Un algorithme équitable d'exclusion mutuelle tolérant les fautes

Julien Sopena, Luciana Arantes, and Pierre Sens
Projet REGAL : LIP6 - Université de Paris 6 - CNRS - INRIA
4, Place Jussieu 75252 Paris Cedex 05, France.
Phone: (33).1.44.27.34.23 - Fax : (33).1.44.27.74.95
email: [julien.sopena,luciana.arantes,pierre.sens]@lip6.fr

Résumé

Cet article présente un nouvel algorithme d'exclusion mutuelle, tolérant les fautes et ayant un faible coût de recouvrement. Notre proposition est une extension de l'algorithme à base de jeton de Naimi-Tréhel qui utilise : Un arbre pour acheminer les requêtes et une file d'attente distribuée. En cas de faute(s), la file d'attente est reconstruite en reconnectant les portions intactes existantes. Cette approche présente deux avantages : un recouvrement efficace par rapport à une réinitialisation totale et la préservation de l'équité des requêtes en présence de fautes.

Nous comparerons les performances de notre algorithme à l'extension tolérante aux fautes proposée par Naimi-Tréhel [7] dans laquelle la file et l'arbre sont réinitialisés. Les expérimentations avec différents scenarii d'injection de fautes montrent clairement l'efficacité de notre approche. Nous mettons également en évidence l'importance du dimensionnement des temporisateurs sur les performances globales du système.

Mot clefs : Algorithme réparti, tolérance aux fautes, exclusion mutuelle, jeton, equité, performances

1 Introduction

L'exclusion mutuelle est une brique de base de l'Algorithmique Répartie, assurant qu'à un instant donné un seul processus peut exécuter une section critique (SC). Dans cet article nous nous intéressons plus particulièrement aux algorithmes utilisant un jeton unique. Dans ces derniers, la possession du jeton donne le droit d'entrer en section critique. Parmi les algorithmes à jeton les plus connus, figurent ceux de Suzuki-Kazami [12], Raymond [10], Naimi-Tréhel [6], Neilsen-Mizuno [8] et Chang, Singhal et Liu [2].

Les algorithmes à base de jeton présent l'avantage d'être relativement peu coûteux en termes de messages. Parmi ceux-ci ceux utilisant un arbre pour faire circuler les requêtes, ont une complexité en O(Log(N)) ; N étant le nombre de nœuds du système. En contrepartie, ils sont particulièrement sensibles aux défaillances.

Dans cet article nous proposons un nouvel algorithme d'exclusion mutuelle tolérant les fautes qui est une extension de l'algorithme de Naimi-Tréhel basé sur un arbre dynamique. Nous avons choisi d'étendre cet algorithme car il présente de très bonnes performances en absence de fautes. Par rapport aux autres approches, notre algorithme vise, d'une part, à minimiser les nombres de diffusions lors des recouvrements et, d'autre part, à conserver l'équité des requêtes en présence de fautes.

Nous avons déjà présenté les principes de base de cet algorithme dans [11]. Ici nous présentons les premières évaluations de ses performances ainsi qu'une comparaison avec la version tolérante aux fautes proposée par Naimi-Tréhel [7].

Cet article est organisé de la façon suivante. La section 2 introduit notre modèle d'environnement. La section 3 présente brièvement les travaux existants. Dans la section 4 nous détaillons l'algorithme de Naimi-Tréhel. La section 5 décrit notre extension et la section 6 présente une étude comparative des performances des deux algorithmes.

2 Modèle

Nous considérons un système distribué composé de N sites $\Pi = \{S_1, S_2, \ldots, S_N\}$ reliés par un réseau complètement maillé. Ces sites communiquent uniquement par envoi de messages sur des canaux de communication supposés fiables, mais dont les messages peuvent ne pas être délivrés dans l'ordre où ils ont été émis.

Le système est supposé synchrone : les délais d'acheminement des messages et la vitesse relative entre les différents sites sont bornés par des valeurs connues. En revanche, on ne fera aucune hypothèse sur la durée des sections critiques.

Les noeuds sont supposés silencieux sur défaillance (*fail-silent*) : un nœud fautif s'arrête proprement en suspendant ses transmissions et réceptions de messages.

Notre modèle tolère $N-1$ fautes. Dans la suite, les termes nœuds et sites sont interchangeables.

3 Travaux existants

Plusieurs auteurs ont défini des extensions tolérantes aux fautes des algorithmes à jeton. Nishio et al. [9] proposent une extension fiable de l'algorithme à diffusion de Suzuki-Kasami[12]. Pour régénérer le jeton, leur algorithme nécessite un acquittement de *tous* les sites. Ainsi, la défaillance d'un seul nœud retarde la régénération du jeton jusqu'à son retour (les nœuds sont supposés avoir une mémoire stable). Afin d'avoir un recouvrement plus rapide Manivannan et Singhal présentent dans [4] un nouvel algorithme dans lequel seuls les nœuds non fautifs doivent répondre. Cependant, leur approche n'est pas équitable devant les fautes car ils supposent que deux sites particuliers ne peuvent pas être en panne : le dernier nœud x ayant exécuté la section critique et celui à qui x a transmis la jeton.

Chang et al. présentent dans [1] un algorithme tolérant les fautes basé sur un arbre de requête. La fiabilisation de l'arbre est assurée par l'introduction de chemins redondants et par l'utilisation d'un mécanisme d'élection. Cependant, l'ajout des chemins alternatifs et surtout le mécanisme de prévention de cycle augmentent significativement le coût de l'algorithme.

Dans [7] Naimi et Tréhel introduisent une version fiable de leur algorithme. En absence de faute, l'algorithme initial n'est pas modifié. En revanche, le recouvrement des fautes est particulièrement coûteux en termes de messages et de latence. Cet algorithme est détaillé dans la section suivante.

Plus récemment, Mueller [5] a proposé aussi une version fiable de l'algorithme de Naimi-Tréhel sans utiliser de diffusions. Sa solution repose sur un anneau reliant tous les nœuds. Les inconvénients d'une telle approche sont sa faible extensibilité et, plus encore, son impossibilité à gérer plusieurs fautes simultanées.

4 L'algorithme de Naimi-Tréhel et son extension tolérante aux fautes

L'algorithme à jeton de Naimi-Tréhel[6] maintient deux structures logiques : un arbre dynamique appelé "arbre des *last*", qui sert à acheminer les demandes de section critique (SC) et une file des sites qui attendent d'entrer en section critique appelée "file des *next*". Chaque site mémorise

uniquement un nœud *last* auquel il enverra ses demandes de SC et, le cas échéant, un nœud *next* auquel il retransmettra son jeton une fois servi.

Initialement, le propriétaire du jeton est la racine de l'arbre des *last* : il est le *last* de tous les autres sites. Lorsqu'un site demande le jeton, sa requête est transmise de sites en sites, suivant l'arbre des *last* jusqu'à en atteindre la racine. A chaque réception de requête, les sites repositionnent leur *last* pour désigner le site demandeur (l'arbre est modifié dynamiquement). Lorsque la requête arrive à la racine, cette dernière positionne sa valeur *next* pour pointer vers le site demandeur. Lorsque la racine aura fini sa section critique, elle transmettra le jeton à son *next*.

La figure 1 illustre l'algorithme. Initialement (a), le site A possède le jeton et les variables *last* de tous les autres sites pointent vers A. Puis (b), le site B demande le jeton en envoyant une requête à son *last*. A change son *next* et son *last* pour pointer vers B. Ainsi B devient la nouvelle racine. Enfin (c), C demande le jeton à A. Sa requête est rédigée de A vers B qui mettent à jour leur *last* pour désigner C. Finalement B prend C comme *next*.

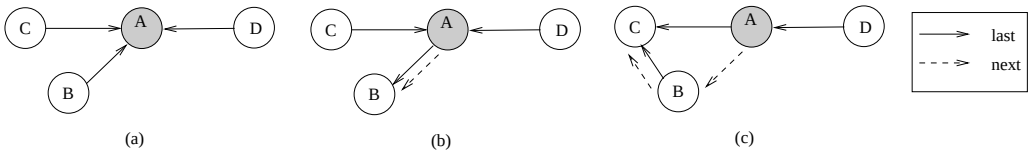

FIG. 1 – Exemple de l'algorithme de Naimi-Tréhel

Dans [7], Naimi et Tréhel proposent une version tolérante aux fautes de leur algorithme. Des extensions sont ajoutées pour détecter les défaillances et régénérer le jeton. En absence de faute, l'algorithme initial n'est pas modifié. Un nœud demandeur de SC, S_i, suspecte une défaillance si il n'a pas reçu le jeton après un certain timeout. Dans ce cas il diffuse un message *CONSULT* pour vérifier l'état de son prédécesseur dans la file des *next*. Un nœud ne répond au message *CONSULT* que si son *next* désigne S_i. Si, après un second timeout, S_i n'a reçu aucune réponse, il en déduit soit que son prédécesseur est défaillant, soit que sa demande de SC a été perdue. S_i diffuse alors un message *FAILURE* pour détecter la présence du jeton. Si S_i reçoit une réponse du possesseur du jeton, il lance une procédure de **recouvrement individuel** en renvoyant sa requête au propriétaire du jeton. Sinon le jeton est perdu. Dans ce cas une **réinitialisation globale** est effectuée : S_i lance une élection pour régénérer le jeton. Alors l'arbre des *last* est réinitialisé et la file des *next* est supprimée. Tous les sites en attente de SC devront ensuite réémettre leur requête.

On peut remarquer que la faute d'un site ne possédant pas le jeton peut entraîner de nombreux recouvrements individuels concurrents. Nous verrons dans l'étude des performances que ce cas entraîne un surcoût très important (souvent plus coûteux qu'une réinitialisation globale).

5 Algorithme équitable tolérant aux fautes

Cette section présente notre algorithme ainsi que ses propriétés.

5.1 Description de l'algorithme

L'idée de base est de reconstruire la file des requêtes (la file des *next*) partitionnées par les fautes, en réassemblant dans l'ordre les morceaux encore intacts. C'est uniquement quand cela est impossible, que des requêtes de SC sont réémises et que l'arbre des *last* est reconstruit.

Dans l'algorithme de Naimi-Tréhel original, un site ne connaît pas ses prédécesseurs dans la file des *next*. Ce qui nécessite de multiples diffusions en cas de fautes. Aussi pour informer chaque site, sur ses prédécesseurs nous ajouterons à l'algorithme initial un mécanisme d'acquittement des requêtes.

Chaque site S_i maintient les variables suivantes :

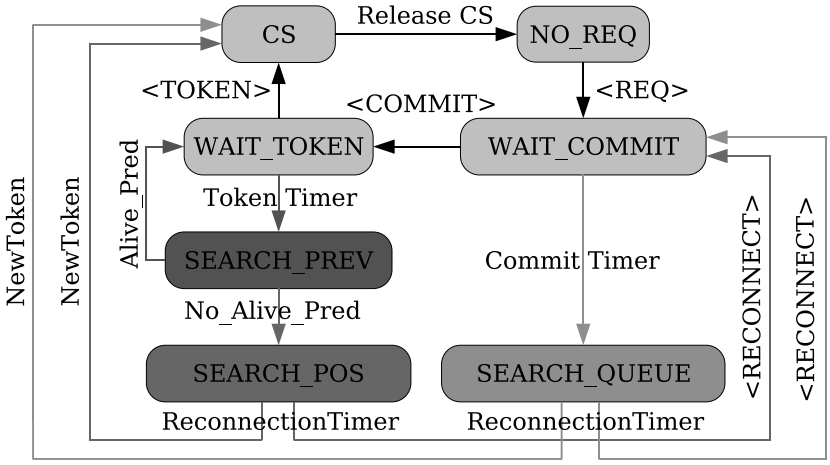

FIG. 2 – Automate de l'algorithme

- *last* : le dernier nœud qui a démandé à S_i l'obtention du jeton.
- *next* : le prochain nœud qui recevra le jeton lorsque S_i aura terminé sa section critique.
- $pred[0..k-1]$: les k predecesseurs de S_i dans la file des *next*, k étant un paramètre de l'algorithme.
- *pos* : la position courante de S_i dans la file des *next* (le nœud possédant le jeton ayant la position la plus petite).

La figure 2 décrit l'automate de notre algorithme. Lorsqu'un site S_i, initialement à l'état NO_REQ, demande une section critique : il envoie une requête à son *last*, arme le temporisateur *CommitTimer* et passe à l'état $WAIT\_COMMIT$. La requête est ensuite transmise via l'arbre des *last* jusqu'à la racine. La racine renvoie alors à S_i un message *COMMIT* contenant les informantions suivantes :

- les k prédécesseurs de S_i (S_j et ses $k-1$ prédécesseurs)
- la position de S_i dans la file.

A la réception du message $COMMIT$, S_i met à jour sa position dans la file et la liste de ses prédécesseurs. Puis, il arme le temporisateur $TokenTimer$ et passe à l'état $WAIT\_TOKEN$. Si après l'expiration du temporisateur, le jeton n'est pas reçu, S_i passe à l'état $CHECK\_PREV$ dans lequel il vérifie la présence de son prédécesseur le plus proche.

Si le prédécesseur ne répond pas, S_i recherche le premier prédécesseur, S_x, valide parmi les k et lui envoie un message de reconnexion. S_x positionne alors son *next* à S_i et lui renvoie sa liste de prédécesseurs. S_i retourne ensuite à l'état $WAIT\_TOKEN$ pour attendre le jeton.

Si S_i ne trouve aucun prédécesseur valide, il passe à l'état $SEARCH\_POS$ (le file est brisée). Il diffuse alors un message $SEARCH\_POS$ contenant sa position dans la file des *next* et la liste de ses prédécesseurs fautifs, puis arme le temporisateur $ReconnectionTimer$. A la réception de $SEARCH\_POS$, tous les sites ayant une position strictement inférieure à S_i lui répondent. Les sites dont les *last* pointent vers les sites fautifs font également pointer leur *last* vers S_i. A l'expiration de $ReconnectionTimer$, S_j choisit parmi les réponses reçues le site, S_{max}, ayant la position la plus élevée, lui envoie une demande de connexion et repasse à l'état $WAIT\_COMMIT$. A la réception de cette demande, S_{max} positionne son *next* à S_i. Si à l'expiration de $ReconnectionTimer$, S_i n'a reçu aucune réponse, il en conclut que tous ses prédécesseurs sont fautifs et régénère le jeton en passant directement à l'état CS.

Nous considérerons maintenant le cas où S_i n'a pas reçu de message *COMMIT* en réponse à sa requête. Pour simplifier, nous supposons, à ce stade, que S_i est le seul site qui traite la faute (le problème des recouvrements concurrents sera abordé plus loin). A l'expiration du temporisateur *CommitTimer*, S_i ne connaît donc ni sa position, ni ses prédécesseurs dans la file des *next*. Le site

passe alors à l'état $SEARCH\_QUEUE$ où il diffuse un message afin de se reconnecter au site ayant le plus grande position dans le file (i.e., en queue de file des *next*). Tous sites ayant une position dans la file répondent à S_i en lui envoyant leur position. Comme précédemment, S_i choisit alors le site S_{max} ayant la plus grande position, lui envoie une demande de reconnexion et retourne à l'état $WAIT\_COMMIT$. Si S_i n'a reçu aucune réponse à l'expiration de $ReconnectionTimer$, il régénère le jeton et passe à l'état *CS*.

Contrairement au recouvrement à partir de l'état $SEARCH\_POS$, l'ordre des requêtes dans la file n'est plus préservé et il faut reconstruire l'arbre de requêtes pour rester cohérent avec la file. Cette reconstruction est faite dynamiquement à la réception du message $SEARCH\_QUEUE$ de la façon suivante :

- Les sites ayant une position dans la file ou n'étant pas demandeurs de SC, positionnent leur *last* à S_i en considérant que S_i est le dernier à demander la SC.
- Les sites attendant le jeton sans avoir encore de position, mettent leur *last* à la même valeur que leur *next*. Ils sont sûrs que le *next* a demandé le jeton après eux, ce qui permet d'éviter des cycles dans l'arbre des *last*.

Enfin, lorsque qu'une faute est détectée par plusieurs sites, il faut élire un site pour mener à bien la procédure de recouvrement. Nous utilisons alors les horloges de Lamport [3]. Ces horloges estampillent tous les messages $SEARCH\_QUEUE$ avec une couple (compteur, identifiant de site). Lorsqu'un site en phase de recouvrement reçoit un message $SEARCH\_QUEUE$ avec une estampille inférieure à son propre message, il abandonne son recouvrement et envoie directement à l'émetteur du message une requête de SC.

5.2 Exemple

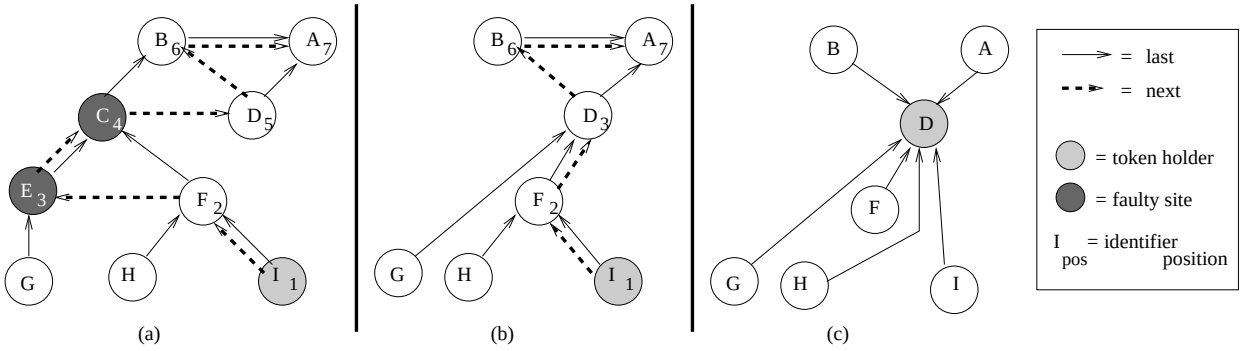

FIG. 3 – Exemple de recouvrement de fautes : (a) configuration intiale, (b) notre algorithme, (c) algorithme de Naimi-Tréhel

La figure 3 montre un exemple de recouvrement des deux algorithmes. Nous considérons initialement que les sites C et E sont défaillants (3.a). La file des *next* est alors brisée en deux morceaux I,F et D,B,A. On suppose que tous les sites présents dans la file ont déjà obtenu une position et attendent le jeton (ils sont donc dans l'état $WAIT\_TOKEN$) sauf I qui est le propriétaire courant du jeton. Chaque site connaît uniquement deux de ses prédécesseurs ($k = 2$).

A la prochaine vérification de la présence de son prédécesseur, D détecte les fautes de C et E. Il entre alors dans l'état $SEARCH\_POS$ en diffusant un message $SEARCH\_POS$ incluant sa position dans la file (5) et les identifiants des sites fautifs (E et F). Seuls F et I dont la position est inférieure à 5 répondent à D. Les sites G et F dont le *last* pointe vers l'un des sites fautifs positionnent directement leurs *last* à D. A l'expiration de $ReconnectionTimer$, D choisit F comme prédécesseur, ce dernier étant parmi les émetteurs de réponses le site ayant la plus grande position. D envoie donc un message de reconnexion à F. F positionne alors son *next* à D et lui envoie

l'identifiant de ses deux nouveaux prédécesseurs (F et I). Le figure 3.b montre la configuration finale obtenue.

Pour illustrer, l'algorithme tolérant aux fautes de Naimi-Tréhel, nous considérerons la même configuration initiale (3.a). Nous supposons que D détecte la faute lorsque le jeton est perdu suite à son envoi au site E. N'ayant aucune réponse à sa diffusion du message *CONSULT* (pour trouver son prédécesseur) et *FAILURE* (pour détecter le jeton), D en conclut que le jeton est perdu. Il lance alors une élection et régénère un nouveau jeton. Tous les sites font alors pointer leur *last* vers D et positionnent leur *next* à $VIDE$. Les sites n'ayant pas encore obtenu le jeton (A et B) devront renouveler leur requête de SC. La figure 3.c montre la configuration finale.

5.3 Dimensionnement des temporisateurs

Le réglage des temporisateurs est essentiel. Il s'agit de trouver les valeurs qui permettent d'avoir un algorithme correct vis-à-vis des hypothèses de synchronisme tout en obtenant une faible latence de recouvrement. Des valeurs petites permettent de réagir rapidement aux fautes mais induisent de fausses détections. De grandes valeurs permettent une détection parfaite mais le recouvrement peut être très inefficace.

Notre algorithme utilise trois temporisateurs :

- *CommitTimer* : armé par un site lors d'une demande de SC. Il expire si aucun message *COMMIT* n'a été reçu.
- *TokenTimer* : armé lorsqu'un message *COMMIT* est reçu. Il expire, si le jeton n'est pas reçu. Ce temporisateur est aussi utilisé pour vérifier périodiquement la présence de son plus proche prédécesseur.
- *ReconnectionTimer* : utilisé lors des phases de reconstruction de la file.

Tout d'abord, nous constatons que notre algorithme (comme l'extension proposée par Naimi-Tréhel) supporte les fausses détections de fautes. En effet, si *CommitTimer* ou *TokenTimer* sont trop petits, la procédure de recouvrement de fautes peut démarrer alors qu'il n'y a aucun site fautif. Cette erreur est corrigée par la suite lorsque que l'on vérifie la présence des prédécesseurs ou lorsque l'on collecte les informations sur les positions. Dans ces deux cas, *ReconnectionTimer* est utilisé. Ce temporisateur est dimensionné pour éviter toutes fausses suspicions. En effet, nous fixons sa valeur à $2 * Tmsg$, $Tmsg$ étant le temps maximum de transfert d'un message (étant en système synchrone, cette valeur est supposée connue).

Pour éviter les fausses suspicions lorsqu'un nœud attend le message *COMMIT*, nous pouvons borner *CommitTimer* à la valeur $((N-1)+1) * Tmsg$ (au pire, la demande de jeton est transmise $N-1$ sites). En revanche, nous ne pouvons pas borner *TokenTimer* qui contrôle la réception du jeton car ce temps dépend de la durée maximum passée en section critique. Or nous ne faisons aucune hypothèse applicative. Ainsi, l'expiration de *TokenTimer* peut entraîner des fausses détections qui seront ensuite corrigées.

5.4 Propriétés de l'algorithme

Notre algorithme présent les propriétés suivantes :

Faible surcoût en absence de faute : Le coût pour véhiculer l'information sur les prédécesseurs se limite a un seul message. La complexité globale reste donc en $\mathcal{O}(log(N))$.

Recouvrement efficace : En minimisant le nombre d'étapes lors d'un recouvrement, en utilisant un mécanisme global pour reconstruire l'arbre et limitant au minimum le nombre des réémissions de requête, notre algorithme restreint le coût en messages d'un recouvrement de faute(s).

Isolation des fautes : En testant périodiquement la vivacité du prédécesseur direct, notre algorithme peut détecter et réparer une faute avant que le jeton ne se perde. La latence du recouvrement est alors complètement recouvert par le temps d'attente du jeton.

Equitable : La réparation de la file, plutôt que sa réinitialisation, permet de minimiser le nombre de réémissions des requêtes, maintenant ainsi l'équité pour l'accès à la section critique.

6 Evaluation des performances

Dans cette section, nous présentons une étude de performance de notre algorithme et de l'algorithme de Naimi-Trehel tolérant aux fautes.

6.1 Environnement et paramètres

Nous avons utilisé une grappe de 20 nœuds. Chaque nœud possède deux processeurs Xeon à 2.8Ghz et 2Go de Ram et fonctionnent sur un noyau Linux 2.6. Ils sont reliés par un réseau ethernet de 1 Gbit/s. En activant l'hyperthreading sur chacun des 40 processeurs nous émulons un système composé de 80 nœuds logiques ($N = 80$).

Les algorithmes ont été implémentés en utilisant les sockets UDP. Pendant chaque expérience, tous les nœuds accèderont 5 fois à la section critique. En plus des scenarii sans fautes, on injectera simultanément : 1, 3, 5, 8, 20 ou 40 fautes. Les résultats présentés ici correspondent à une moyenne sur 20 expériences.

Une application répartie utilisant un mutex, se caractérise par :

- α : le temps moyen d'exécution d'une section critique
- β : le temps entre la fin d'une section critique et l'émission d'une nouvelle requête sur le même nœud.
- ρ : le ratio α/β

Nous avons considéré trois types d'applications qui diffèrent par leur degré de parallélisme (resp. petit, moyen et grand). Elles correspondent a des valeurs différentes de ρ, respectivement :

- $\rho = 1$: une application à faible degré de parallélisme dans laquelle presque tous les sites attendent la section critique. La file des *next* est alors longue.
- $\rho = N$: une application à parallélisme moyen dans laquelle quelques sites attendent la section critique. La file des *next* est petite.
- $\rho = 2 * N$: une application hautement parallèle dans laquelle il n'y a pas de concurrence pour l'accès à la section critique. La file des *next* est alors vide ou réduite à un seul site.

Comme on l'a vu à la section 5.3, le choix des temporisateurs est essentiel. Nous avons alors défini trois valeur de temporisateurs en fonction du temps moyen (*pingAvg*) et du temps maximum (*pingMax*) d'un aller pour un "ping" dans notre grappe. :

- Temporisateur passif : $PassT = N * pingMax = 11.85s$
- Temporisateur intermédiaire : $InterT = log2(N) * pingMax = 3.95s$
- Temporisateur agressif : $AggrT = log2(N) * pingAvg = 0.32s$

Pour chaque expérience on utilisera la même valeur pour les temporisateurs *CommitTimer* et *TokenTimer*, ainsi que pour le temporisateur (*TimerNT*) utilisé pour la détection dans l'algorithme de Naimi et Tréhel. Le timer de reconnexion *ReconnectionTimer* sera quant à lui fixé à $1s$.

Dans chaque étude nous considérerons trois métriques : **le nombre de messages envoyés**, **le nombre total de messages reçus** et **le temps moyen d'attente du jeton**.

La différence entre le nombre de messages envoyés et reçus caractérise l'utilisation de diffusion. Le temps d'attente du jeton, en absence de faute, est quand à lui directement lié à la taille de la file et donc à la valeur du paramètre ρ.

Remarque : Dans chacun des graphiques présentés ci-après, l'abscisse représente le nombre de fautes. Pour bien analyser les courbes, il faudra tenir compte de la diminution du nombre de

sites après les fautes. En dehors du coût de la détection et de celui du recouvrement, le nombre de messages envoyés et reçus ainsi que le temps d'attente du jeton diminuent automatiquement avec l'augmentation du nombre de fautes puisqu'il y a moins de sites dans le système.

6.2 Impact du type d'application

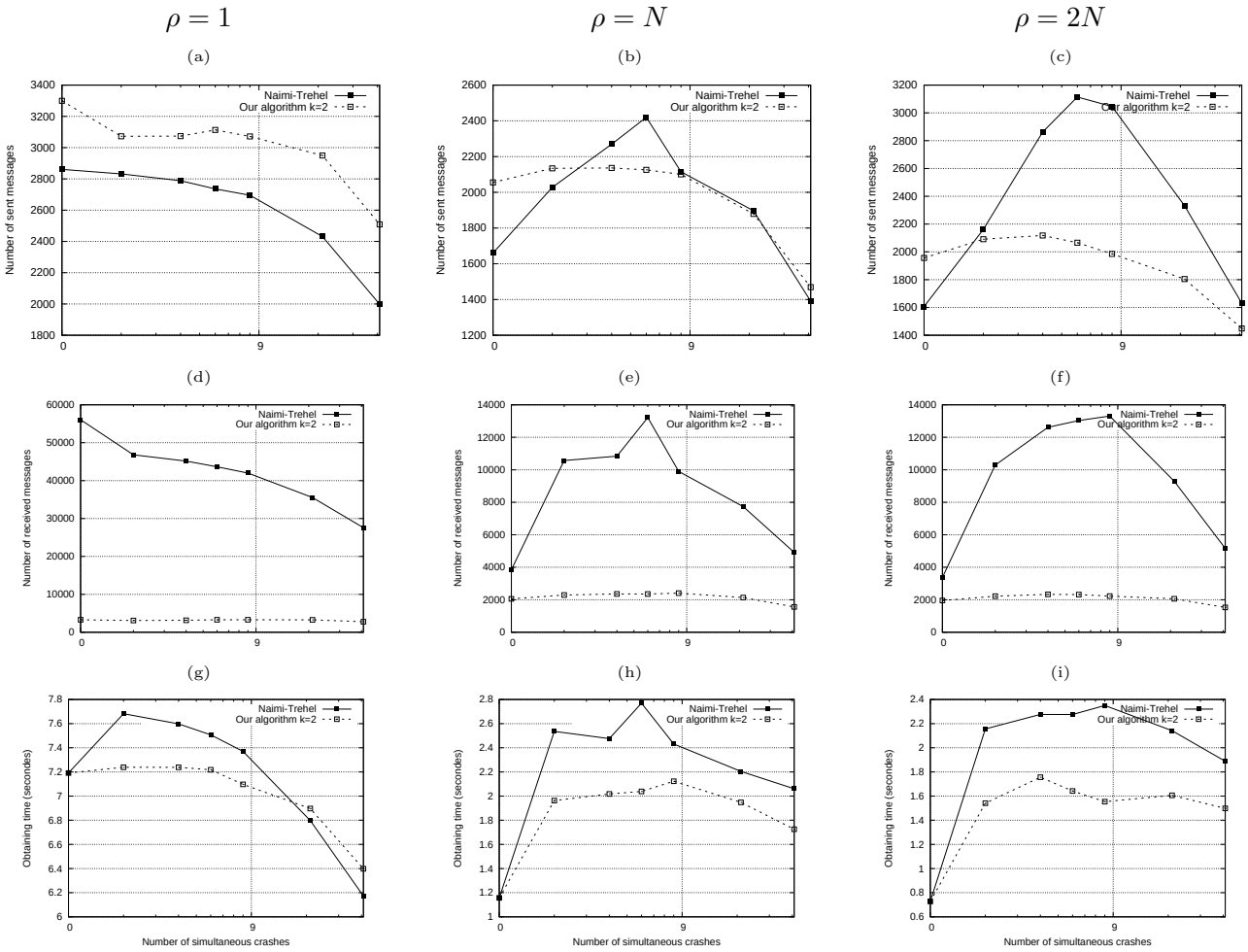

FIG. 4 – Impact du type d'application

Pour étudier l'influence du type d'application sur les algorithmes, on fixe les temporisateurs *CommitTimer*, *TokenTimer* et *TimerNT* à une valeur intermédiaire *InterT* (3.95 s). Le nombre k de prédécesseurs connus est, quant à lui, fixé à 2.

Nous remarquons qu'en absence de faute, le temps d'obtention du jeton est très dépendent du degré de parallélisme : 7.2s (figure 4.f), 1.2s (figure 4.g) et 0.7s (figure 4.h).

Nombre de messages envoyés

Plus le ratio est petit, plus le temps d'obtention est long et donc plus le nombre de fausses suspicions est élevé. Ceci explique pourquoi le nombre de messages envoyés diminue quand ρ augmente (figures 4.a, 4.b et 4.c).

Sur les mêmes figures, on peut aussi observer que notre algorithme présente un surcoût en nombre de messages envoyés dû au message *COMMIT* en comparaison avec l'algorithme de Naimi-Tréhel.

Quand $\rho = 1$ avec un petit nombre de faute, la différence entre les deux algorithmes diminue. En effet, dans notre solution, la connaissance des prédécesseurs permet une réparation locale de la

longue file des *next* ($\rho = 1$), là où celle de Naimi-Tréhel nécessitait une réinitialisation totale des structures. Cette stratégie atteint sa limite lorsque 50% de nœuds sont en faute. Il devient alors moins coûteux en terme de messages, de réinitialiser tout le système aux dépends de l'équité.

Quand ρ augmente, le nombre de sites qui ne sont pas en attente du jeton augmente aussi. Ces sites sont amenés à envoyer des requêtes le long de l'arbre des *last*, endommagé par les fautes. Sur les graphes 4.b et 4.c, on observe un comportement régulier de notre algorithme quand le nombre des fautes augmentent. En effet, la perte d'une seule de ces requêtes lance une reconstruction de l'arbre des *last* autour du site élu (*SEARCH_QUEUE* mécanisme, voir section 5.1). A l'inverse, dans Naimi-Tréhel, chaque perte de requête lance un recouvrement individuel (voir section 4).

Plus le nombre de fautes augmente, plus l'arbre est endommagé et plus le nombre de *recouvrement individuel* augmente. Mais à partir d'un certain nombre de fautes, le jeton finit par se perdre. Le mécanisme de réinitialisation global prend alors la place des recouvrements individuels. Ceci explique les pics dans l'algorithme de Naimi-Tréhel. Ce comportement s'intensifie lorsque ρ augmente, car la taille de la file diminue (voir le dernier paragraphe de la section 4).

Nombre de messages reçus

L'importante différence, en nombre de messages reçus, dénote une utilisation réduite de diffusions dans notre algorithme en comparaison de celui de Naimi-Tréhel.

On peut analyser le coût des mécanismes de détection en s'intéressant plus spécifiquement aux scenarii "sans faute". Pour Naimi-Tréhel, quand $\rho = 2N$, le nombre de messages reçus est de 3.700 messages mais pour $\rho = 1$, ce chiffre atteint les 56.000 messages. A l'inverse notre algorithme reste peu sensible aux changements de comportement. Lorsque ρ diminue le *temps d'attente du jeton* augmente et avec lui le nombre de fausses suspicions. Or, à chaque suspicion d'une faute, l'algorithme de Naimi-Tréhel procède à une diffusion du message *CONSULT* tandis que notre algorithme utilise un simple *ping*.

On peut donc conclure que le *temps d'attente du jeton* et, implicitement avec lui, la durée en section critique ont une grande influence sur l'algorithme de Naimi-Tréhel contrairement au nôtre.

Temps moyen d'attente du jeton

4.h, and 4.i que notre algorithme présente un *temps moyen d'attente du jeton* plus faible pour presque toutes les expériences. Cette différence s'explique par un mécanisme de recouvrement plus rapide que ceux de Naimi-Tréhel. La seule exception est lorsqu'on injecte 50% de fautes dans une application faiblement parallèle ($\rho = 1$). Comme on l'a dit pour le nombre de messages envoyés, il est alors plus facile de réinitialiser une longue file très endommagée que d'essayer de la réparer.

Plus précisément, on peut aussi remarquer un comportement intéressant quand $\rho = 1$: Lorsque le nombre de fautes varie de 0 à 5, le *temps d'attente du jeton* pour notre algorithme n'augmente pas (Figure 4.g). Ceci illustre bien le phénomène d'*isolation des fautes* présenté à la section 5.4. En effet comme la file des *next* est grande, le recouvrement des fautes est entièrement couvert par le temps d'attente. Aussi lorsque le jeton sera envoyé au site en faute la file sera déjà réparée.

6.3 Impact des temporisateurs

Pour mesurer l'impact des temporisateurs de suspicion nous considérons l'application à parallélisme moyen ($\rho = N$) en conservant $k = 2$. Les temporisateurs *CommitTimer*, *TokenTimer* et *TimerNT* prendront ensemble les valeurs $PassT$, $InterT$ et $AggreT$ décrites dans la section 6.1.

Nombre de messages envoyés

En comparant le nombre de messages envoyés en absence de faute sur les Figures 5.a, 5.b et 5.c on retrouve le surcoût dû aux messages *COMMIT*. Ce surcoût devient proportionnellement moins important quand le temporisateur de suspicion diminue. En effet lorsque le temporisateur diminue le nombre de fausses suspicions augmente. Les messages dus à ces fausses suspicions deviennent alors beaucoup plus important que ceux dus au mécanisme d'acquittement.

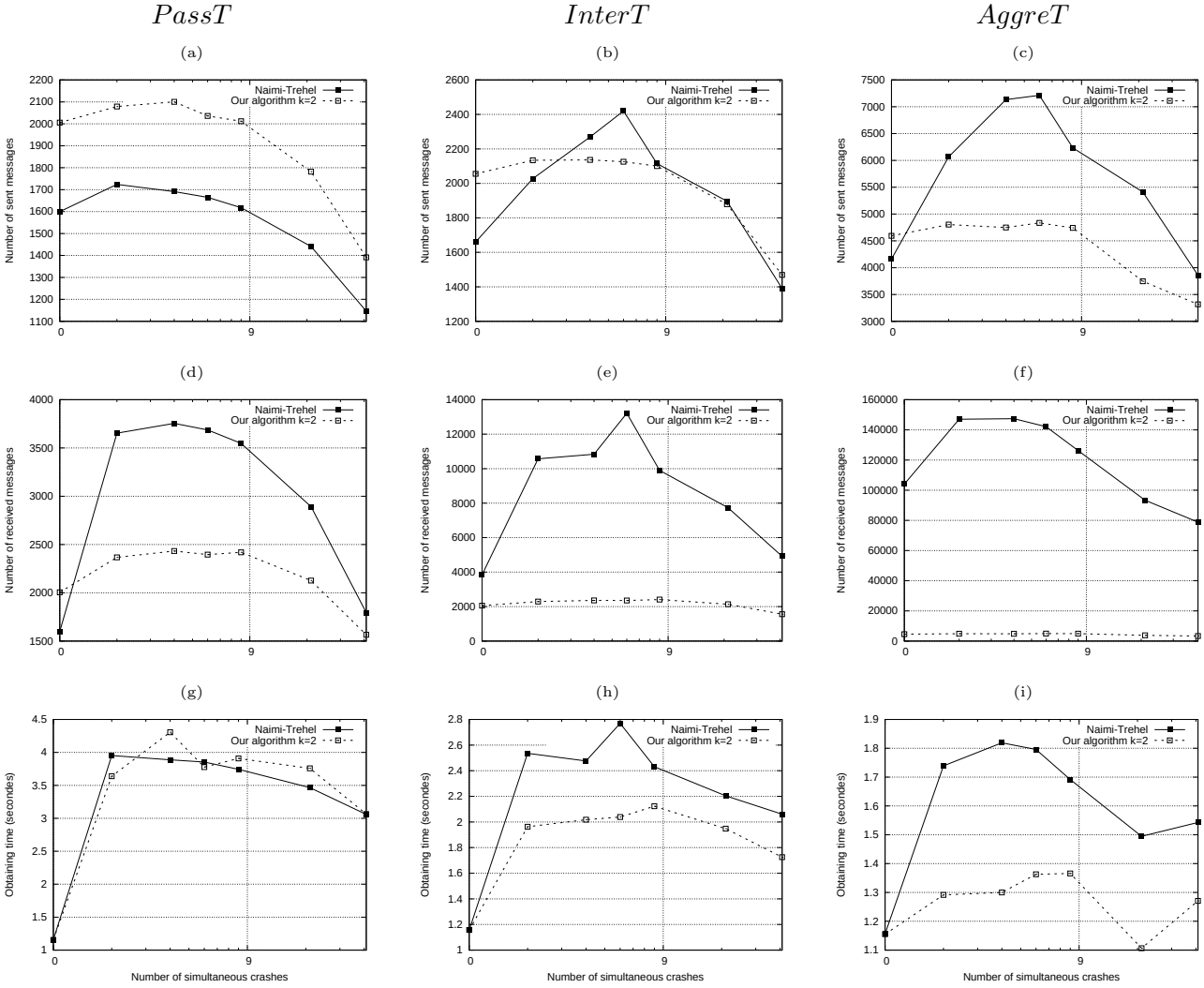

FIG. 5 – Impact des temporisateurs

Le dimensionnement du temporisateur agit aussi sur le mécanisme de recouvrement. Ainsi, avec un temporisateur relativement large (Figure 5.a), la majorité des sites sont en attente du jeton lorsque les fautes sont enfin détectées. Les comportements des deux algorithmes sont alors relativement semblables : dans les deux cas il y a une élection pour reconstruire l'arbre. A l'inverse, pour des temporisateurs plus agressifs (Figures 5.b 5.c), le coût du recouvrement en nombre de messages pour l'algorithme de Naimi-Tréhel augmente énormément. En effet pour ce dernier, une diminution du temporisateur s'accompagne d'une augmentation de l'importance des *recouvrement individuel* (voir section 4). En comparaison le nombre de messages envoyés reste stable avec notre algorithme.

Nombre de messages reçus

L'ensemble des courbes (Figures 5.d, 5.e, et 5.f), met bien en évidence l'utilisation de diffusions dans l'algorithme de Naimi-Tréhel, pour le recouvrement de vraie(s) faute(s) comme pour les mécanismes de détection (scenarii sans fautes).

Ceci représente une vraie limite pour le choix du temporisateur chez Naimi-Tréhel. Avec le temporisateur *AggrT* on dépasse les 100.000 messages. En comparaison notre algorithme reste aux environs de 2.000 messages reçus.

Temps moyen d'attente du jeton

Nous venons d'observer le coût, en nombre de messages émis et reçus, résultant de l'utilisation d'un temporisateur agressif. Mais y a-t-il un intérêt à choisir ce type de temporisateur ?

Pour répondre à cette question nous comparerons le *temps moyen d'attente du jeton* en cas de faute pour les trois temporisateurs (Figures5.g, 5.h et 5.i). Pour un temporisateur $PassT$, l'ordre de grandeur est de $4s$ pour les deux algorithmes. Si on prend un temporisateur $InterT$, la latence se réduit à $2.1s$ pour notre algorithme contre $2.4s$ pour l'algorithme de Naimi-Tréhel. Enfin, en choisissant $AggrT$, on obtient un temps d'attente de $1.3s$ pour notre solution contre $1.8s$ pour celle de Naimi-Tréhel.

Nous observons donc un réel gain de latence lorsqu'on diminue la durée du temporisateur. Ce gain se révelle par ailleurs légérement plus grand pour notre algorithme.

6.4 Impact du nombre de prédécesseurs : k

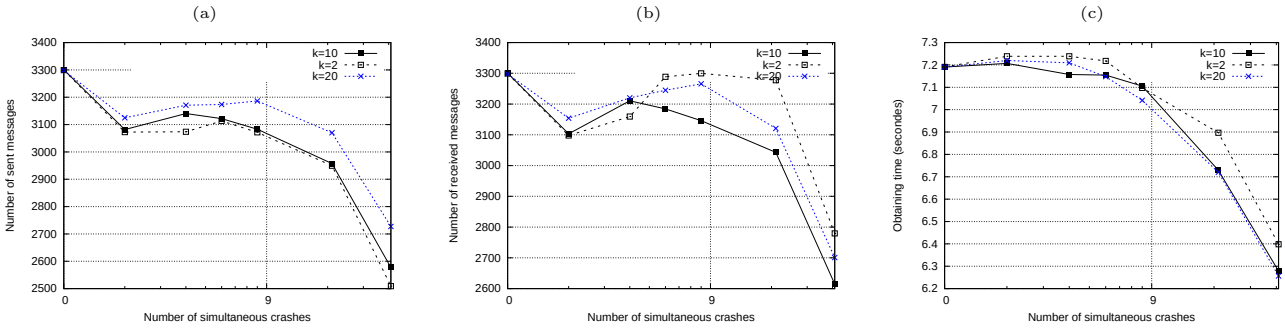

FIG. 6 – Predecessors number influence

Notre algorithme est basé sur la connaissance de K-prédécesseurs. Comme nous pouvons l'observer sur la Figure 6, en absence de faute la valeur de ce paramètre n'a pas d'influence sur l'algorithme. En effet, les informations relatives aux prédécésseurs sont véhiculées sur le même message $COMMIT$ et le nombre de prédécesseurs n'influe pas sur les mécanismes de détection : "attente du commit" ou "ping du premier prédécesseur".

Pour étudier l'impact de k sur les mécanismes de recouvrement, nous avons choisi une application où la file des *next* est longue ($\rho = 1$). En effet avec un ρ plus grand l'impact devient moins visible. Il est même inexistant avec $\rho = 2N$, puisqu'il n'y a plus de file des *next*. Pour ces expériences les temporisateurs *TokenTimer* et *CommitTimer* sont fixés à la valeur intermédiaire $InterT$.

Lorsqu'il y a quelques fautes, nous observons que le nombre de messages envoyés augmente proportionnellement à la valeur de k. En effet, le coût de la recherche du premier prédécesseur non fautif augmente avec k.

Pour un nombre de fautes plus grand, la probabilité d'avoir au moins 2 sites consécutifs en faute devient plus forte. En analysant la Figure 6.b, on remarque que l'utilisation d'un grand nombre de prédécesseurs ($k = 10$ ou $k = 20$) limite l'utilisation de diffusion du message $SEARCH\_POS$.

Enfin, en présence de nombreuses fautes, on peut relever un léger gain en latence (Figure 6.c) pour des valeurs élevées de k ($k = 10$ ou $k = 20$).

7 Conclusions

La plupart des algorithmes existant ne préserve pas l'ordre de la file des requêtes après un recouvrement de faute(s). Ainsi, tous les noeuds attendant d'entrer en section critique sont dans l'obligation de réémettre leur requête, ce qui engendre un important surcoût en messages. Notre

algorithme répare la file répartie en réassemblant dans l'ordre les morceaux encore intacts. Cette approche permet de minimiser les réémissions et de préserver l'équité en dépit des fautes.

Par ailleurs, en absence de faute, notre algorithme reste en $\mathcal{O}(log(N))$, ce qui permet un passage à l'échelle.

Nous avons conduit des expériences comparatives entre notre extension et celle proposée par Naimi-Tréhel dans un environnement réel. Dans la plupart des cas, notre algorithme s'est montré à la fois plus rapide et plus efficace. Nous avons aussi observé que le comportement de notre extension n'était pas dépendant du type d'application. Ceci est particulièrement appréciable pour des applications dont le degré de parallélisme n'est pas constant. Enfin, nous avons montré que notre algorithme résistait à l'usage de temporisateur agressif, permettant ainsi d'optimiser le temps de recouvrement.

Références

[1] I. Chang, M. Singhal, and M. T. Liu. A fault tolerant algorithm for distributed mutual exclusion. In *Proceedings of the IEEE 9th Symposium on Reliable Distributed Systems*, pages 146–154, 1990.

[2] I. Chang, M. Singhal, and M. T. Liu. An improved O(log N) mutual exclusion algorithm. In *Proceedings of the 1990 International Conference on Parallel Processing*, pages 295–302, August 1990.

[3] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7) :558–564, July 1978.

[4] D. Manivannan and Mukesh Singhal. An efficient fault-tolerant mutual exclusion algorithm for distributed systems. In *ISCA International Conference on Parallel and Distributed Computing Systems*, pages 525–530, 1994.

[5] Frank Mueller. Fault tolerance for token-based synchronization protocols. *Workshop on Fault-Tolerant Parallel and Distributed Systems, IEEE*, april 2001.

[6] M. Naimi, M. Trehel, and A. Arnold. A log (N) distributed mutual exclusion algorithm based on path reversal. *Journal of Parallel and Distributed Computing*, 34(1) :1–13, 10 April 1996.

[7] Mohamed Naimi and Michel Trehel. How to detect a failure and regenerate the token in the log(n) distributed algorithm for mutual exclusion. *Lecture Notes In Computer Science LNCS*, 312 :155–166, 1987.

[8] M. L. Neilsen and M. Mizuno. A dag-based algorithm for distributed mutual exclusion. In *Proceedings of the 11th International Conference on Distributed Computing Systems (ICDCS)*, pages 354–360, Washington, DC, 1991.

[9] Shojiro Nishio, Kin F. Li, and Eric G. Manning. A resilient mutual exclusion algorithm for computer networks. *IEEE Trans. on Parallel and Distributed Systems*, 1(3) :344–355, july 1990.

[10] K. Raymond. A tree-based algorithm for distributed mutual exclusion. *ACM Transactions on Computer Systems (TOCS)*, 7(1) :61–77, 1989.

[11] J. Sopena, L. Arantes, M. Bertier, and P. Sens. A fault-tolerant token -based mutual exclusion algorithm using a dynamic tree. In *Euro-Par'05*, LCNS, pages 654–663, Portugal, August 2005.

[12] I. Suzuki and T. Kasami. A distributed mutual exclusion algorithm. *ACM Transactions on Computer Systems (TOCS)*, 3(4) :344–349, 1985.