillances peuvent être oracles efficaces et de configuration efficace. Ils ont aussi montré que la structure de l'information du consensus indulgent basée sur une nouvelle abstraction nommée $Lambda$ encapsule l'utilisation de n'importe quel oracle au cours de chaque tour. L'article présente aussi un nouveau résultat d'impossibilité portant sur l'optimalité, montrant que la complexité temporelle (borne inférieure) du problème de consensus ne peut être traitée par le même algorithme.
Duatta et Guerraoui [5] ont prouvé qu'au moins $t + 2$ tours sont nécessaires pour atteindre une décision globale dans le cas d'une éxecution synchrone ou partiellement synchrone tel que t est

le nombre maximum de défaillances. Le protocole suggéré est une amélioration de l'algorithme indulgent le plus efficace qui nécessite $2t + 2$ tours dans une exécution synchrone [20].
Gilbert et.[17] ont suggéré deux algorithmes efficaces dans un modèle partiellement synchrone, le premier a une complexité optimale de $O(n)$ en terme de messages échangés mais s'exécute en un temps super-linéaire $O(n^{1+\epsilon})$. Par contre, le deuxiéme a une compelxité de $O(n.polylog(n))$ en terme de messages échangés avec un temps d'exécution optimal de $O(f)$ tours tel que $f \leq t$. Ces deux algorithmes ont amélioré le protocole le plus efficace d'une compelxité de $\Omega(n^2)$.
Alistrath et al.[21] ont introduit une technique permettant de transformer un algorithme synchrone à un algorithme indulgent reposant sur la primitive de détection d'asynchronisme qui assure la détection locale, la détection globale et la non trivialité basées sur les propriétés de terminaison : la terminaison forte, la terminaison faible et "quiescence" [22].
Pour terminer, Dolev et Lenzen [18] ont prouvé que la décision au bout de $f + 1$ tours est impossible à moins que $f = t$ avec une complexité $\Omega(n^2 f)$ dans le pire des cas. De plus, Ils ont présenté la notion "orderly crash faults" et ont illustré l'efficacité de l'algorithme garantissant la décision au bout de $f+1$ tours avec $\Omega(nt)$ messages échangés (borne minimale) [19]. De plus, ils ont démontré que la borne minimale pour n'importe quel protocole décidant au bout de $f + 2$ est de complexité $\Omega(t^2 f)$ en terme de messages échangés dans le pire des cas qui tolère jusqu'à t fautes byzantines. Cependant, ils ont suggéré un algorithme décidant au bout de $f+1$ tours (se termine au tour $f+2$) d'une complexité de $O(nt)$ avec crash ordonné.
Les protocoles suggérés jusque la ne sont pas optimaux en communication. Pour ce faire,nous allons adopter le plan suivant pour présenter les résultats les plus récents et décrire notre contribution. En premier lieu, nous allons définir les concepts nécessaires à la compréhension de notre contribution. Il s'agit des types de systèmes distribués, modèles de communication et le modèle de défaillances. En deuxième lieu, nous allons aborder les notations asymptotiques et les modèles de calcul de complexité pour les systèmes distribués afin d'analyser l'efficacité et l'optimalité de nos résultats en les comparant aux travaux récents les plus efficaces. En troisième lieu, nous évoquerons le problème de consensus, les oracles distribués et leurs propriétés afin de mieux comprendre leur fonctionnement et évaluer l'éfficacité de ces outils pour résoudre le consensus. En quatrième lieu, nous énumérons certains travaux antérieurs liés à la résolution du consensus à base de protocoles indulgents les plus efficaces connus jusqu'à présent. Puis, nous allons aborder notre contribution. On commence par décrire le principe de notre approche et présenter les deux protocoles, calculer la complexité de nos protocoles et enfin on démontre formellement la validité de nos résultats par un ensemble de lemmes, preuves et théorème.
Nous concluons notre travail en présentant un certains nombre de problèmes ouverts majeurs et nos travaux de recherche en cours.

Première partie

Définitions, modèles de défaillances et analyse de la complexité

Chapitre 1

Concepts et modèles de systèmes distribués

1.1 Modèles de systèmes distribués

Nous notons un système distribué Π composé d'un ensemble de n processus. Chaque processus est doté d'un identifiant unique tel que les identifiants appartiennent à l'intervalle d'entiers suivant $P = [1; n]$ .Nous noterons p_i le processus d'identité $i \in P$ tel que Π=$p_1, p_2, .., p_n$ et $|\Pi| = n > 1$.
Ces processus distants communiquant via un réseau exécutent une suite d'instructions élémentaires : calculs locaux, appels à des primitives de communication.
Dans notre étude, nous considérons le modèle de calcul partiellement synchrone doté d'un détecteur de défaillances non fiable [10,23]. Ce modèle a été proposé après les résultats d'impossibilié [2]. Le modèle est caractérisé par l'absence de borne portant sur la vitesse relative des processus et les délais de communications.

1.1.1 Temps

Il est nécessaire de rappeler l'importance de la notion du temps dans le calcul distribué afin de pouvoir raisonner sur la causalité entre les événements. Il nous permet de modéliser les échanges de messages sur les canaux de communiction et le calcul de la vitesse relative de chaque processus.
$l'horloge\ globale$: Dans notre analyse des protocoles, on suppose qu'il existe une horloge globale discrète qui capture l'état des processus à chaque fois qu'un processus procède à une nouvelle étape. Cette horloge demeure inaccessible par les processus.
On pose var_i une variable locale d'un processus p_i tel que var_i^t la valeur de var_i à l'instant t et i, $t, T \in N$ [24].

1.1.2 Système distribué synchrone

Un système distribué synchrone est caractérisé par deux paramètres connus au préalable : la vitesse relative de chaque processus et les délais de transmission. On note γ une valeur qui borne supérieurement la vitesse relative de chaque processus (ϕ), et Δ le délais de communication.
La vitesse relative d'un processus veut dire si un processus quelconque prend γ étapes,alors le reste

des processus prennent au moins une étape.
Le délai de délivrance des messages veut dire si un processus envoie un message m après l'étape k dans une éxecution, le message m doit être délivré après au plus $k + \Delta$ exécutions de son envoi. La borne σ entre deux processus p et q peut être calculée en utilisant γ, Δ et le paramètre s (le nombre maximum d'étapes necessaires pour que le processus q reçoive un message et émette une reponse) comme suit [25] :

$$\sigma = \Delta + s.\gamma + \gamma.\Delta.$$

La valeur σ demeure difficile à déterminer, la fixer revient à mettre un compromis entre l'efficacité et la correction.Paradoxalement, poser la valeur σ très grande engendre le blocage du protocole pour une longue période avant de reprendre le calcul dans le cas de présence de défaillances, et la poser petite génère des erreurs : considérer un processus distant défaillant alors qu'il est uniquement lent. Le défi consiste à poser une valeur σ grande et suffisante mais petite possible.

1.1.3 Système distribué asynchrone

Le système distribué asynchrone est dépourvu d'hypothèses de sychronie portant sur la vitesse relative des processus. De plus, les délais de communication existent mais demeurent inconnus d'où l'impossibilité de distinguer un processus lent d'un processus défaillant[12]. Les processus dans ce modèle de communication réagissent à l'événement réception plutôt que d'attendre l'événement (communications non bloquantes).
Un système distribué asynchrone est caractérisé par :

- Un ensemble de processus connectés par des canaux de communication fiables.
- Un système de communication dit "passage de message point à point" en utilisant les deux primitives $send$ et $receive$.
- Topologie complétement connectée.
- L'absence de l'ordre dans la délivrance des messages.
- L'absence de borne sur la vitesse relative des processus et le délais de délivrance des messages.

1.1.4 Système distribué partiellement synchrone

Un systéme distribué partiellement synchrone[11] est caractérisé par sa synchronicité partielle. Ce modèle est défini par trois paramètres qui sont connus à priori : n le nombre de processus ; γ la borne supérieure de la vitesse relative des processus ; Δ représente les délais de communication. Deplus, il existe un temps de stabilisation général nommé GST qui est connu qu'après que le systéme devienne synchrone. On dit qu'une éxecution est synchrone si $GST = 0$. Avant le GST, les messages peuvent être arbitrairement retardés, après le GST, chaque message est délivré au bout d'un temps Δ. Chaque processus a une horloge locale, après le GST=0 la vitesse relative de chaque processus est bornée supérieurement par γ.Les processus fonctionnent à la même vitesse mais ne sont pas actifs à chaque tour. C'est-à-dire, le système oscille entre le modèle synchorne et asynchrone. Il est caractérisé par des périodes d'instabilité.De plus, Il a été prouvé par Cristian et Fetzer [34] que la longueur moyenne de la période de stabilité (synchronisme) est plus longue que la période d'instabilité(asynchronisme) dans les résaeux locaux.

Dans notre étude, nous considérons ce type de systéme dont $t < (n/2)$ processus au maximum peuvent être défaillants. Les processus communiquent par passage de messages.Son fonctionnement repose sur le principe de tour composé de quatres phases [5,26]. Le temps est divisé en tours synchrones mais le système est asynchrone. Il ne garantit pas que le message envoyé dans un tour est délivré dans le même tour. Cependant, on suppose que les processus reçoivent au moins $n - t$ messages dans chaque tour. On note ce système par $ESC(n, t)$.

1.2 Types de communication et d'exécution

Le rôle de la synchronisation des processus dans le calcul distribué a un impact considérable lorsqu'il s'agit de construire un système tolérant aux fautes qui fournit des services fiables. Pour ce faire, nous allons procéder à élucider les différents types de communications qui sont :

- Communication et processus synchrones.
- Communication et processus asynchrones.
- Communication partiellement synchrone et processus syschrones.
- Communication et processus partiellement synchrones.
- Communication synchrone et processus partiellement synchrones.

Notre étude se base sur les résultats obtenus dans [11] pour décrire la complexité temporelle de chaque type de communication. La complexité temporelle pour tout types de fautes dans ces types de communication est d'ordre polynomial dépendant des trois paramètres n (le nombre de processus), ϕ (la vitesse relative des processus) et Δ (le délai de communication). Le tableau suivant récapitule les types de communications avec leur paramètres de complexité :

paramètre / modèle	n	ϕ	Δ
Communication et processus synchrones	✔		✔
Communication et processus asynchrones	✔		✔
Communication partiellement synchrone et processus syschrones			
Communication et processus partiellement synchrones	✔	✔	✔
Communication synchrone et processus partiellement synchrones	✔	✔	✔

TABLE 1.1: Types de communication avec leurs paramètres de complexité

Dans le type de communication partiellement synchrone, les délais de communications Δ existent mais demeurent inconnus à priori ou bien connus qu'à partir d'un temps T inconnu. Pour cela, la délivrance du message se fait soit tardivement ou ne se fait pas du tout.
Les processus partiellement synchrones sont caractérisés par la relativité du paramètre ϕ qui varie d'un processus à un autre(dans le cas sychrone $\phi = 1$).
Lorsque le système de communication et le processus sont partiellement synchrones on suppose que la communication et les processus disposent du méme type de synchronie comme suit :

– soit Δ et ϕ existent mais inconnus
– ou bien Δ et ϕ connus mais à partir d'un temps T inconnu

1.3 Types de défaillances

Les algorithmes répartis ont pour objectif de fournir des services tolérants aux fautes. Pour cela, nous difénissons les différents types de fautes logicielles.
Pour une éxecution donnée, nous noterons F l'ensemble des identités de processus fautifs, et $C = P/F$ l'ensemble des identités corrects.
On note t le nombre maximum de processus qui peuvent être défaillants lors d'une éxecution tel que $t < n/2$ et $1 \leq |C| \leq n$.[15]
Notre étude prend en considération ce type de défaillances.

1.3.1 Pannes franches (crash)

Ce type de panne est appelé aussi arrêt sur défaillance. On distingue deux cas pour un processus : il est soit opérationnel et répond à ses spécifications, soit il ne fait rien (processus défaillant). Dans ce dernier cas, un processus s'arrête prématurément et ne fait rien à partir de ce point et il ne peut reprendre son exécution.[15]

1.3.2 Pannes par omission

Dans ce type de défaillance, un processus défaillant peut omettre aussi bien certaines actions. Ces actions concernent l'envoi et la réception des messages. La défaillance se situe au niveau du réseau qu'au niveau des processus[15].

1.3.3 Pannes de temporisation

Ce type de panne concerne les comportements anormaux par rapport à un temps,un processus ne respecte pas les délais d'exécution des tâches. Le cas de l'expiration d'un délai de garde. Ce type de pannes est appelé aussi panne de performance[15].

1.3.4 Pannes byzantines

Un processus est dit byzantin s'il ne se comporte pas selon le protocole. Il dévie de sa spécification de maniére arbitraire, c'est-à-dire, le processus se comporte arbitrairement. A titre d'exemple,un processus byzantin peut corrompre des messages ou encore ne pas exécuter volontairement des parties entières de l'algorithme [15].

1.4 Modèles de défaillances

Un processus peut être défaillant à cause d'un $crash$. Le modèle de défaillances F est une fonction de τ vers 2^{Π}, ou $F(\tau)$ est l'ensemble des processus défaillants jusqu'à l'instant τ tel que τ représente l'horloge globale.
$Crashed$(F)=$\cup_{t\in\tau}$F(t) représente l'ensemble des processus défaillants dans F, et $correct$(F)=$C(F)$ represente l'ensemble des processus corrects dans F. Le modéle de défaillances de pannes franches suppose que s'il existe au moins un processus correct, $correct$(F)$\neq\oslash$. Si un processus subit une défaillance alors il le reste, c'est à dire $\forall\, t\in\tau : F(t)\subseteq F(t+1)$.
Dans ce modèle t ($1 \leq t < n$) indique la borne supèrieure sur le nombre de processus pouvant être défaillants dans une éxecution.Lorsque $t = n - 1$, l'ensemble de détecteurs de défaillances fournissent des informations éventuellement fausses sur le modèle de défaillance F. Formellement, un détecteur de défaillances D est une fonction qui associe à chaque modèle de défaillances F un ensemble d'historiques de détecteurs de défaillances $D(F)$.$D(F)$ est l'ensemble de tous les historiques pouvant exister durant les exécutions avec l'ensemble des défaillances F et le détecteur de défaillances D.
Un **environnement** E est un ensemble de modèles de défaillances. Il décrit les défaillances qui peuvent avoir lieu dans un système.On considère un environnement qui contient un modéle F_0 dépourvu de défaillances et au moins un modéle de défaillances dont certains processus peuvent être défaillants.
La **couverture d'un modèle de défaillances** est définie formellement comme suit :
on pose F_1 et F_2 deux modèles de défaillances.On dit que F_2 $couvre$ F_1 si $\forall t \in \phi$, $F_2(t)\subseteq F_1(t)$.Le modèle de défaillance F_0 couvre tous les modèles de défaillances.

1.5 Algorithme distribué

Un algorithme est une collection A de n automates déterministes A_i (un par processus p_i). Le calcul procède en étape d'algorithme.Dans chaque étape de l'algorithme A, un processus p_i effectue d'une façon atomique ces trois actions :

1. p_i reçoit un message de certains processus p_j, ou bien un message null λ.
2. p_i envoie une requête à son propre détecteur de défaillances et reçoit une valeur d.
3. p_i change son état et envoie un message à certains processus(éventuellement nul).Cette action est effectuée suivant :
   (a) l'automate A_i.
   (b) l'état du p_i au début de l'étape.
   (c) le message reçu à $l'action1$.
   (d) la valeur d vu par le p_i à $l'action2$.
   Le message reçu par un processus est choisi d'une manière non dèterministe parmi les messages dans le buffer destiné à p_i, et le message λ.
   Une configuration est une paire de (I,M) où I est une fonction reliant un processus p_i à son état local, et M est l'ensemble des messages courants dans le buffer. Une configuration (I,M)

est une configuration initiale si $M = \oslash$.
Une étape d'un algorithme A est un tuple $e = (p_i, m, d, A)$ défini par l'algorithme A dont m et le message reçu par p_i durant cette étape, et d la valeur vue par p_i durant cette étape.
Une étape $e = (p_i, m, d, A)$ est applicable à une configuration (I,M) si seulement si $m \in (M \cup \lambda)$. L'unique configuration qui résulte de l'application de e à la configuration $C = (I, M)$ est notée $e(C)$ [3].

1.6 Exécution d'un algorithme distribué

L'éxecution d'un algorithme A utilisant un détecteur de défaillance D est un tuple nommé $R =< F, H_D, I, S, T >$ tel que F est le modèle de défaillance, $H_D \in D(F)$ est l'historique du détecteur de défaillances D pour le modèle de défaillances F, I est la configuration initiale de A, S est une séquence infinie d'étapes de A et T est la liste des valeurs du temps incrémentées indiquant à quelle étape S s'est produite.
Une éxecution doit satisfaire les propriétés suivantes :

(a) Un processus ne peut exécuter une étape une fois crashé.

(b) Quand un processus exécute une étape et envoie une requête à son module, il obtient la valeur courante retournée par son détecteur de défaillance local.

(c) Chaque processus correct dans F exécute une infinité d'étapes dans S reçoit chaque message qui lui a été destiné.

1.7 Problème dans un système distribué

Un problème P est défini par un ensemble de propriétés que les exéctutions (R) doivent satisfaire. Un algorithme A résout un problème P utilisant un détecteur de défaillances D si toutes les éxecutions de A utilisant D satisfont les propriétés requises par P.On pose $\wp$ une classe de détecteur de défaillances.
Un algorithme A résout le problème P en utilisant $\wp$ si pour tout $D \in \wp$, A résout P.
On dit qu'un problème P peut être résolu en utilisant $\wp$ si pour tout $D \in \wp$, il existe un algorithme A qui résout P en utilisant D.
On pose ν une variable dans l'algorithme A. On note ν_p la valeur de p. L'historique de ν dans une exécution R est notée ν^R qui est $\nu^R(P, t)$ la valeur de ν_p à l'instant t dans une exécution R. On note D_p le module de détecteur de défaillances local associé au processus P, d'ou la valeur de D_p à l'instant t dans l'éxecution $R =< F, H_D, I, S, T >$ est $H_D(p, t)$.

1.8 Modes d'accès à la mémoire

On distingue trois type d'accès à la mémoire.

1.8.1 Mémoire partagée

Ce type de mémoire est utilisée par un multi-processeur non distribué.Plusieurs processeurs se partagent une même mémoire. Cependant les machines doivent être dotées de mécanismes matériels pour éviter les conflits.
On distingue trois modéles de ce type :

(a) UMA accès à la mémoire uniforme

(b) NUMA accès à la mémoire non uniforme

(c) COMA accès à la mémoire cache uniquement

1.8.2 Mémoire distribuée

Dans ce type de mémoire, les processus peuvent être parallèles ou distribués. La mémoire est utilisée par un réseau de multi-ordinateurs communiquant par le transfert de message point à point. Contrairement à la mémoire partagée, il n'y a pas de conflits à gérer.
Notre systéme repose sur ce type de modéle : mémoire distribuée dans un système distribué partiellement synchrone.

1.8.3 Mémoire distribuée partagée

C'est une mémoire distibuée mais partagée.Chaque processus dispose de sa propre mémoire et d'une mémoire distribuée accessible par tous les processus du système.
Un middleware émule une mémoire partagée. Il permet l'interopérabilité entre applications homologues (de même nature), on peut distinguer deux grandes familles :
– Les middlewares qui permettent l'invocation synchrone de fonctions et méthodes, parmi lesquels on trouve la famille des request brokers, avec CORBA ou encore DCOM.
– Les middlewares d'échange asynchrones, qui sont principalement à base de messages, ce sont les MOMs, les Message Oriented Middleware.

1.9 Conclusion

Dans ce chapitre nous avons abordé les modèles de systèmes distribués, les types de systèmes distribués, les types de communication et les types de défaillances. De plus, nous avons évoqué le modèle de défaillances associé à notre contribution et les différents modes d'accès à la mémoire.
Dans le chapitre qui suit nous allons introduire les notations asymptotiques qui nous permettent de d'analyser l'efficacité des protocoles de consensus indulgents.

Chapitre 2

Analyse de la complexité algorithmique

2.1 Introduction

Dans ce chapitre, nous introduisons les différents concepts et métriques de performances permettant de prouver l'efficacité et l'optimalité de nos protocoles (protocoles de consensus indulgent) qu'on va présenter dans la chapitre six.
Tous les problèmes décidables (théorie de la calculabilité) disposent des algorithmes pour les résoudre.Cependant, il y a des problèmes décidables qui ne sont pas traitables en pratique à cause de leur temps de réponse et de l'espace mémoire utilisé. Pour cela, la théorie de la complexité à été introduite pour étudier la traitabilité des problèmes décidables.
La complexité d'un algorithme consiste à calculer le nombre d'opérations élementaires nécessaires à son éxecution dans le pire des cas.Elle permet de mesurer les ressources critiques (temps et espace mémoire) utilisées par l'algorithme.Cela permet de connaitre le type de croissance en fonction de la taille du problème.

2.2 Notations asymptotiques

La mesure de la complexité des algorithmes utilise trois types de fonctions :
$exponentielle$, $polynomiale$ et $logarithmique$. Les notations asymptotiques permettent une analyse de la borne supérieure et inférieure telles que toutes les fonctions manipulées sont du type : $g(n)\ N \rightarrow N$.
Il est difficile de déterminer certains paramètres tels que le temps de communication d'une façon exacte. Pour cela des méthodes approchées ont été introduites, il s'agit de la notation de $Bachman - Landau$ [35,36,37].

(a) Grand O : Elle pose une borne supérieure telle que $g(n) = O(f(n)) : \exists\ c > 0, \exists\ n_0, \forall\ n > n_0\ |g(n)| \leq c|f(n)|$. On dit que la fonction $|g|$ est bornée par la fonction $|f|$ asymptotiquement à un facteur près.
Cette notation sera utilisée pour décrire la borne supérieure sur le nombre de tours et le

nombre de messages échangés dans le pire des cas concernant nos deux protocoles qu'on décrira dans le chapitre six.

(b) Théta Θ : Cette notation permet de capturer la relation entre plusieurs facteurs. On écrit $g(n) = \Theta(f(n))$ si $g(n) = O(f(n))$ et $g(n) = \Omega(f(n))$ si seulement si $\exists$ deux constantes $c_1 > 0$, $c_2 > 0 \in R$ et un entier $N \geq 0$ tel que : $c_1 f(n) \leq g(n) \leq c_2 f(n)\ \forall n \geq N$.

(c) Omega Ω : Elle pose une borne inférieure, on écrit $g(n) = \Omega(f(n))$, s'il existe deux constantes strictement positives c et n_0 telles que $cf(n)$ est une borne inférieure de $g(n)$ tel que $g(n) \geq cf(n)$, $\forall\ n \geq n_0$.

(d) Petit o : $f(n) = o(g(n))$ si et seulement si pour toute constante positive réelle ϵ, il existe un entier non négatif N tel que : $f(n) \leq \epsilon g(n)\ \forall\ n \geq N$, c'est à dire $f(n)$ est négligeable devant $g(n)$ asymptotiquement.

La performance des algorithmes séquentiels est mesurée à base de ces notations asymptotiques. Pour les algorithmes distribués, la définition de la complexité en temps et espace doit être redéfinie.On rajoute à ces deux paramètres, la complexité en terme de messages échangés pour les systèmes communiquant par le passage de messages.

2.2.1 Complexité en espace d'exécution

La complexité en espace d'exécution d'un algorithme A est le nombre de bits total utilisés par A sur un réseau dans le pire des cas [4].

2.2.2 Complexité en temps d'exécution

La complexité en temps d'exécution d'un algorithme est le nombre d'opérations élémentaires,sachant qu'une opération est dite élementaire si son temps d'éxecution est borné supérieurement par une constante [4].

2.2.3 Complexité en nombre de messages échangés

La complexité en espace mémoire est proportionnelle avec le nombre de messages, et la complexité en temps dépend de la taille du message (elle augmente le temps de communication) [4].

2.2.4 Complexité sur un système synchrone

La complexité en temps d'un algorithme A sur un graphe est le nombre de tops d'horloge généré par A dans le pire des cas entre le début d'exécution du premier processeur jusqu'à l'arrêt du dernier processeur [4].

2.2.5 Complexité sur un système asynchrone

La complexité en temps d'un algorithme A sur un graphe est le nombre d'unités de temps du début jusqu'à la fin dans le pire des cas [4].

2.3 Déterminisme et non-déterminisme

Un algorithme déterministe correspond à une machine de turing déterministe. Autrement dit, à chaque configuration de la machine de turing correspond au plus à une seule transition, c'est-à-dire une exécution sur une même machine avec les mêmes données donnera toujours les mêmes résultats.
Contrairement à l'algorithme non-déterministe, la configuration de la machine de turing peut correspondre à plusieurs transitions (il peut exister une infinité finie de transitions), c'est-à-dire les résultats dépendent de la transition, d'où les résultats ne sont pas les mêmes [38].

2.4 Modèles de complexité pour les systèmes distribués

Afin d'étudier la complexité des systèmes distribués, trois modèles ont été dédiés :
LOCAL, CONGEST, ASYNC. Dans notre cas, on s'interesse aux deux modèles $CONGEST$ et $ASYNC$ [39].

- **Modèle LOCAL** : on calcule le temps requis pour un problème sur un système forcément synchrone et sans congestion de messages. La taille des messages est supposée illimitée.
- **Modèle CONGEST** : on étudie le volume de communication.Le système peut être synchrone ou asynchrone.
- **Modèle ASYNC** : on étudie l'effet de l'asynchronisme.

Deuxième partie

Oracles et les protocoles de consensus indulgents

Chapitre 3

Problème de consensus et étude de détecteurs de défaillances

3.1 Définition

Le problème de consensus est un problème de décision, il peut être défini comme suit : plusieurs processus proposent différentes valeurs et décident à la fin sur une valeur proposée.Résoudre le consensus exige l'implémentation de deux primitives $Propose()$ et $Decide()$ On appelle protocole de consensus tout protocole vérifiant les propriétés suivantes :
– (1) **terminaison** : tous les processus corrects décident finalement.
– (2) **validité** : si un processus décide v alors v a été proposé par au moins un processus.
– (3) **intégrité** : aucun processus ne décide deux fois.
– (4) **accord** : si un processus correct décide une valeur v,de même si un autre processus correct décide une valeur $v^{'}$ alors $v = v^{'}$.

3.2 Types de consensus

3.2.1 Consensus binaire

Dans ce type de consensus, seules les valeurs 0 et 1 peuvent être proposées et la valeur décidée ne peut être autre que 0 ou 1.

3.2.2 K-accord

Le $k - accord$ [27] est une généralisation du problème de consensus ($k = 1$).Il est défini comme suit : chaque processus propose une valeur et doit décider sur une valeur en respectant les propriétés suivantes :
– Terminaison : chaque processus décide sur une valeur.
– Validité : La valeur décidée doit être proposée.

– Accord : au plus k valeurs différentes sont décidées.
– intégrité : aucun processus ne décidé deux fois.

3.3 Impossibilité de FLP

Il a été démontré [12] qu'il est impossible de résoudre le problème de consensus dans un système distribué purement asynchrone même en présence d'une seule défaillance.Pour cela, la réception de l'ensemble des messages attendus devient incertaine étant donné qu'un processus ne detienne aucun moyen qui lui permet de distinguer un processus défaillant d'un processus lent.Dans ce cas, la communication devient bloquante du fait qu'un processus correct attende la réception d'un ensemble de messages qu'il ne va pas recevoir finalement.
La preuve du consensus est présentée en se basant sur le consensus binaire dont n processus possédant une valeur d'entrée et sortie $\in (0, 1, \perp)$. Un processus ne peut décider qu'une seule fois sa valeur de sortie (INTÉGRITÉ) et doit vérifier les propriétés de VALIDITÉ et d'ACCORD.Les processus sont modélisés par des automates(avec un nombre potentiellement infini d'états) communiquant par des messages via un buffer. Le systéme de messages agit de façon non déterministe.
Une configuration C est l'état interne de chaque processus ainsi que l'état du buffer(messages envoyés mais pas encore reçus). Un évènement $e = (p, m)$ est la réception de m par p puis l'envoi d'un nombre fini de messages et p passe à un nouveau état interne.On appelle σ une séquence finie ou infinie d'évenements pouvant être appliquée à la configuration C. Une valeur de décision v d'une configuration est la sortie d'un processus ayant décidé.Une configuration C est dite $bi - valente$ s'il existe des configurations atteignables prenant la décision 0 et des configurations atteignables prenant la décision 1.Si toutes les configurations atteignables décisionnelles prennent la décision 0(1 respectivement) alors la configuration C est dite $0 - valente$ (respectivement $1 - valente$).
Un protocole de consensus est $partiellement\ correct$ si aucune des configurations accessibles n'a plus d'une valeur de décision (proprité d'accord) et s'il existe au moins une configuration accessible avec $v = 0$ et au moins $v = 1$.Un processus n'est pas défaillant s'il exécute un nombre infini d'évènements (reçoit(p) est exècuté un nombre infini de fois). Une exècution est $admissible$ si au plus un des processus est défaillant et si tous les messages sont reçus.Une exécution est decionnelle si au moins un processus prend une décision.D'ou un protocole est $correct$ s'il est partiellement correct et si toutes les exècutions admissibles sont décisionnelles.
Le théorme de $Fisher$ et al. [12] démontre que tout protocole de consensus partiellement correct possède au moins une exècution admissible qui n'est pas décisionnelle.
La démonstration démarre du principe que chaque protocole correct ait une configuration initiale bivalente.La décision ne dependrait que de $n - 1$ processus. $Fisher$ et al. ont montré que $\forall$ l'évènement $e = (p, m)$ applicable a une configuration C(bivalente), $\exists$ une configuration bivalente atteignable dont e est le dernier évènement appliqué. On peut donc construire une exècution admissible non décisionnelle de la façon suivante : on crée une liste de processus, initialement rangés dans un ordre arbitraire, et le buffer de messages est organisé en

liste $FIFO$. On crée une exècution par partie. Chaque partie se termine lorsque le premier processus dans la liste fait un pas(s'il existe, le plus ancien message est reçu). Puis ce processus est passé en fin de liste. Cette construction garantie une exècution admissible car tous les messages sont reçus et aucun processus n'est défaillant(ils exécutent tous un nombre infini de pas). Si on trouve dans une exécution bi-valente C, que le processus p est le premier de la liste et que le message m est le plus ancien dans le Buffer pour p alors il existe une configuration bivalente C' atteignable dont e=(p,m) est le dernier évènement. Par récurrence (configuration initiale est bivalente), on peut construire une exècution qui ne prend jamais de décision. Cela contredit le caractère correct du protocole et démontre donc le théorme de Fisher et al. [12].

3.4 Oracles

Dans l'objectif d'augmenter la synchronie des systèmes asychrones et de remédier au manque d'informations induit par l'asynchronisme, la notion d'oracle avait été introduite. Un oracle donnant des informations plus au moins fiables sur l'état des processus (en panne ou non) permet d'avoir un système plus robuste que s'il était purement asynchrone.
Dans cette partie, nous évoquerons les différents oracles : oracle probabiliste,les détecteurs de défaillances et l'oracle leader.
Cependant, lorsque tous les processus qui ne sont pas défaillants initialement proposent la même valeur initiale, aucun oracle n'est nécessaire pour atteindre une décision. Dans ce cas, deux étapes de communication sont nécessaires et suffisantes pour parvenir à un accord.

3.4.1 Métriques d'optimalité

Dans cette sous-section, nous aborderons les quatres métriques d'optimalité en terme de la complexité temporelle : oracle-efficace, zéro-dégradation, décision en une seule étape,
configuration efficace. On considère le nombre d'étapes de communication nécessaires pour qu'un processus décide dans une exécution donnée[1].

- **Oracle-efficace :** lorsqu'un oracle se comporte parfaitement, deux étapes de communication sont nécessaires et suffisantes pour atteindre le consensus dans une exècution dépourvue de défaillances(oracle parfait) [28]. Les algorithmes atteignant le consensus en deux étapes (la borne minimale) [29,30,31] sont appelés *oracles efficaces* car ils sont optimisés pour le bon fonctionnement de l'oracle.
- **Zero-dégradation :** Cette propriété est une extension de la propriété précedente, c'est-à-dire basculant d'une exécution dépourvue de défaillances à une exécution avec défaillances initiales. L'algorithme de consensus décide au bout de deux étapes de communication lorsque l'oracle est parfait P avec la présence de processus défaillants à l'état initial[8]. Cette propriété est importante étant donné que la défaillance d'un processus donné dans un tour r_i apparait comme processus défaillant à l'instant initial du tour suivant r_{i+1}. Cela n'a pas d'impact sur les performances des tours ultérieurs

– **Décision en une seule étape :** Les algorithmes fonctionnant à base d'apprentissage sont dotés de cette proprité : décision après une seule étape d'exècution et sont appelés $one - step - decision\ algorithms$. Le processus exploite une connaissance initiale sur une valeur privilégiée ou sur un sous-ensemble de processus. Lorsque tous les processus corrects proposent la valeur privilégiée. Cette valeur a une probabilité importante d'être proposée plus que les autres valeurs.
– **Configuration efficace :** Lorsque tous les processus proposent la même valeur initiale,aucun oracle n'est désormais nécessaire pour atteindre une décision. Dans ce cas, deux étapes de communication sont nécessaires et suffisantes pour atteindre le consensus. Les algorithmes [32] vérifiant cette propriété sont appelés : algorithmes de configuration efficace.

3.4.2 Oracle leader

L'oracle leader est une entité distribuée qui dote chaque processus p_i d'une fonction $leader()$. Cette fonction renvoit le nom du processus élu à chaque fois qu'elle est invoquée. Le processus élu peut être connu mais le moment de sa désignation comme $leader$ reste inconnu et chaque processus n'est pas en mesure de déduire la fin de pèriode d'asynchronisme.
Cet oracle noté Ω satisfait la propriété suivante :
– **Eventual leadership** : il existe un temps t et un processus correct p tel qu'après t chaque invocation de $leader()$ par n'importe quel processus correct renvoit p
Cette propriété fait référence à la notion du temps global qui n'est pas accessible par les processus.
Les algorithmes de consensus basés sur Ω sont décris dans [4,11,21,27] et sont $oracles\ efficaces$. L'algorthime décrit dans [11] est $oracle\text{-}efficace$ et $zero\text{-}degradant$.

3.4.3 Oracle probabiliste

l'oracle probabiliste dote chaque processus p_i d'une fonction $random$ qui donne comme sortie une valeur binaire aléatoirement choisie 0 (respectivement 1) avec une probabilité de $1/2$.Un algorithme de consensus binaire basé sur cet oracle est présenté dans [7].
Le consensus est atteint aprés deux étapes de communication avec la contrainte d'absence de processus défaillant initialement. Tous les processus vérifiant cette contrainte ont initialement la même valeur.

3.4.4 Détecteurs de défaillances

Il est nècessaire d'avoir des informations sur l'état operationnel de processus afin d'assurer des services fiables. L'idée de dètecteur de défaillances a surgit afin de circonvenir l'impossibilité de FLP [12] en dotant chaque processus de registres d'informations nècessaires qui renseignent chaque processus sur l'état des autres processus. Cela permet d'assurer un service tolérant aux fautes même avec la présence de défaillances dans un système asynchrone. Le détecteur de défailances couvre l'effet de l'asychronisme tout en augmentant la

synchronie du système grâce aux modules d'informations.Initialement le système est synchrone mais peut devenir purement asynchrone à tout moment suite à la surcharge imprévue d'un processus quelconque par exemple.En suivant, l'exemple cité, un processus distant ne peut distinguer un processus lent d'un processus défaillant, ce qui permet faussement de le considérer comme étant défaillant malgré qu'il ne l'est pas. Cela nous renvoit à reconsidérer minutieusement la programmation avec $timeout$ et la difficulté de dèterminer un $timeout$ approprié($timeout = clock + p$). L'approche avec $timeout$ se limite à des communications synchrones bloquantes alors que les systèmes actuels ne sont pas purement synchrones donc il faudra considérer de concevoir des systèmes tolérant des périodes d'asynchronisme avec des communications non bloquantes. C'est dans cette perspective que la notion de détecteur de défaillance avait été introduite.
Les détecteurs de défaillances sont des dispositifs abstraits qui offrent des informations sur l'état des processus. Un détecteur de défaillances selon [10] peut être perçu comme un ensemble de modules associés à chaque processus. Ces modules fournissent à chaque processus une liste de processus suspectés d'être défaillants. Toutefois,la liste des processus suspectés peut être différente d'un processus à un autre ($non\ determinisme$).
Nous distinguons deux types de détecteurs de défaillances : $fiables$ et $non\ fiable$. Pour notre étude, nous tenons en compte les détecteurs de défaillances nos fiables.

3.4.5 Popriétés d'un détecteur de défaillances

Un détecteur de défaillances est défini par deux propriétés : la **complétude** et la **précision**

3.4.5.1 Complétude

La complétude d'un détecteur de défaillances signifie que les défaillances dans un système doivent être détectées.On distingue deux types de complétudes : $forte$ et $faible$.

3.4.5.1.a Complétude forte

Tous les processus défaillants sont finalement et définitivement suspectés par tous les processus corrects, c'est-à-dire, il existe un instant après lequel un processus défaillant est suspecté d'une maniére permanente par tous les processus corrects. Formellement, un détecteur de défaillance D satisfait la complétude forte si :
$\forall F, \forall H \in D(F), \exists\, t \in \tau, \forall p \in crashed(F), \forall q \in correct(F), \forall t' \geq t : p \in H(q, t')$.

3.4.5.1.b Complétude faible

Pour tout processus défaillant, il existe au moins un processus correct qui le suspecte, c'est à dire il existe un instant aprés lequel tout processus défaillant est suspecté d'une maniére permanente par au moins un processus correct. Formellement, le détecteur de défaillance D satisfait la complétude faible si :
$\forall F, \forall H \in D(F), \exists t \in \tau, \forall p \in crashed(F), \exists q \in correct(F), \forall t' \geq t : p \in H(q, t')$

3.4.5.2 Précision

La précision d'un détecteur de défaillances indique le degré de fiabilité de la non suspicion des processus corrects(un processus correct ne doit pas être suspecté) et uniquement les processus défaillants doivent être détectés. On distingue quatre cas de précision : $forte$, $faible$, $ultime\text{-}forte$, $ultime\text{-}faible$.

3.4.5.2.a Précision forte

Aucun processus n'est suspecté avant qu'il ne devienne défaillant. Formellement,le détecteur de défaillances D satisfait la précision forte si :
$\forall F, \forall H \in D(F), \forall t \in \tau, \forall p, q \in \Pi - F(t) : p \notin H(q,t)$.

3.4.5.2.b Précision faible

Il existe au moins un processus correct qui n'est jamais suspecté. Formellement, le dètecteur de défaillances D satisfait la précision faible si :
$\forall F, \forall H \in D(F), \exists p \in correct(F), \forall t \in \tau, \forall q \in \Pi - F(t) : p \notin H(q,t)$.

3.4.5.2.c Précision ultime forte

Il existe un instant aprés lequel tous les processus corrects ne sont pas suspectés par un autre processus correct. Formellement, le détecteur de défaillances D satisfait la précision ultime forte si :
$\forall F, \forall H \in D(F), \exists t \in \tau, \forall t' \geq t \; \forall p, q \in correct(F) : p \notin H(q,t')$.

3.4.5.2.d Précision ultime faible

Il existe un instant aprés lequel il existe des processus corrects qui ne sont pas suspectés par un processus correct. Formellement, le déteteur de défaillances D satisfait la précision ultime faible si :
$\forall F, \forall H \; D(F)$, $\exists t \in \tau$, $\exists p \in correct(F), \forall t' \geq t, \forall q \in correct(F) : p \notin H(q,t')$

3.4.6 Classes des détecteurs de défaillances

Nous introduisons les différentes classes de détecteurs de défaillances ayant été définies par $Chandra$ et $Toueg$ [10] à base des propriétés de complétude et de précision. Le tableau ci-dessous regroupe les huits classes de détecteurs de défaillances. $Chandra$ et $Toueg$ ont montré qu'il existe une équivalence entre les classes de détecteurs de défaillances. Cette équivalence provient de la transformation permettant d'avoir la propriété de complétude forte à partir de la complétude faible. Nous allons voir les techniques de réductibilité entre les classes dans la section suivante. La réductibilité permet de restreindre l'étude aux quatres classes caractérisées par la propriété de la complétude forte : P,S,$\diamond P$,$\diamond S$

	Précision-forte	Précision-faible	Précision-ultime-forte	Précision-ultime-faible
Complétude-forte	parfait P	fort S	eventuellement parfait $\diamond P$	eventuellement fort $\diamond S$
Complétude-faible	quasi-parfait Q	faible W	eventuellment quasi-parfait $\diamond Q$	eventuellement faible $\diamond W$

TABLE 3.1: Classes des détecteurs de défaillances de $Chandra$ et $Toueg$ [10]

3.4.6.1 Détecteurs de défaillances fiables

Un détecteur de défaillances est dit fiable si seulement s'il ne commet pas d'erreurs : une erreur comme prendre un processus correct pour un processus défaillant et vice versa. Ce cas de figure ne se produit que si le système est purement synchrone.
Le detecteur de défaillance vérifiant les deux propriétés : complétude forte et précision forte est appelé détecteur de défaillance parfait noté P. Ce détecteur de défaillances ne suspecte pas faussement un processus et éventuellement détecte toutes les défaillances.
La propriété de complétude forte est garantie du fait que le processus défaillant n'envoit pas de messages de type $heartbeat$ et éventuellement chaque processus va remarquer l'absence de message $heartbeat$ [25].
La propriété de précision forte est garantie du fait que si un processus est $alive$ tous les processus vont recevoir $heartbeat$ à temps tel que le temps maximum entre deux $heartbeats$ est de $\gamma + \Delta$.

3.4.6.2 Détecteurs de défaillances non fiables

La différence entre ce type de détecteurs de défaillances et celui décrit précèdemment est que ce dernier ne fonctionne que sur un système synchrone contrairement aux détecteurs de défaillances non fiables fonctionnent sur un système éventuellement synchrone. Un détecteur de défaillances éventuellement parfait noté $\diamond P$, ce détecteur de défaillances n'est parfait qu'aprés une période de temps finie, c'est-à-dire, durant cette période du temps il peut se comporter arbitrairement.
L'idée de l'algorithme de consensus utilisant $\diamond P$ est de maintenir les deux propriétés de consensus : accord et validité toujours et l'algorithme se termine (propriété de terminaison) uniquement si le détecteur de défaillances associé s'arrête de commettre des erreurs.On appelle ce type d'algorithme , des algorithmes $indulgents$ car ils sont indulgents envers les erreurs de leurs détecteurs de défaillances.Nous allons étudier profondément les algorithmes de consensus indulgents dans le chapitre suivant en montrant comment le problème de consensus peut être résolu même avec la présence de crash dans un systéme partiellement synchrone.
Le principe des détecteurs de défaillances non fiable est comme suit : Chaque module de dé-

tection de défaillances fournit l'ensemble des processus qu'il suspecte d'être défaillants.
L'historique H d'un détecteur de défaillances est une fonction de l'ensemble $\Pi * T$ vers 2^{Π} dont $H(p,t)$ est la valeur du détecteur de défaillances du processus p à l'instant t dans H.Si $q \in H(p,t)$ alors p $suspecte$ q à l'instant t dans H.La liste des processus suspectés peut varier d'un processus à un autre, c'est-à-dire si $p \neq q$ alors $H(p,t) \neq H(q,t)$ est possible.
Les détecteurs de défaillances non fiables sont plus réalistes que les détecteurs de défaillances fiables d'un point de vue implémentation étant donné que les propriétés des détecteurs de défaillances non fiables sont plus garanties en pratique car les systèmes caractérisés par le synchronisme partiel permettent d'implementer les détecteurs de défaillances non fiables.
Le détecteur de défaillances èventuellement parfait $\diamond P$ peut être implementé dans un système partiellement synchrone où le détecteur de défaillances commence avec une approximation ρ' (inconnu) de la valeur réelle ρ.Lorsque l'algorithme constate qu'il a commis une erreur : réception d'un message provenant d'un processus suspecté d'être défaillant, la valeur ρ' est augmentée et lorsque le systéme devient synchrone et $\rho' > \rho$ le détecteur de défaillances arrête de commettre des erreurs et devient parfait P [10].

3.4.6.2.a Propriétés des détecteurs de défaillances non fiables

Un détecteur de défaillances est non $fiable$ s'il ne permet à aucun processus de distinguer si les processus distants sont défaillants ou pas[3]. Nous introduisons trois classes des détecteurs défaillances non fiables et dans le chapitre quatre nous allons définir trois classes d'algorithmes indulgents associées aux classes de détecteurs de défaillances non fiables.

(a) **Non fiablité complète** notée $\square U$: Un détecteur de défaillance D est complétement non fiable si pour chaque paire de modèles de défaillances F et F', $D(F) = D(F')$.

(b) **Non fiablité forte** notée ∇U : Un détecteur de défaillance D est fortement non fiable si pour chaque paire de modèles de défaillances F et F' ,pour chaque historique $H \in D(F)$, pour chaque temps $t_k \in \phi$, il existe un historique de détecteur de défaillances $H' \in D(F')$ tel que $\forall\, t \leq t_{k'}\ \forall p_i \in \Pi, H'(p_i,t) = H(p_i,t)$.

(c) **Non fiabilité faible** notée ΔU : Un détecteur de défaillance D est faiblement non fiable (ou non fiable) si pour chaque historique de modéle de défaillances F,pour chaque historique $H \in D(F)$,pour chaque modèle de défaillances F' qui $couvre$ F, pour chaque temps $t_k \in \phi$, il existe un historique de détecteur de défaillances $H' \in D(F')$ tel que $\forall t \leq t_k, \forall p_i \in \Pi, H'(p_i,t) = H(p_i,t)$

3.4.7 Réductibilité

Au lieu de travailler sur les huits classes de détecteurs de défaillances, un nouveau concept a été introduit pour ne se focaliser que sur quatres classes. Il s'agait de la réductibilité.
On nomme $T_{D \to D'}$ "algorithme de réduction" transformant un détecteur de défaillances D en un autre détecteur de défaillances D'.Etant donné cet algorithme de réduction, tout probléme résolu avec le détecteur de défaillances D peut être résolu avec D'. S'il existe un

algortihme$T_{D\to D'}$ qui transforme D en D', on écrit $D \geq D'$ et on dit que D' est $reductible$ à D, et on dit aussi que D' est $plus\ faible$ que D. Si D$\geq$ D' et D'$\geq$D on dit écrit D$\cong$ D' et on dit que D et D' sont équivalents.Similairement, étant donné deux classes de détecteurs de défaillances C et C', si pour chaque détecteur de défaillances $D \in C$, il existe un détecteur de défaillances $D' \in C'$ tel que $D \geq D'$,on écrit $C \geq C'$ et on dit que C' est $plus\ faible$ que C. De plus, si C$\geq$C',alors si un problème peut être résolu en utilisant C' alors il peut aussi être résolu en utilisant C.Si $C \geq C'$ et $C' \geq C$,on écrit $C \cong C'$ sont équivalentes.
La figure ci-dessous schématise les relations existantes entre les classes de dètecteurs de défaillances. Toutes les relations d'équivalence ou de réduction sont représentées respetivement par un trait plein ou par une flèche pointillée.Les classes qui ne sont pas reliées sont incomparables. D'aprés le schéma, toutes les classes de complétude faible sont équivalentes aux classes de complétudes fortes. La classe $\diamond S$ est la plus faible pour résoudre le consensus [10].

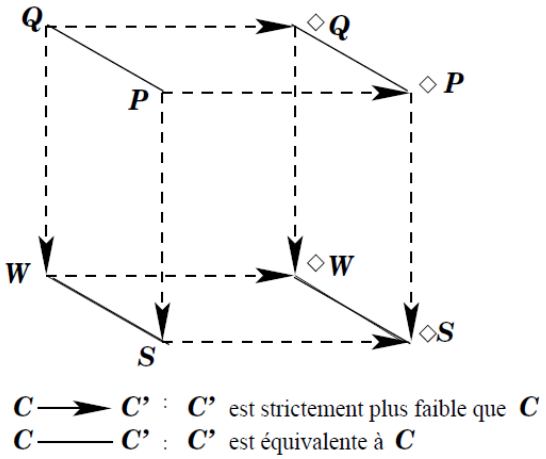

FIGURE 3.1: Réductibilité des classes de détecteurs de défaillances

3.5 Détecteur de défaillances le plus faible $\diamond S$ pour résoudre le consensus

Le détecteur de défaillances le plus faible pour résoudre le consensus est de la classe de détecteurs de défaillances non fiable $\diamond S$. Le nombre maximum de processus défaillants dans le système doit être strictement infèrieur à la moitié du nombre de processus participant au consensus [10].$\lceil (n+1/2) \rceil$ processus corrects est nécessaire.
Un détecteur de défaillance de classe $\diamond S$ peut rajouter éventuellement tous les processus corrects à sa liste de suspects. Toutefois, il existe un processus correct et un instant après lequel ce dernier n'est plus suspecté. L'algorithme presenté dans [10] est basé sur le paradigme du coordinateur à tour de rôle ; il procède par tours successifs asynchrones. Durant le tour r, tous

les processus savent que le processus coordinateur est le processus c=(r *mod* n)+1.Durant ce tour, les processus communiquent uniquement avec c. Le nombre de tours qu'effectue un processus dépend des défaillances et du comportement des détecteurs de défaillances. A chaque fois qu'un processus devient coordinateur, il tente de dèterminer une valeur cohérente. S'il est correct et n'est suspecté par aucun processus, il décide une valeur et la diffuse aux autres processus.
Chaque tour de cet algorithme de consensus est divisé en quatres phases asynchrones comme suit :

(a) Chaque processus p maintient une variable $estimate_p$ (initialisée à sa valeur initiale v_i) .Dans cette phase, chaque processus p envoit au coordinateur courant la valeur de sa variable $estimate_p$ estampillée du numéro du tour auquel elle appartient.

(b) c collecte $\lceil (n+1)/2 \rceil$ estimations proposées par les processus. Il choisit une des estimations reçues dont l'estampille est la plus grande et met à jour $estimate_c$ qu'il envoit à tous les processus.

(c) Chaque processus p attend l'estimation du coordinateur. Deux cas peuvent se présenter : le processus suspecte le coordinateur courant d'être défaillant après avoir consulté son détecteur de défaillances ou bien il reçoit l'estimation du coordinateur.
Dans le premier cas,il envoie un acquittement négatif au coordinateur courant et passe au tour suivant ($r+1$).
Dans le second cas,il envoit un acquittement positif au coordinateur et met à jour sa variable $estimate_c$ à la valeur de l'estimation reçue du coordinateur.

(d) Cette phase est effectuée par le coordinateur. Il attend une majorité d'aquittements.
Le premier cas : tous les acquittements reçus sont positifs, c'est-à-dire la majorité des processus a adopté l'estimation du coordinateur. Ce dernier verrouille la valeur de sa variable $estimate_c$ et diffuse d'une maniere fiable la valeur de décision. Le second cas : au moins un message d'acquittement négatif, c'est-à-dire, un des processus a suspecté le coordinateur. Le coordinateur passe au tour suivant sans décider.

3.6 Conclusion

Dans ce chapitre, nous avons abordé le problème de consensus et nous avons caractérisé ce dernier dans un environnement asynchrone dont il est impossible de résoudre [12] même avec la présence d'une seule défaillance. Par la suite, nous avons évoqué une famille d'oracles pour circonvenir cette impossibilité. En premier lieu, nous avons commencé par étudier les propriétés des détecteurs de défaillances et leurs classes. Ensuite, nous avons illustré la réductibilité des classes des détecteurs de défaillances. Pour terminer, nous avons présenté le détecteur de défaillances le plus faible pour résoudre le consensus.
Résoudre le consensus dans un système distribué asynchrone n'est possible qu'avec les détecteurs de défaillances non fiables.Toutefois, ces derniers commettent des erreurs suite à des périodes d'asynchronisme dont les processus ne peuvent distinguer un processus lent d'un

processus défaillant. Par conséquent, l'établissement d'un timeout n'est pas efficace, il induit le système aux erreurs. Pour cette raison, Guerraoui [3] a introduit les algorithmes indulgents pour remedier aux fausses suspicions des détecteurs de défaillances non fiables. Ces algorithmes tolérent des périodes d'asynchronisme, c'est-à-dire ils sont indulgents envers leur détecteurs de défaillances. Dans le chapitre suivant nous allons évoquer minutieusement cet famille d'algorithmes.

Chapitre 4

Travaux et résultats antérieurs du consensus indlulgent

4.1 Introduction et historique de l'indulgence

Le concept de l'algorithme indulgent a été introduit pour la premiere fois par Guerraoui [3] dans l'objectif d'élaborer des protocoles qui tolèrent des périodes d'asynchronisme et remèdier aux fausses suspicions commises par les détecteurs de défaillances non fiables.
L'algorithme indulgent est l'algorithme le plus tolérant aux fautes et offre des services plus fiables.Depuis cette contribution majeure, plusieurs recherches ont suivi ce chemin dans la perspective de trouver des algorithmes indulgents plus efficaces et optimaux en nombre de tours et en nombres de messages échangés. Cependant la quête d'algorithme induglent optimal en en nombre de tours et en nombre de messages échangés est toujours à l'ordre du jour.En 2013 [18], la question portant sur l'existence d'$early-deciding$ et $early-stopping$ algorithmes a été étudiée avec attention. La première propriété est importante dans le sens où la pluspart des exécutions sont caractérisées soit par l'absence de fautes ou par l'existence de quelques fautes uniquement. La deuxième propriété $early-stopping$ est inhérente à la première dans le sens où l'algorithme se termine une fois la décision globale a été prise(une étape de plus après la décision globale).
La complexité en nombre de messages échangés pour atteindre le consensus sans les contraintes $early-deciding$ et $stopping-early$ a été étudiée avec discernement, notamment le cas de la borne minimale et maximale.Reischuk à traiter le problème de l'amélioration de l'efficacité en se basant sur la complexité moyenne [25]. Puis Perry et Toueg [23] se sont focalisés sur le nombre de fautes dans un modèle de défaillances faible puis repris par Raynal [21] par la suite. Fitzi et Martin[12] ont introduit la complexité minimale en nombre de messages $\Omega(nt)$ avec une petite probabilité d'erreur. Keidar et Rajsbaum[15] ont prouvé que les algorithmes rèsolvant le consensus uniforme (la valeur de décision doit étre la même pour tous les processus qui décident et même pour ceux qui deviennent défaillants après la décision) dans le cas de crash ne peuvent décider avant $min(f+2, t+1)$ tours. Cependant, Charron et schiper[2] ont prouvé que cette borne ne concerne pas les algorithmes de consensus pour le cas

de crash dont il est possible de décider à la fin du tour $f + 1$. Raynal[24] prénsente une étude compléte des algorithmes $early - deciding$ dans le modéle de crash. Jusque la, aucun de ces algorithmes n'est $early - deciding$ et $early - stopping$ dans le sens où ils s'exècutent pour au moins $t + 1$ tours même s'il y a peu de fautes.
Dans ce chapitre, nous allons en premier lieu définir l'algorithme indulgent, puis nous introduisons les travaux antérieurs et les résultats les plus récents (plus efficaces). Ensuite, nous allons présenter notre contribution qui consiste en deux protocoles de consensus indulgents optimaux en nombre de tours et en nombres de messages échangés. Puis, nous procédons à la démonstration de la validité de nos résultats. Nous terminerons par un ensemble de problèmes ouverts et perspectives.

4.2 Définition de l'algorithme indulgent

Un algorithme indulgent est un algorithme distribué qui tolère les fausses suspicions commises par un détecteur de défaillances non fiables, c'est-à-dire, l'algorithme est indulgent envers son détecteur de défaillances. Un algorithme est indulgent envers son détecteur comme suit : pour une période de temps arbitraire, cet algorithme tolére les erreurs commises par son détecteur. Un détecteur de défaillances est non fiable étant donné que l'établissement d'un timeout de petite valeur assure la réaction rapide aux défaillances mais augmente la probabilité de fausse suspicion durant la période d'instabilité, et l'établissement d'une grande valeur de timeout réduit la valeur de fausse suspicion et réagit lentement aux défaillances[3,5]. De ce fait, un détecteur de défaillances non fiable ne peut distinguer un processus lent d'un processus défaillant. Les détecteurs de défaillances ne sont fiables que dans des exècutions synchrones.
On définit trois classes d'indulgence correspondantes aux classes des détecteurs de défaillances non fiables :

(a) L'indulgence faible correspondante à la classe ΔU (détecteur de défaillance non fiable faible)

(b) L'indulgence forte correspodante à la classe ∇U (détecteur de défaillance non fiable fort).

(c) L'indulgence complète correspondante à la classe $\Box U$ (détecteur de défaillance non fiable complet).

On pose A un algorithme indulgent utilisant un détecteur de défaillances D. On dit que A est complètement indulgent si $D \in \Box U$,fortement indulgent si $D \in \nabla U$ et faiblement indulgent si $D \in \Delta U$. Ces classes sont réliées entre elles de la façon suivante : $\Box U \subset \nabla U \subset \Delta U$.
Cependant, pas de solution indulgente pour les algorithmes dotés de propriétés de divergence. On dit qu'un algorithme de consensus est divergent si dans un environnement E $(t \geq (n/2))$ peuvent être défaillants. C'est-à-dire, aucun algorihme ne peut résoudre le consensus en utilisant le détecteur de défaillances non fiable si la majorité des processus est défaillante.

4.3 Modèles de communication pour résoudre le consensus indulgent

On considère deux modèles de crash basés sur les tours :
– Modèle synchrone noté SCS
– Modèle partiellement synchrone noté ESC
Dans les deux cas cités, si un processus entre dans un tour, deux choses peuvent se produire : soit il complète le tour, soit il devient défaillant.Chaque tour est caractérisé par une phase d'envoi et une phase de réception de message (tel que détaillé dans la section les détecteur de défaillances non fiables). Dans la phase $receive$ de chaque tour k, si un processus p_i ne reçoit pas le message du tour k provenant de certains processus p_j, on dit que p_i suspecte p_j dans le tour k. On dit qu'un message m envoyé par un processsus p_i à p_j est perdu si p_j ne reçoit pas m dans dans cette exécution.
Les deux modèles sont décris comme suit :
– dans le SCS, si un processus p_i devient défaillant dans un certain tour k,le sous-ensemble de messages envoyé par p_i dans ce tour peut être perdu,et le reste des messages envoyé par p_i sont reçus dans le méme tour. Si p_i ne devient pas défaillant dans le tour k, chaque processus complétant le tour k reçoit le message provenant de p_i associé au tour k.
– Dans ESC, l'exécution peut être $asynchrone$ pour un nombre fini de tours mais devient eventuellement $synchrone$. Contrairement au SCS, dans ESC un message peut être retardé pour une période finie de tours (message sera reçu dans un tour superieur qu'au tour où il a été envoyé). Pour cela les propriétés suivantes doivent être vérifiées :

  (a) $t - resilience$: chaque processus complétant un tour k,reçoit dans le même tour des messages par au moins $n - t$ processus.

  (b) $canaux\ fiables$: les messages échangés entre les processus corrects ne sont jamais perdus mais peuvent être retardés pour un nombre arbitraire de tours (fini).

  (c) Le $synchronisme\ partiel$: il existe un nombre fini de tours K mais inconnu, tel que $K \geq k$ si :
  – (a) un processus p_i devient défaillant dans un tour k, alors n'importe quel sous-ensemble de messages envoyé par p_i dans ce tour peut être perdu, et le reste des messages est reçu dans le même tour.
  – (b) un processus p_i ne devient pas défaillant dans un tour k, alors chaque processus complétant le tour k, reçoit le message provenant de p_i associé au tour k.

  Remarque :les exécutions synchrones de ESC fournissent plus de garantie que celles de SCS dans le sens suivant : les messages envoyés par un processus p_i aux processus corrects dans le tour ou p_i devient défaillant, les messages sont uniquement retardés dans l'exécution synchrone de ESC par contre ces messages peuvent être perdus dans une exécution synchrone de SCS. D'où les ESC sont plus tolérants aux fautes.Une exécution synchrone est une exécution où le déteteur de défaillances est fiable [5]. Par contre dans une exécution asynchrone des fausses suspicions peuvent avoir lieu.

4.4 Protocole de consensus indulgent de Dutta et Guerraoui

Cet Algorithme de consensus indulgent proposé par [5] est considérablement le plus efficace dans les exécutions synchrones au pire des cas que tous les algorithmes de consensus $\diamond S-based-consensus$ (qui le précede). Ce protocole basé sur le détecteur de défaillances non fiable $\diamond S$, atteint une décision globale au tour $t+2$ dans les éxecutions synchrones. Cependant, l'algorithme $\diamond S-based-consensus$ de [20] était le protocole le plus efficace(jsuque la) qui nécessite $2t+2$ tours pour atteindre une décision globale contrairement à l'algorithme de Dutta et Guerraoui ($t+2$).
La complexité du protocole de [5] est de $O(t)$ tours ($t+3$ au plus) et de $\Omega(n^2)$ pour le nombre de messages échangés. l'algorithme de consensus indulgent A_{t+2} [5] associé se trouve à la page 92 (Fig.2).

4.5 Protocoles de consensus indulgents de Gilbert et al.

Gilbert et al.[17] ont presenté deux algorithmes de consensus indulgents.Les deux protocoles améliorent considérablement les protocoles les plus efficaces [5,12,4,8], l'un d'eux est présenté dans la section précedente, ayant une complexité d'au moins $\Omega(n^2)$ en terme de messages échangés. Lorsque le systéme est synchrone, les deux algorithmes garantissent une bonne performance en terme de tours et en nombre de messages échangés. Le premier (décrit dans la section 4 page 288 [17]) est optimal en nombre de messages échangés $O(n)$ et termine au bout de $O(n^{1+\epsilon})$. Le deuxiéme (décrit dans la section 5 page 289 [17]) est optimal en nombre de tours $O(f)$ tel que $f \leq t$ et a une complexité de $0(n\ log^6\ n)$ en nombre de messages échangés. Les deux protocoles atteignent une performance efficace si l'éxecution est synchrone. Dans le cas contraire, les processus se mettent à synchroniser leurs vues sur le système à travers le protocol $gossip$.
Pour terminer, on récapitule en soulignant la complexité en nombre de messages échangés et en nombre de tours de ces deux algorithmes comme suit :
– Algorithme 1 : $O(n)$ en nombre de messages échangés et $O(n^{1+\epsilon})$ tours
– Algotithme 2 : $O(n.log^6.n)$ en nombre de messages échangés et $O(f)$ tours

Chapitre 5

Protocoles pour le consensus indulgent avec communication optimale

Dans notre étude, nous proposons deux protocoles de consensus induglents qui résolvent le consensus de façon optimale : optimal en nombre de tours et efficace en terme de nombre de messages échangés.Pour atteindre à la fois l'efficacité en nombre de tours et en nombre de messages échangés, nos protocoles utilisent $(t+1)$ coordinateurs pour envoyer les messages dans chaque tour. Nos protocoles garantissent qu'au moins un processus décide au tour $t+2$ et la complexité en nombre de messages échangés est au plus $O(n.t)$. De plus, nos protocoles réduisent la taille des messages échangés.
Pour ce faire, nous allons en premier lieu présenter le protocole de consensus indulgent agissant dans un environnement où les défaillances peuvent éventuellement avoir lieu. Puis, nous allons montrer un cas particulier du premier protocole.Ce protocle agit efficacement dans un environnement dépourvu de défaillances. Pour terminer, nous allons démontrer formellement la validité de nos résultats par un ensemble de preuves et lemmes afin d'élaborer un théorème prouvant la correction de nos protocoles.

5.1 Protocole de consensus indulgent en présence de défaillances

La figure 6.1 présente un protocol de consensus basé sur le détecteur de défaillances $\diamond S$ dans un système éventuellement synchrone lorsque $0 < t < (n/2)$. Initialement, $t+1$ processus sont désignés comme coordinateurs (ligne 1). On suppose que les $t+1$ premiers processus sont les coordinateurs. Uniquement les coordinateurs peuvent envoyer leurs estimations dans les premiers $t+1$ tours et chaque processus met à jour son estimation à la valeur minimale. La valeur minimale est calculée suite à la réception de toutes les estimations dans le tour puis met à jour sa variable $S[1..(t+1)]$. $S[1..(t+1)]$ est un tableau utilisé par chaque processus pour suspecter n'importe quel processus $\in A$. On utilise cette variable pour minimiser la taille du message échangé.Au lieu d'envoyer la liste des processus suspectés parmi l'ensemble A_i, uniquement un tableau de $t+1$ de $t+1$ bits est envoyé. Si

un processus p_i ne reçoit pas le message $PROPOSE(r_i, *, *)$ de $p_j \in A_i$ et $S_i[j] = 0$ (veut dire que p_j n'est pas suspecté par p_i), alors p_i met sa variable $S_i[j]$ à 1. Cela veut dire que p_j est suspecté par p_i (line 4). A la ligne 5, si p_i reçoit le message $PROPOSE(r_i, v_j, S_j)$ de $P_j \in A_i$ alors il met à jour sa variable S_i en comparant sa variable avec celle de p_j, qui a pour objectif de mettre à jour sa liste contenant des processus suspecptés. A titre d'exemple, si $S_i[k] = 0$ alors p_k n'est pas suspecté sinon ($S_i[k] = 1$) p_k est suspecté.

Function Consensus(v_i)

Init : $r_i \leftarrow 1$; $S_i[1..(t+1)] \leftarrow 0$;

Task $T1$: % basic task %

repeat for $r_i \leq t+1$

——————————- first $t+1$ rounds ——————————

01 **let** A_i = a set of processes with $A_i = \{p_1, p_2, .., p_t, p_{t+1}\}$;
02 **if** $p_i \in A_i$ **then** *send* PROPOSE(r_i, v_i, S_i) to all ;
03 **wait until** received PROPOSE$(r_i, *, *)$ messages of round r_i ;
04 **if** ((p_i did not receive PROPOSE(r_i, v_j, S_j) message from $p_j \in A_i$)**and**($S_i[j] = 0$)) **then** $S_i[j] \leftarrow 1$;
05 **if** p_i receives PROPOSE (r_i, v_j, S_j) message from $p_j \in A_i$ **then** for each k, such that $1 \leq k \leq t+1$ **do** $S_i[k] \leftarrow max\{S_i[k], S_j[k]\}$;
07 **let** rec_i = a set of PROPOSE$(r_i, *, *)$ messages received from p_j such that $s_i[j] = 0$
08 **if** $|rec_i| = 0$ **then** $aux_i \leftarrow \perp$ **else** $aux_i \leftarrow \mathbf{min}\{v_k | propose(r_i, v_k, s_k) \in rec_i\}$;
06 $r_i \leftarrow r_i + 1$;

——————————————————————————————————————

end repeat

——————————————————- round $t+2$ ——————————————————-

09 *send* FILT(r_i, aux_i) to all processes in A_i ;
10 **if** $p_i \in A_i$ **then wait until** received FILT(r, aux) of round $t+2$ **store values** in V_i ;
11 **case** $(V_i = \{v\})$ **then** decide(v) ;
12 $(V_i = \{v, \perp\})$ **then** $est_i \leftarrow v$;
13 **endcase** ;
14 $r_i \leftarrow r_i + 1$;

——————————————————- round $t+3$ ——————————————————-

15 **if** p_i has decided **then** *send* DEC(r_i, v) to all except of itself **else** ESC(est_i) ;
16 **if** p_i has not decided **then**
17 **upon** *receipt* of DEC(r, v) : **wait until** $r_i \geq r$;
18 stop ESC(est_i) ; decide(v) *send* DEC(r_i, v) to all except of itself ;

FIGURE 5.1: Protocole de consensus indulgent en présence de défaillances

5.2 Protocole de consensus indulgent en absence de défaillances

La figure 6.2 présente un cas particulier du protocole décrit dans la figure 6.1. Ce protocole fonctionne correctement dans un environnement dépourvu de défaillances, c'est à dire $t = 0$. Il est basé sur le détecteur de défaillances P (parfait). Par conséquent, le protocole exige un seul coordinateur au lieu de $t + 1$ (absence de pannes).
Etant donné que les exècutions sont synchrones, le détecteur de défaillances associé à ce modèle est un détecteur de défaillances parfait (P). De plus, ce protocole ne nécessite pas un tableau de $t + 1$ bits comme le protocole précédent du moment que $t = 0$.
Ce protocole nécessite au plus $O(3)$ tours et $O(n)$ messages échangés pour terminer avec une décision globale.

Function Consensus(v_i)

Init :$r_i \leftarrow 1$;

——————————————— First round ———————————————

01 **if** $p_i = p_1$ **then** *send* PROPOSE$(1, v_i)$ to all ;
02 **wait until** received PROPOSE$(1, *)$ message from p_1 ;$r_i \leftarrow r_i + 1$;

——————————————— Second round ———————————————

03 **if** p_i did not receive PROPOSE$(1, *)$ message from p_1 **then** $aux_i \leftarrow \perp$;
04 **if** p_i receives PROPOSE $(1, v)$ message from p_1 **then** $aux_i \leftarrow v$;
05 *send* FILT$(2, aux_i)$ to p_1 ;
06 **if** $p_i = p_1$ **then wait until** received FILT(r, aux) from all processes ; **store values** in V_i ;
07 **case** $(V_i = \{v\})$ **then** decide(v) ;
08 $(V_i = \{v, \perp\})$ **then** $est_i \leftarrow v$;
09 **endcase** ;
10 $r_i \leftarrow r_i + 1$;

——————————————— Third round ———————————————

11 **if** p_i has decided **then** *send* DEC(r_i, v) to all except of itself **else** ESC(est_i) ;
12 **upon** *receipt* of DEC(r, v) : **wait until** $r_i \geq r$;
13 **if** p_i has not decided **then** stop ESC(est_i) ; decide(v) *send* DEC(r_i, v) to all except of itself ;

———————————————————————————————————————

FIGURE 5.2: Protocole de consensus indulgent en absence de défaillances

Dans le tableau ci-dessous nous illustrons la complexité algorithmique du protocole de Dutta et Guerraoui, Gilbert et al. , et notre protocole.

	complexité en messages	complexité en tours
Alg Dutta et Guerraoui [5]	$\Omega(n^2)$	$O(t)$
Alg 1 Gilbert et al. [17]	$O(n)$	$O(n^{1+\epsilon})$
Alg 2 Gilbert et al. [17]	$O(n.log^6.n)$	$O(f)$
Notre Algorithme	$O(n.t)$	$O(t)$

TABLE 5.1: Comparison des protocoles de consensus indulgents

5.3 Démonstration et preuve de validité des protocoles

La preuve de validité et les propriétés de terminaison de nos deux protocoles sont trés simples. Maintenant nous allons prouver les propriétés d'accord et de décision-rapide du protocole de la figure 6.1. Le protocole de la figure 6.2 n'est qu'un cas particulier du protocole 6.1, alors les preuves que nous allons présenter s'appliquent aussi sur le protocole de la figure 6.2.

Lemme 5.3.1. ***Terminaison***
Dans chaque exècution synchrone du protocole de la figure 6.1*, n'importe quel processus correct décidant, termine au plus au tour* $t + 3$*.*

Démonstration. A la fin du tour $t+1$ au moins un processus correct a sa variable $aux \neq \perp$ (ligne 8), et décide v au tour $t+2$ (ligne 11) car au plus $n-t$ messages seront reçus. Le reste des processus corrects qui n'ont pas pu décider au tour $t + 2$ vont exècuter la procédure $ESC(est_i)$ (ligne 15) avec la valeur invoquée v. Ils finissent par arrêter ESC et décident finalement (ligne 18). ■

Lemme 5.3.2. ***Validité***
$\forall$ *le processus décidant, un processus ne peut décider une valeur autre que celle qui a été proposée par certains processus.*

Démonstration. La valeur décidée dans la $ligne 11$ est le minimum des valeurs proposées à la $ligne\ 8$. Ces dernières ne peuvent être autres que celles proposées par les coordinateurs en $ligne\ 2$. D'où la valeur décidée est l'une des valeurs proposées. ■

Lemme 5.3.3. ***Intègrité***
Dans chaque exècution synchrone du protocole de la figure 6.1*, n'importe quel processus décidant, décide au plus une fois.*

Démonstration. Un processus décidant, décide soit à l'issue de la $ligne\ 11$ ou à l'issue de la $ligne\ 18$ au plus. Dans le premier cas, aprés avoir décidée, il envoie un message

$DEC(r_i, v)$ a tous les processus restant (autre que lui) et ne peut exécuter la $ligne$ 16 du moment qu'il a déja décidé. Dans le deuxième cas, un processus qui n'a pas pu décider au tour $t+2$ ($ligne$ 12) décide finalement au tour $t+3$ en exécutant l'instruction de la $ligne$ 16. D'où un processus décide soit au tour $t+2$ soit au tour $t+3$. ■

Lemme 5.3.4. ***Accord***
La propriété d'accord assure qu'il n'y ait pas deux processus qui décident différement.

Démonstration. Si aucun processus ne décide, alors la propiété d'accord se tient trivialement. On considère le tour le plus bas k dont quelques processus décident. On dit qu'un processus p_i décide v dans le tour k. Si $k > t+2$ alors aucun processus ne décide dans les premiers $t+2$ tours, d'où aucun processus n'envoit un message $DECIDE$ dans la line 15. Par conséquent, la propriété d'accord résulte de la propriété de l'accord de ESC. On considère alors le cas lorsque $k = t+2$. (à partir de l'algorithme,$t+2$ est le tour le plus bas dont un processus peut décider).
Etant donné p_i décide v au tour $t+2$, chaque message reçu par p_i dans le tour $t+2$ a la valeur $aux_i \neq \perp$ et p_i reçoit au moins un message avec $aux = v$. A partir de la propriété de ESC, il y a au moins $n - t > (n/2)$ messages reçus par p_i dans le tour $t+2$. Par conséquent,au moins une minorité de processus envoie le message FILT(r, aux) avec $aux \neq \perp$. De plus, de la propriété d'élimination de la ligne 4,5,6,et pour chaque $FILT$ message, $aux \in v,\perp$. N'importe quel processus $\in A_i$ qui complète le tour $t+2$ reçoit au moins une majorité de message $FILT$ parceque $n - t > (n/2)$, et ainsi reçoit au moins un message avec $(aux = v)$. Par conséquent, si un processus décide dans le tour $t+2$ alors ce processus décide v et envoie un message $DECESION$ avec la valeur décidée v dans le tour $t+3$, et si le processus invoque $ESC()$ alors la valeur invoquée est v. Par la validité de la propriété de l'algorithme ESC, aucun processus ne peut décider une valeur autre que v. ■

Lemme 5.3.5. ***Décision rapide***
Dans chaque exècution synchrone du protocole de la figure 6.1*, n'importe quel processus décidant, décide au plus au tour* $t+3$*.*

Démonstration. On suppose par contradiction qu'il y a une exècution synchrone dans laquelle certains processus de l'ensemble A_i complètent le tour $t+2$ sans pouvoir décider. Alors, suivant la ligne 9 certains processus p_i ont envoyé le message $FILT$ avec $aux = \perp$. Par conséquent l'exècution synchrone $|rec_i| = 0$ (ligne 7).
De ce fait, on conclut que plus de t processus sont défaillants avant d'avoir complété le tour $t+1$, alors contradiction. ■

Théorème 5.3.6. *Il existe un protocole de consensus indulgent s'exècutant avec une complexité de* $O(n.t)$ *en terme de nombre de messages échangés et de* $O(t)$ *tours dans les exècutions synchrones.*
La preuve découle directement du ***lemme 5.3.1****,* ***lemme 5.3.2****,* ***lemme 5.3.3****,* ***lemme 5.3.4****, du* ***lemme 5.3.5*** *et le fait que dans chaque étape de communication du protocole il y a au plus* $(n.t)$ *messages échangés. Dans les* $t+1$ *tours,* $t+1$ *messages (au plus) sont envoyés*

à n processus ($ligne2$). Dans le tour $t+2$, n messages sont envoyés aux $t+1$ coordinateurs (au plus) $ligne9$. Enfin, dans le tour $t+3$ les processus qui ont décidé envoient $DEC(r_i, v)$ à tous les processus et ceux qui n'ont pas encore décidé finissent par décider, d'où $n.t$ messages (au plus) échangés dans chaque tour.

Chapitre 6

Problèmes ouverts et travaux de recherche en cours

Dans cette partie, nous allons évoquer certains axes de recherches, que nous jugeons fondamentaux dans le domaine du calcul distribué,les résoudre revient à résoudre aussi certaines problématiques qui sont intrinséquement liées à ces principaux problèmes qui sont la quintessence des systèmes distribués fiables tolérant aux fautes. Pour cela, ces problèmes doivent étre abordés avec perspicacité. Avant d'énumerer les problémes ouverts liés à notre thématique et nos travaux de recherches en cours, nous allons définir deux nouveaux concepts[18] portant sur la décision et la terminaison de l'algorihme de consensus afin de mieux élucider les problèmatiques liées à ces deux concepts.

6.1 Algorithme : early-deciding

Un algorithme est dit $early - deciding$, si dans chaque exécution il minimise le nombre de tours jusqu'à ce que tous les processus décident. Le nombre de tours dépend du nombre de fautes actuelles f tel que $f \leq t$. Cette propriété est trés importante et souhaitable, étant donné que dans la pluspart des scénarios il est raisonnable de supposer que la plus part des exècutions sont caractérisées par l'absence ou la présence de quelques fautes uniquement[18].

6.2 Algorithme : early-stopping

Un algorithme est dit $early - stopping$ s'il est $early - deciding$ et se termine après une étape au plus. C'est-à-dire, les processus s'arrêtent de communiquer une fois qu'ils auront partagé la valeur décidée avec tout le reste des processus.
La borne inférieure pour qu'un algorithme de consensus termine en considérant ces deux contraintes est de $min(f + 2, t + 1)$ tours [7,8,18].

6.3 Problèmes ouverts

Nous présentons une liste de problèmes ouverts en tenant en compte le nombre de messages échangés et le nombre de tours nécessaires pour résoudre ces problèmes.

- Peut-on trouver un algorithme de consensus optimal en nombre de tours de$O(t)$ et $O(n.t)$ en nombre de messages échangés avec les contraintes $early - deciding$ et $early - stopping$ dans le cas de présence de défaillances franches ?
- $Early - stopping$ est-il plus coûteux en terme de communication ? [18]
- La borne minimale en nombre de messages échangés est de $\Omega(n^2 f)$ tolérant jusqu'à t défaillances byzantines.Nous posons la question suivante : peut-on réduire sa complexité à $O(f)$ en nombre de tours et $O(n^2 f)$ en terme de messages échangés avec les contraintes $early - deciding$ et $early - deciding$?
- Peut-on trouver un protocole de consensus indulgent pour le cas de défaillances byzantines ?
- Peut-on trouver un protocle d'indulgence résolvant le k-accord ? si oui, peut-il être $early-deciding$ et $early - stopping$

6.4 Recherches en cours

Après avoir résolu dans ce travail de mémoire de Master 2 le problème portant sur la recherche d'un protocole de consensus indulgent optimal en nombre de tours et efficace en nombre de messages échangés, nous avons abordé les thématiques suivantes :

- Peut-on trouver un algorithme de consensus optimal en nombre de tours de$O(t)$ et $O(n.t)$ en nombre de messages échangés avec les contraintes $early-deciding$ et $early-stopping$ dans le cas de présence de défaillances franches ? (en cours)
- La recherche du détecteur de défaillance le plus faible pour résoudre le probléme du k-accord dans les systémes distribués asynchrones. Notre piste consiste à chercher le synchronisme minimal en se basant sur le modèle $M^{sink(x)}$ [33]. (en cours)

Conclusion générale

Au cours de ce mémoire de Master 2 le problème de consensus a été étudié avec les différents oracles :leader,probabiliste et détecteurs de défaillances. Ces derniers sont apparus pour circonvenir le résultat d'impossibilité [12] afin de pouvoir résoudre le consensus dans un système distribué purement asynchrone même avec la présence d'une seule défaillance uniquement.Afin de palier cette impossibilité, les détecteurs de défaillances non fiables ont été introduits [10]. Cependant, ces derniers commettent des erreurs, il s'agit de fausses suspicions qu'un détecteur de défaillances au niveau de chaque processus peut faire : prendre un processus correct pour un processus défaillant et vice-versa, d'où la difficulté de distinguer un processus lent d'un processus défaillant. Pour cela nous avons étudié avec attention les détecteurs de défaillances non fiables afin de remédier aux fausses suspicions. Pour ce faire nous avons analysé les protocoles de consensus indulgents les plus efficaces jusqu'à présent qui permettent de tolérer des périodes d'asynchronisme et être tolérants envers les détecteurs de défaillances non fiables.
Cependant, ces protocoles ne sont pas aussi efficaces et des améliorations peuvent être ajoutées afin d'avoir des protocoles de consensus indulgents efficaces en temps et en nombres de messages échangés. C'est dans cette optique que notre étude est inscrite. Pour terminer, nous avons proposé deux protocoles de consensus indulgents optimaux en nombres de tours et meilleurs connus en terme de nombre de messages échangés. Le premier, il agit en présence de défaillances, et il a une complexité de $O(t)$ en nombre de tours et $O(n.t)$ messages échangés. Le deuxième protocole n'est qu'un cas particulier du premier, il agit dans un environnement dépourvu de défaillances (détecteur de défaillances parfait), il a une complexité de $O(3)$ au plus en nombre de tours et $O(n)$ messages échangés.
Il reste désormais à trouver un protcole de consensus indulgent optimal en considérant les deux contraintes $early - deciding$ et $early - stopping$ avec les mêmes résultats de complexité que les nôtres.

Bibliographie

[1] Guerraoui, R., Raynal, M. : The information structure of indulgent consensus. IEEE Transactions on Computers, 53(4), 453-466, 2004).

[2] Gafni, E. : Round-by-round fault detectors (extended abstract) : Unifying synchrony and asynchrony. In : Proceedings of the 17th Symposium on Principles of Distributed Computing, 143-152, 1998.

[3] Guerraoui, R. : Indulgent algorithms (preliminary version). In : Proceedings of the 19th Symposium on Principles of Distributed Computing (PODC), 289-297, 2000.

[4] Lynch, N. : Distributed Algorithms. Morgan Kaufman publisher, San Francisco, California, 1996.

[5] Dutta, P., Guerraoui, R. : The inherent price of indulgence. Distributed Computing 18(1), 88-97, 2005.

[6] Sampaio, L.,Rasileiro, F. :Adaptive indulgent consensus. In Proceedings of the International Conference on Dependable Systems and Networks (DSN), 422-431, 2005.

[7] Ben-Or M. : Another Advantage of Free Choice : Completely Asynchronous Agreement Protocols. *Proc. 2nd ACM Symposium on Principles of Distributed Computing (PODC'83)*, ACM Press, pp. 27-30, 1983.

[8] P. Dutta, R. Guerraoui :Fast Indulgent Consensus with Zero Degradation, EDCC ?2, in : LNCS, vol. 24-85, 2002.

[9] Wu, W., Cao, J., Yang, J. and Raynal, M. : Using asynchrony and zero degradation to speed up indulgent consensus protocols. Journal of Parallel and Distributed Computing, 68(7), 984-996, 2008.

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

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

[12] 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.

[13] Mostefaoui, A. and Raynal, M. :Leader-Based Consensus Parallel Processing Letters, 11(1) :95-107. 2001.

[14] Sampaio, L., Brasileiro, F. :Adaptive indulgent consensus. In Proceedings of the International Conference on Dependable Systems and Networks (DSN05), 422-431, 2005.

[15] Pease L., Shostak R. and Lamport L., Reaching Agreement in Presence of Faults. *Journal of the ACM*, 27(2) :228-234, 1980.

[16] Rabin M., Randomized Byzantine Generals. *Proc. 24th IEEE Symposium on Foundations of Computer Science (FOCS'83)*, pp. 403-409, 1983.

[17] Gilbert, S., Guerraoui, R.,Kowalski, Dariusz R. : On the message complexity of indulgent consensus. A pele(Ed.) : DISC 2007, LNCS 4731, pp.283-297,2007.

[18] Dolev,D.,Lenzen,C. : Early-deciding consensus is Expensive.Proceeding of the ACM symposium on principles of distributed computing page 270-279, 2013.

[19] Fitzi,M.,Hirt, M. : Optimally Efficient Multi-valued Byzantine agreement. Proceedings of the twenty-fifth annual ACM symposium on principle of distributed computing, 163-168, 2006.

[20] Hurfin,M.,Raynal,M. : simple and fast synchronous consensus protocol based upon a weak failure detector. Distributed Computing, 12(4), 209-223, 1999.

[21] Alistrath,D.,Gilbert, S.,Guerraoui,R.,Travers,C. : generating fast indulgent algorithms. Proceeding ICDCN'11 Proceedings of the 12th international conference on Distributed computing and networking page 41-52, 2011.

[22] Aguilera, Marcos K.,Chen, W.,Toueg, S. : heartbeat a timeout-free failure detector for quiescent reliable communication. ACM communication, 1997.

[23] Chandra,T.,Hadzilacos,V.,Toueg,S. : The weakest failure detectors for solving consensus. ACM, 1994.

[24] Raynal,M. : failure detectors to solve asynchronous k-set agreement : a glimpse of recent results.Bulletin of the EATCS, European Association for Theoretical Computer Science, 103, 74-95, 2011.

[25] Freiling,F.,Guerraoui,R.,Kuznetsov,P. : the failure detector abstraction.ACM Computing Surveys,Vol.$43(2)$, Article 9, 2011.

[26] Alistarh,D.,Gilbert,S.,Guerraoui,R.,Travers,C. :of choices failures and asynchrony the many faces of set agreement. Proceeding ISAAC '09 Proceedings of the 20th International Symposium on Algorithms and Computation, 943-953, 2009.

[27] Chaudhuri,S. : more choices allow more faults set consensus problem in totally asynchronous system. ACM, 1992.

[28] Keidar,L.,Rajsbaum :On the cost of fault-tolerant consensus when there are no faults-a tutorial.ACM SIGACT News, 32(2), 45-63,2001.

[29] Hurfin,M.,Mostefaoui,A.,Raynal,M. : a versatile family of consensus protocoles based on Chandra-Toueg's unreliable failure detectors.IEEE transactions, 51(4), 395-408, 2002.

[30] Mostefaoui,A.,Raynal,M. : solving consensus using Chandra-Toueg's unreliable failure detectors : a general quorum based approach. Distributed computing, 1693, 49-63, 1999.

[31] Schiper,A. :early consensus in asynchronous system with a weak failure detector.Distributed computing, 10(4), 198, 1997.

[32] Mostefaoui,A.,Rajsbaum,S.,Raynal,M. :Conditions on input vectors for consensus solvability in asynchronous distributed systems. Journal of the ACM, 50(6), 922-954, 2003.

[33] Biely,M.,Robinson,P.,Schmid,U. : The generalized loneliness detector and weak system models for k-set agreement. IEEE transactions on parallel and distributed systems, 25(4), 1078-1088, 2014.

[34] Cristian,F.,Fetzer C. : The timed asynchronous distributed system model. IEEE transactions on parallel and distributed systems, 10(6), 642-657, 1999.

[35] Thomas, H., Charles, .L, Ronald, R., Clifford, S. : Introduction to algorithms.The MIT presse,Cambridge, Massachusetts London, England, 2009.

[36] Sedgewick, R., Flajolet, P. : An introduction to the analysis of algorithms, Addison-Wesley, Boston, 2013.

[37] Flajolet, P., Sedgewick, R. : Analytic combinatorics, Cambridge university press, 2009.

[38] Hromkovic, J. : Theoretical Computer Science : Introduction to Automata, Computability, Complexity, Algorithmics, Randomization, Communication, and Cryptography, Springer, Berlin, 2013.

[39] Gavoile, C. : Algorithmes distribuès, Laboratoire bordelais de recherche en informatique, Université de Bordeaux, 2014.

Résumé
Ce rapport de Master "recherche en informatique" option réseaux-systémes distribués présente les travaux et les derniers résultats obtenus dans le domaine des systèmes distribués asynchrones portant sur le problème de consensus. Il s'agit d'études visant à proposer un protocole indulgent optimal en nombre de tours et en nombre de messages échangés résolvant efficacement le problème de consensus.
Un algorithme indulgent est un algorithme distribué tolérant des périodes d'asychronisme lorsque les détecteurs de défaillances ne sont pas fiables.
Le travail effectué, introduit un nouveau protocole de consensus indulgent qui résout le consensus de façon optimale :optimal en nombre de tours et efficace en nombre de messages échangés. Afin d'atteindre l'efficacité en terme de messages échangés et en nombre de tours ,le protocole élaboré utilise $(t + 1)$ coordinateurs pour envoyer des messages dans chaque tour. Le protocole en question garantit qu'au moins un processus décide au bout de $(t + 2)$ tours avec une complexité en nombre de messages échangés de $O(n.t)$ et de $t + 3$ (au plus) $O(t)$ en nombre de tours, avec t est le nombre maximum de processus défaillants dans le système.
Cependant, la quête d'un algorithme indulgent optimal en tenant compte des contraintes "early-deciding" et "early stopping" est toujours d'actualité.
On conclut avec ce problème ouvert : peut-on trouver un algorithme de consensus indulgent optimal prennant en compte les deux contraintes citées avec $O(n.t)$ messages échangés et $O(t)$ tours ?

Abstract
This master thesis in computer science by research in the area of distributed systems and networking presents works and recent results obtained in the domain of asynchronous distributed systems based upon consensus problem. The study aims to propose an optimal indulgent consensus protocol with efficient communication complexity.
An indulgent algorithm is a distributed algorithm that tolerates asynchronous periods when failure detectors are unreliable.
In our work, we introduce a new indulgent consensus protocol that optimally solves the problem of consensus. In order to achieve efficiency in term of the number of exchanged messages and rounds needed, the developed protocol utilizes $(t + 1)$ coordinators to send messages in each round. The aforementioned protocol guaranties that at least one process decides at $(t + 2)$ round with a complexity in number of exchanged messages of $O(n.t)$ and $O(t)$ rounds at most $(t + 3).t$ is the maximum number of failures that a system can tolerates.
However, the quest for an optimal indulgent algorithm regarding "early-deciding" and "early-stopping" is still open. We conclude with the following open problem : Can we find an optimal indulgent consensus algorithm which takes into consideration the above mentioned constraints with $O(n.t)$ exchanged messages and $O(t)$ rounds ?