Un algorithme équitable d'exclusion mutuelle distribuée avec priorité

Jonathan Lejeune, Luciana Arantes, Julien Sopena, Pierre Sens

ComPAS'2013 : RenPar'21/ SympA'15/ CFSE'9
Grenoble, France, du 16 au 18 janvier 2013

Un algorithme équitable d'exclusion mutuelle distribuée avec priorité

Jonathan Lejeune, Luciana Arantes, Julien Sopena, Pierre Sens

LIP6/CNRS et INRIA,
4, place Jussieu
75252 Paris Cedex 05, France
prénom.nom@lip6.fr

Résumé
Les algorithmes distribués d'exclusion mutuelle à priorité ([4], [1], [2], [8]) permettent de définir un ordre d'accès aux sections critiques protégeant des resources partagées. Ces algorithmes sont très utiles dans les applications temps-réel ou pour assurer différents niveaux de qualité de service [6]. Cependant, la prise en compte des priorités peut conduire à des famines dans le cas où des demandes de sections critique les plus prioritaires empêchent la satisfaction des moins prioritaires. Pour palier ce problème, certains algorithmes comme celui de Kanrar-Chaki proposent d'incrémenter progressivement les priorités des requêtes pendantes mais ceci peut conduire à une violation de l'ordre des priorités. Ainsi, pour minimiser ces violations sans engendrer de famine et de surplus de messages, nous proposons des modifications de l'algorithme de Kanrar-Chaki pour ralentir la fréquence d'incrémentation des priorités. Nos évaluations des performances confirment l'efficacité de notre approche.

Mots-clés : Algorithmique distribuée, Exclusion mutuelle distribuée, Priorités

1. Introduction

L'exclusion mutuelle est l'un des paradigmes fondamentaux des systèmes distribués pour garantir des accès cohérents aux ressources partagées. Elle assure qu'au plus un processus peut exécuter une portion de code manipulant une ressource partagée, appelée section critique (propriété de sûreté) et que toute demande d'accès à la section critique sera satisfaite dans un temps fini (propriété de vivacité). Plusieurs algorithmes d'exclusion mutuelle existent dans la littérature ([5], [12], [7],[13], [11],[10]). Ils peuvent être divisés en deux catégories [14] : les algorithmes à permissions (Lamport [5], Ricart-Agrawala [12], Maekawa [7]) et les algorithmes à jeton (Suzuki-Kazami [13], Raymond [11], Naimi-Trehel [10]). Dans la première catégorie, un processus peut entrer en section critique après avoir reçu la permission de l'ensemble des autres processus (ou d'un sous-ensemble [12]). Dans la seconde catégorie, un unique jeton est partagé entre les processus. Le fait de posséder ce jeton donne le droit exclusif d'entrer en section critique.
Dans la majorité des algorithmes, les requêtes sont satisfaites selon une politique du "premier arrivé, premier servi" grâce à une horloge logique sur les requêtes ou bien grâce à l'horloge physique du détenteur du jeton (ou du futur détenteur). Cependant, cette approche n'est pas adaptée lorsque l'on doit gérer des processus avec des priorités différentes comme dans les applications temps-réel [2] [1], ou dans les systèmes basés sur des niveaux de qualité de service [6]. Pour palier ce problème, des auteurs ([4], [2], [8], [9], [1], [6]) ont proposé des algorithmes distribués d'exclusion mutuelle (généralement une version modifiée de ceux mentionnés ci-avant) où chaque requête est associée à un niveau de priorité. Cependant, l'ordre de priorité peut entraîner des famines, i.e., des temps infinis pour qu'un processus obtienne la section critique violant ainsi la propriété de vivacité. La famine apparaît lorsqu'un processus de haute priorité peut empêcher en permanence les autres processus de plus basses priorités d'exécuter la section critique. Ainsi, pour éviter ceci, les requêtes à basses priorités peuvent être dynamiquement

augmentés [4] pour atteindre éventuellement la priorité maximale. Mais cette stratégie présente un inconvénient. En effet, l'ordre des priorités des requêtes n'est plus respecté ,i.e., des inversions de priorité peuvent apparaître quand une requête avec une priorité originale basse est satisfaite avant une autre requête pendante de plus haute priorité.
Nous proposons ainsi dans cet article un nouvel algorithme d'exclusion mutuelle distribué à priorité qui minimise les violations de priorité sans introduire de famine. Ces travaux se basent sur l'algorithme de Kanrar-Chaki [4] qui utilise un arbre logique statique pour faire circuler un jeton. Les mécanismes proposés permettent de ralentir la fréquence des incrémentations de priorité des requêtes pendantes et par conséquent de réduire le nombre de violations de priorité. Ces améliorations se font sans messages supplémentaires et ne dégradent pas les temps de réponse.
L'article est organisé de la façon suivante. La section 2 présente un état de l'art sur les algorithmes d'exclusion mutuelle à priorité. On y trouvera aussi une description des algorithmes de Kanrar-Chaki [4] et de Chang [1] sur lesquels se basent notre algorithme. Notre solution est présentée dans la section 3. Les résultats de l'évaluation de performance sont présentés dans la section 4. Enfin, la section 5 conclut l'article.

2. État de l'art

Dans cette section nous résumons les principaux algorithmes distribués d'exclusion mutuelle à priorité. Par ailleurs, puisque nos mécanismes sont basés sur l'algorithme de Kanrar-Chaki [4], ce dernier est décrit plus précisément. Les algorithmes distribués d'exclusion mutuelle sont généralement une extension ou une combinaison d'algorithmes sans priorité.

L'algorithme de Goscinksi [2] est une extension de l'algorithme de Suzuki-Kasami [13]. Cet algorithme se base sur un mécanisme de diffusion. Sa complexité en nombre de messages est en O(N), N étant le nombre de sites. Les requêtes pendantes sont enregistrées dans une file globale qui est incluse dans le jeton. La famine est possible car l'algorithme peut perdre des requêtes lorsque le jeton est en transit.
L'algorithme de Mueller [8] s'inspire de celui de Naimi-Tréhel à circulation de jeton qui maintient une structure logique d'arbre dynamique pour la transmission des requêtes. Chaque nœud enregistre les dates de reception de chaque requête dans une file locale. Ces files forment une file globale virtuelle triée par priorité. Son implémentation est cependant relativement complexe et l'arbre dynamique tend à devenir une chaîne car, contrairement à l'algorithme de Naimi-Tréhel, la racine n'est pas le dernier demandeur mais le détenteur du jeton. Ainsi dans ce cas, l'algorithme présente une complexité en message de $O(\frac{N}{2})$.
L'algorithme de Housni-Tréhel [3] adopte une approche hiérarchique où les nœuds sont groupés par priorité. Dans chaque groupe, un nœud routeur représente le groupe auprès des autres groupes. Les processus d'un même groupe sont organisés en arbre statique comme dans l'algorithme de Raymond [11] et les routeurs appliquent entre eux l'algorithme de Ricart-Agrawala [12]. La famine est possible pour les processus à basses priorités, si beaucoup de requêtes de haute priorité sont pendantes. De plus, chaque processus ne peut faire des requêtes qu'en gardant toujours la même priorité (celle de son groupe).
Dans cet article nous proposons un nouvel algorithme reprenant les mécanismes des algorithmes de Kanrar-Chaki, qui est une extension de l'algorithme de Raymond, et de Chang (détaillés ci-dessous). Le but de cette proposition est d'éviter les famines tout en réduisant au maximum les inversions de priorités et en limitant le surcoût en nombre de messages.

2.1. Algorithme de Raymond

L'algorithme de Raymond [11] est un algorithme basé sur la circulation d'un jeton entre processus dans un arbre statique : seule la direction des liens change de sens pendant l'exécution de l'algorithme. Les noeuds forment ainsi un arbre orienté dont la racine est toujours le possesseur du jeton et donc le seul à pouvoir entrer en section critique. Cet arbre est utilisé pour acheminer les requêtes en direction du jeton. Cependant, pour des raisons d'efficacité, les requêtes ne sont pas retransmises par un noeud ayant déjà fait une demande, celles-ci sont alors ajoutées dans une FIFO locale. Lorsque la racine sort de la section critique et relâche le jeton, elle envoie le jeton au premier élément de sa file locale et prend ce dernier

comme père. Quand un processus reçoit le jeton, il dépile la première requête de sa file locale. Si cette requête est de lui, il peut exécuter la section critique ; sinon il retransmet le jeton au premier élément de sa file locale, le choisit comme nouveau père et si des requêtes demeurent pendantes, lui envoie une requête pour récupérer le jeton.

2.2. Algorithme de Kanrar-Chaki

L'algorithme de Kanrar-Chaki [4] introduit un niveau de priorité (valeur entière) pour chaque requête. Plus la valeur est haute, plus la priorité de la requête est haute. Ainsi, les requêtes pendantes d'une file locale d'un processus sont triées par niveau de priorité décroissant. De façon similaire à l'algorithme de Raymond, un processus souhaitant avoir le jeton envoie une requête à son père. Cependant, à la réception, le processus père insère la requête dans sa file locale en fonction de son niveau de priorité. Les retransmissions des requêtes ne se font uniquement si une requête de priorité supérieure n'a pas déjà été enregistrée et donc retransmise. L'algorithme se comporte alors comme celui de Raymond décrit précédemment. Pour éviter la famine, le niveau de priorité des requêtes pendantes est incrémenté d'un niveau à la réception de chaque requête ayant une priorité supérieure.

2.3. Algorithme de Chang

De façon similaire à l'algorithme de Kanrar-chaki, Chang a modifié l'algorithme de Raymond dans [1] pour appliquer un mécanisme de priorités dynamiques aux requêtes et pour améliorer le trafic de communication. Pour la priorité, un mécanisme appelé *aging strategy* est ajouté : si un processus sort de la section critique ou bien s'il détient le jeton sans en être demandeur et reçoit une requête, il incrémente la priorité de chaque requête dans sa file locale ; par ailleurs, à la réception du jeton qui inclut le nombre total d'exécutions de la section critique, le processus incrémente la priorité de toute ses anciennes requêtes du nombre de sections critiques qui ont été exécutées depuis son dernier passage.
Une telle approche réduit l'écart en terme de temps de réponse moyen entre les priorités (contrairement à l'algorithme de Kanrar-Chaki). Cependant, cela induit un plus grand nombre d'inversions de priorité ; ces différences de performances entre ces deux algorithmes sont présentées dans la section 4.

3. Exclusion mutuelle avec priorité

Dans la suite, nous considérons des systèmes distribués composés d'un nombre fini de nœuds fiables organisés en arbre statique. Ces nœuds n'exécutant qu'un processus, les mots "nøeud", "processus" et "site" sont interchangeables. Les communications se font par envois de messages au travers de liens point à point, fiables, FIFO et asynchrones. Enfin, les processus se comportent correctement : un processus demande la section critique si et seulement si sa précédente requête a été satisfaite et le jeton est libérée après l'exécution de la section critique.

3.1. Définition d'une violation de priorité

Nous définissons une violation de priorité à chaque fois que l'ordre des priorités n'a pas été respecté. Nous distinguons alors deux classes de requêtes :
- Une **requête favorisée** est une requête qui est satisfaite avant une requête de priorité plus importante.
- Une **requête pénalisée** est une requête qui attend le jeton pendant qu'une requête de priorité inférieure entre en section critique.

La figure 1 montre cinq requêtes avec leur priorité originale respective où les lignes horizontales représentent le temps d'attente de chaque requête. La requête A est une requête favorisée puisqu'elle est satisfaite pendant que la requête B attend le jeton. La requête B est alors une requête pénalisée. Nous notons qu'une requête peut être à la fois favorisée et pénalisée (par exemple les requêtes B et C).
Soit P l'ensemble ordonné des priorités possibles, tel que la priorité p est plus grande que la priorité p', si $p > p'$. Nous considérons que le temps est discrétisé par les événements de l'exécution de l'algorithme : demande de section critique et acquisition du jeton. Nous notons T ce temps discrétisé.
Soit le triplet $(p, t_r, t_a) \in R \subset P \times T \times T$ qui représente une requête où p est la priorité, t_r est la date où elle a été émise, et t_a la date où le jeton a été acquis. Chaque triplet est unique car il n'est pas possible que la date d'acquisition du jeton soit la même pour deux requêtes différentes. Les concepts relatifs à la violation de priorité peuvent être exprimés par :

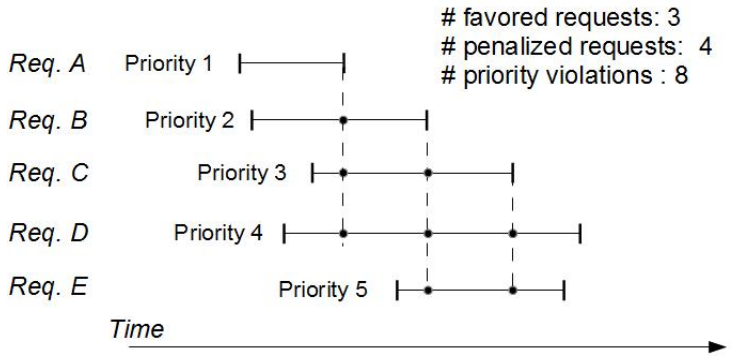


FIGURE 1 – Violation de priorité

– Le nombre de requêtes favorisées :
$\#\{(p, t_r, t_a) \in R \mid \exists (p', t'_r, t'_a) \in R, p < p' \wedge t'_r < t_a \wedge t_a < t'_a\}$
– Le nombre de requêtes pénalisées :
$\#\{(p, t_r, t_a) \in R \mid \exists (p', t'_r, t'_a) \in R, p' < p \wedge t_r < t'_a \wedge t'_a < t_a\}$
– Le nombre total de violations :
$\#\{((p, t_r, t_a), (p', t'_r, t'_a)) \in R^2 \mid p' < p \wedge t_r < t'_a \wedge t'_a < t_a\}$

Si l'on considère de nouveau la figure 1, on dénombre aussi :
– 3 requêtes favorisées (lignes en pointillés) ;
– 4 requêtes pénalisées (lignes horizontales ayant au moins un point) ;
– 8 violations (nombre de points total).

3.2. Notre algorithme

Notre solution est basée sur l'algorithme de Kanrar-Chaki car son faible nombre de messages par section critique en $\mathcal{O}(\text{Log}(N))$ permet un passage à l'échelle. De plus le mécanisme d'incrémentation de priorité assure l'absence de famine. Notre proposition est donc de modifier l'algorithme de Kanrar-Chaki pour minimiser le nombre de violations de priorité mais sans dégrader ses bonnes performances en nombre de messages envoyés et temps de réponse moyen des requêtes.

Ainsi, nous appliquons dans un premier temps l'optimisation du trafic de messages proposé dans [1] à savoir l'ajout d'une requête dans le jeton lorsqu'il reste des requêtes pendantes. Nous ajoutons ensuite deux mécanismes incrémentaux :
– le mécanisme *"level"* qui utilise des paliers pour ralentir l'incrémentation des priorités ;
– le mécanisme *"level-distance"* qui utilise en plus du mécanisme précédent le nombre de nœuds intermédiaires entre l'actuel possesseur du jeton et les nœuds demandeurs.

L'optimisation du trafic de messages et ces deux mécanismes sont décrits ci-après.

3.2.1. Optimisation des communications

Dans l'algorithme de Kanrar-Chaki, dès qu'un site ayant plusieurs requêtes en attente envoie le jeton pour satisfaire la première requête de la file, il envoie dans la foulée un message de requête pour récupérer le jeton pour pouvoir satisfaire les autres requêtes en attente. Cet envoi successif de deux messages causalement liés pose un problème d'efficacité. Une solution proposée dans l'algorithme de Chang est d'adjoindre la requête lors de l'envoi du jeton. On économise ainsi un message à chaque envoi du jeton tout en maintenant la cohérence du protocole. Un autre défaut de l'algorithme de Kanrar-Chaki, réside dans l'envoi systématique d'une requête lorsqu'un site désire entrer en section critique. Nous proposons alors de limiter cet envoi au seul cas où aucune requête pendante de priorité supérieure n'a été émise. On évite ainsi de nombreux envois inutiles en cas de forte charge.

3.2.2. Le mécanisme *"Level"*

Outre l'envoi de message inutile, l'algorithme de Kanrar-Chaki présente un grand nombre d'inversions. En effet le mécanisme d'incrémentation conduit rapidement à ce que des requêtes initialement de faible priorité obtiennent la priorité maximale. Nous proposons donc un nouveau mécanisme permettant de retarder l'incrémentation de priorité : la valeur d'une priorité d'une requête pendante n'est pas incrémentée à chaque insertion d'une nouvelle requête de plus haute priorité mais seulement après un certain nombre d'insertions.

Le nombre d'insertions nécessaires aux changements de priorité dépend alors directement de la priorité actuelle suivant une fonction exponentielle : pour augmenter sa priorité à la valeur p, une requête de priorité $p-1$ doit attendre 2^{p+c} insertions de requêtes de plus haute priorité. La constante c est un paramètre qui a pour but d'éviter que les petites priorités n'augmentent trop rapidement. Il faut alors au minimum 2^c requêtes pour passer de la priorité 0 à la priorité 1.

3.2.3. Le mécanisme *"Level-distance"*

Dans l'algorithme original de Kanrar-Chaki, les requêtes de même priorité ne sont pas ordonnées. Cependant la topologie induit des coûts de transmission différents suivant la localisation des sites demandeurs. En ajoutant de l'information à une requête, il devient envisageable d'optimiser le transfert du jeton et donc le nombre de messages ainsi que le taux d'utilisation de la section critique. On ajoute à chaque requête un compteur incrémenté à chaque retransmission. On utilise alors ce compteur pour prendre en compte la localité des requêtes. La distance d'un site R à un site S s'exprime en nombre de nœuds intermédiaires entre R et S que le jeton doit traverser. Ainsi, pour deux requêtes pendantes de même priorité, le jeton sera envoyé à la plus proche. Il est intéressant de souligner que la topologie de l'arbre a un impact direct sur l'efficacité de ce mécanisme.

4. Évaluation de performances

4.1. Protocole d'expérimentation et configuration

Les expériences ont été réalisées sur une grappe de 20 machines (un processus par machine, i.e., un processus par carte réseau). Chaque nœud a deux processeurs Xeon 2.8GHz et 2 Go de mémoire RAM, s'exécute sur un noyau Linux 2.6. Les nœuds sont reliés par un switch de 1 Gbit/s en ethernet. Les algorithmes ont été implémentés en C++ en utilisant l'intergiciel OpenMPI. Une application se caractérise par :

- α : temps d'exécution de la section critique.
- β : intervalle de temps d'attente entre la sortie d'une section critique et l'émission de la prochaine requête.
- γ : temps de transmission réseau entre deux nœuds voisins.
- ρ : le rapport $\beta/(\alpha+\gamma)$, qui exprime la fréquence avec laquelle la section critique est demandée. La valeur de ce paramètre est inversement proportionnelle à la charge de requêtes pendantes (une petite valeur implique une grande charge et vice versa).
- θ : Le temps d'une expérience.

Par la suite nous comparerons les algorithmes suivant les métriques suivantes :

- *Le nombre de messages par requête* : pour chaque type de message, on étudie le rapport du nombre total de messages envoyés sur le réseau sur le nombre total de requêtes.
- *Le nombre de violations de priorité* : ceci exprime la troisième expression formelle de violation dans la section 3.1 en pourcentage de requêtes émises pendant l'expérience.
- *Le temps de réponse* : Le temps entre le moment où une requête est émise et le moment où le jeton donne accès à la section critique.
- *Le taux d'utilisation de la section critique* : le rapport du temps total passé en section critique sur θ.

Pour toutes les expériences, nous considérons une topologie logique en arbre binaire et huit niveaux de priorité. Dans les figures suivantes, *"Notre_solution"* correspond à l'algorithme de Kanrar-Chaki modifié pour l'optimisation du trafic de communication expliqué en section 3.2.1 tandis que *"Notre_solution_Level"* et *"Notre_solution_LevelDistance"* correspondent à cette optimisation de trafic en y appliquant respectivement les mécanismes *"Level"* et *"Level-Distance"* (voir sections 3.2.2 et 3.2.3). Nous avons également inclus l'algorithme de Chang (voir section 2) dans nos expériences d'évaluation de performances. Les

algorithmes de Kanrar-Chaki, Chang et *"Notre_solution"* seront appelés algorithmes *"no-level"* tandis que les algorithmes *"Notre_solution_Level"* et *"Notre_solution_LevelDistance"* seront appelés algorithmes *"level"*.

4.2. Étude de performances pour une charge moyenne

Dans cette section, nous présentons et discutons des résultats d'évaluation de performances lorsque la charge de requêtes pendantes reste constante dans une même expérience.

Les processus émettent une requête à la fois, le temps séparant ces requêtes étant choisi aléatoirement suivant une loi de Poisson dont la moyenne est déterminée par le paramètre ρ (charge du système). À chaque émission de requête, le site choisit aléatoirement une priorité suivant une distribution uniforme. De façon à obtenir des mesures en régime stationnaire on ne prend pas en considération les cinq premiers accès à la section critique de chaque site. Par conséquent, la charge moyenne de requêtes pendantes reste constante pendant toute la durée de l'expérience.

Notre étude est divisée en deux parties : nous considérons d'abord une charge moyenne fixe, et ensuite nous évaluons l'impact de la variation de charge sur le comportement des algorithmes. Pour toute expérience, nous avons considéré que le temps de transmission réseau entre deux nœuds voisins est d'environ 100 μs.

4.2.1. Performance globale des algorithmes

La figure 2 résume le comportement des algorithmes quand la charge est fixée aux alentours de 55% de sites en attente de la section critique.

Nous pouvons observer dans la figure 2(a) que l'algorithme de Kanrar-Chaki présente 25 % de violations en moins que l'algorithme de Chang. Cependant ceci se fait au détriment de la complexité en nombre de messages (figure 2(b)) : 40 % de messages supplémentaires. Les performances de *"Notre_solution"* montrent que le simple ajout des mécanismes d'optimisation des messages à l'algorithme de Kanrar-Chaki suffisent à obtenir une complexité comparable à celle de Chang tout en conservant le même taux de violations.

Toutefois, les performances en termes de violations obtenues par *"Notre_solution"* sont encore très importantes. L'ajout du mécanisme *"level"* permet de réduire considérablement ce nombre d'un facteur 25. Ce bon résultat sur la métrique la plus importante pour un algorithme à priorité se fait au détriment du nombre de messages. En effet, les sites atteignent la priorité maximale plus lentement et sont donc susceptibles de retransmettre d'avantage de requêtes. Pour contrebalancer le surcoût en messages généré par le mécanisme *"level"*, nous avons ajouté le mécanisme *"distance"* pour obtenir l'algorithme *"Notre_solution_LevelDistance"*. Ce mécanisme permet d'économiser 15 % des messages tout en conservant le très faible taux d'inversions de priorité.

En ce qui concerne le temps de réponse moyen, nous observons dans la figure 2(c) que l'algorithme original de Kanrar-Chaki a un comportement régulier (en forme d'escalier),i.e., quand la priorité augmente, le temps de réponse moyen diminue. À l'inverse, avec le mécanisme *"level"*, les écarts de latence entre les différentes priorités sont moins réguliers : la priorité 0 voit sa latence largement augmentée (traitement "best-effort"), tandis que les plus hautes priorités bénéficient d'une forte amélioration des temps d'accès. Enfin, lorsque l'on compare les latences de *"Notre_solution_Level"* avec *"Notre_solution_LevelDistance"*, on remarque qu'il n'y a pas de différence sur les latences moyennes des différentes priorités. Le gain en nombre de messages ne se fait qu'au prix d'une augmentation de l'écart type pour les faibles priorités.

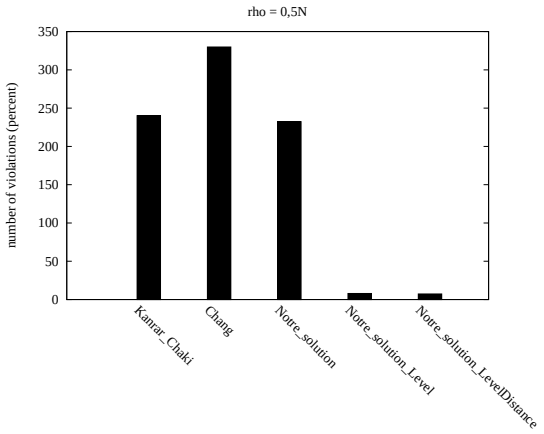
Pour terminer, la figure 2(d) nous permet de vérifier que les performances de nos différents mécanismes en termes de respect des priorités ne se font pas au dépend des performances globales : le taux d'utilisation du jeton reste inchangé autour de 80%.

En conclusion les figures confirment que le retard de l'incrémentation est essentiel pour le respect de l'ordre des priorités tandis que la localité des requêtes est utile pour la réduction du nombre de messages générés par l'algorithme.

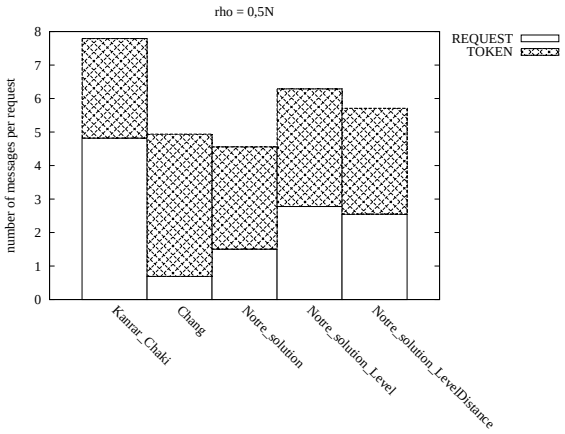
4.2.2. Étude des différents types d'inversion

La figure 3 présente une série de résultats visant à affiner l'étude de performance suivant le type de violation. Ainsi, les sous-figures montrent :

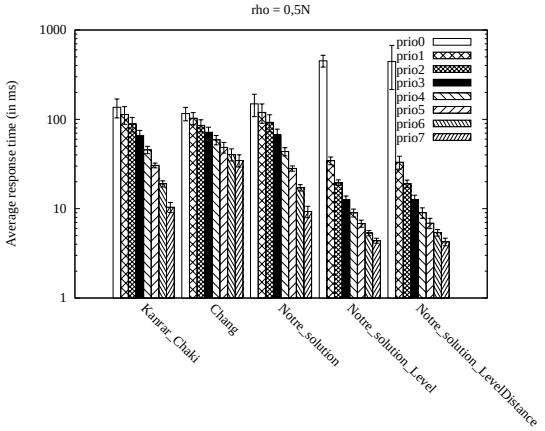
– Figures 3(a) et 3(b) : le pourcentage de requêtes pénalisées et le nombre de requêtes favorisées pour



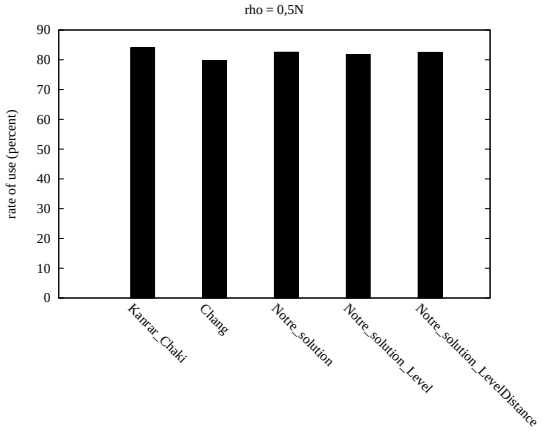
(a) Pourcentage de violations par rapport au nombre total de requêtes



(b) Nombre de messages envoyés sur le réseau par requête classifié par type



(c) Temps de réponse moyen pour entrer en section critique par priorité



(d) Taux d'utilisation de la section critique

FIGURE 2 – Performances des algorithmes à priorité

chaque algorithme.
- Figures 3(c) et 3(d) : pour chaque niveau de priorité, le pourcentage de requêtes pénalisées et favorisées respectivement pour chaque algorithme.
- Figure 3(e) et 3(f) : Le nombre moyen de requêtes favorisées par requête pénalisée et le nombre moyen de requêtes pénalisées par requête favorisée

Comme nous pouvons l'observer dans les figures 3(a) et 3(b), les algorithmes *"no-level"* induisent plus de requêtes pénalisées que de favorisées mais pour les algorithmes *"level"*, ces deux métriques ont la même valeur.

Dans la figure 3(c), on retrouve cette différence en valeur absolue pour chacune des priorités. Mais cette figure nous montre aussi des différences sur la distribution des requêtes pénalisées au sein d'un même algorithme : dans les algorithmes *"level"*, le nombre de requêtes pénalisées augmentent linéairement avec la priorité, tandis que dans les autres algorithmes, les requêtes les plus pénalisées sont les requêtes de priorités intermédiaires.

Pour comprendre ces différences, il faut savoir que le risque d'être pénalisé est un compromis entre deux phénomènes :
- le nombre de requêtes susceptibles de dépasser une requête plus prioritaire dépend uniquement de la priorité initiale, et non de l'algorithme. En effet, plus la priorité d'une requête est haute, plus le nombre de requêtes de priorité inférieure augmente. Ainsi, la probabilité de se faire dépasser par une requête de priorité inférieure augmente.
  plus le nombre de requêtes susceptibles de dépasser est grande.
- La probabilité de dépasser qui dépend du mécanisme d'incrémentation et de la priorité initiale.

Dans les algorithmes *"level"*, le retard de l'incrémentation rend le deuxième phénomène négligeable. Le premier phénomène explique à lui seul la linéarité entre le nombre de violations et la priorité. À l'inverse, dans le cas des autres algorithmes, l'augmentation est rapide et donc non négligeable, on remarque ainsi que :
- La vitesse d'augmentation est très rapide pour les faibles priorités (priorités 0 et 1) puisqu'elles ont de grande chance de se voir dépasser par des requêtes de priorité supérieure. Cette augmentation rapide a pour effet de fortement pénaliser les priorités 3.
- Les requêtes de plus hautes priorités (priorités 6 et 7) peuvent être doublées par plus de requêtes mais ces dernières (priorités 5 et 4) n'augmentent que plus lentement leur priorité ou sont très éloignées (priorités 0 et 1).

Dans ces algorithmes on observe donc un compromis entre les deux phénomènes conduisant à pénaliser d'avantage les priorités intermédiaires.

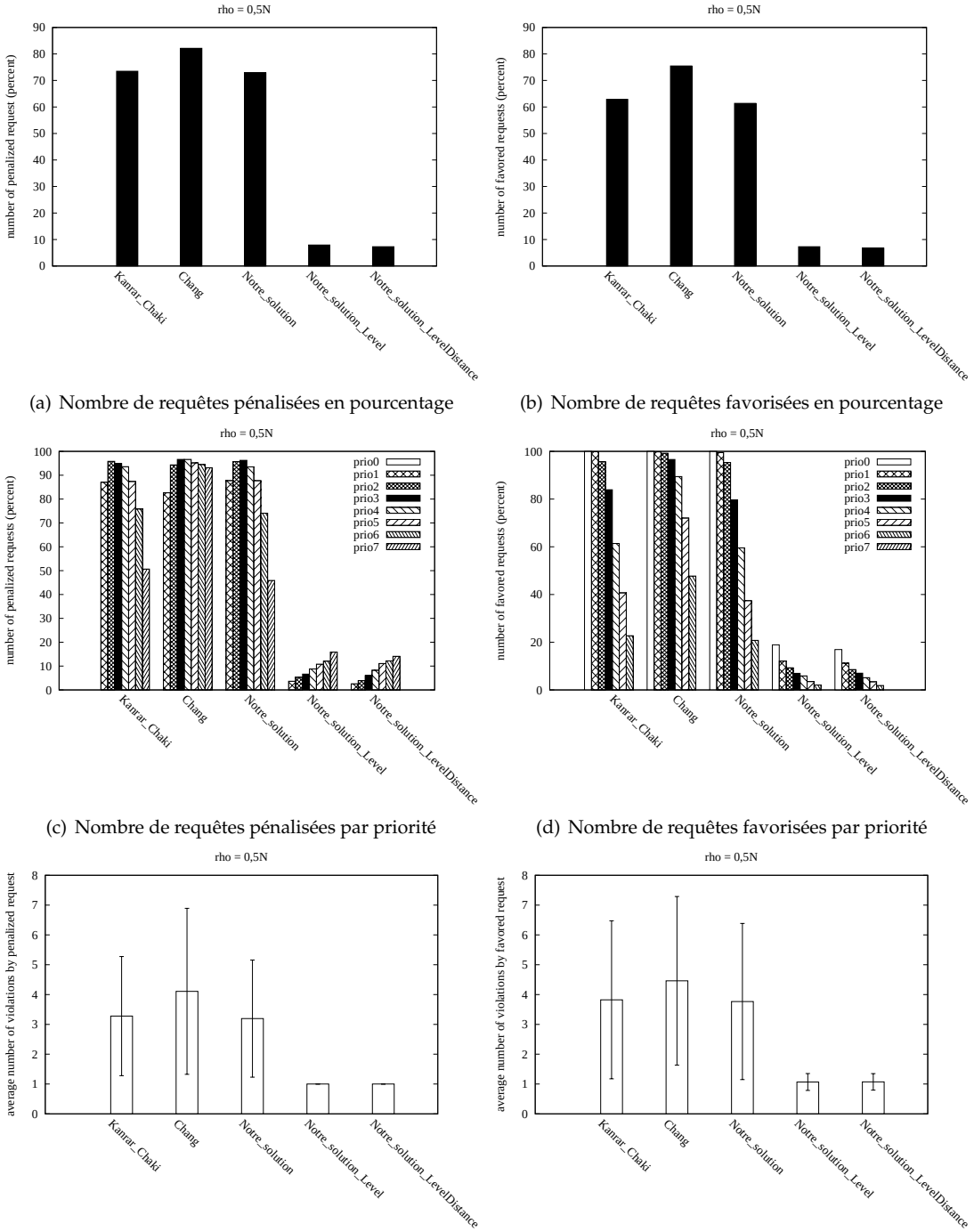
La figure 3(e) montre que pour les algorithmes *"level"*, les requêtes pénalisées ne sont doublées qu'une seule fois (écart-type nul), tandis qu'elles le sont en moyenne 3 à 4 fois dans les autres approches. En considérant les figures 3(a) et 3(b), on peut conclure que les requêtes pénalisées sont moins nombreuses et qu'elles sont doublées par moins de requête. Ceci explique les très bons résultats en termes de violations globales observées dans la figure 2(a). Ce discours s'applique également à la figure 3(f).

4.3. Impact de la charge sur les performances des algorithmes

Nous présentons maintenant les résultats d'une étude de l'impact de la charge, i.e., le nombre de requêtes pendantes sur les performances. Nous avons ainsi renouvelé les expériences avec différentes valeurs de ρ proportionnellement au nombre de processus N : 0.1N, 0.375N, 0.5N, 1N, 3N, 5N, et 10N qui correspondent respectivement à 86.7%, 64.2%, 54.2%, 15.2%, 0.8%, 0.5% et 0.2% de processus en attente (abscisse des figures 4 et 5). Dans la figure 4, nous étudions le nombre de messages par requête, le taux d'utilisation de la section critique et le nombre de violations tandis que dans la figure 5, nous affinons l'analyse des violations.

4.3.1. Performance globale des algorithmes

Tout d'abord remarquons sur la figure 4(c) que les algorithmes *"level"* sont insensibles aux fortes charges : le nombre de violations reste faible quelle que soit la valeur de ρ. En revanche, le nombre de violations augmente considérablement pour les algorithmes *"no-level"* lorsque la charge augmente. En effet, la forte charge accroît le nombre de requêtes concurrentes, ce qui les conduit rapidement à atteindre la priorité maximale. Ces algorithmes ne pouvant plus distinguer les priorités, génèrent un grand nombre



(a) Nombre de requêtes pénalisées en pourcentage

(b) Nombre de requêtes favorisées en pourcentage

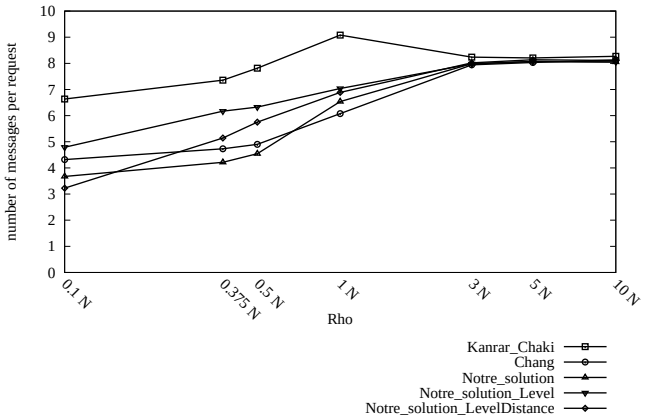
(c) Nombre de requêtes pénalisées par priorité

(d) Nombre de requêtes favorisées par priorité

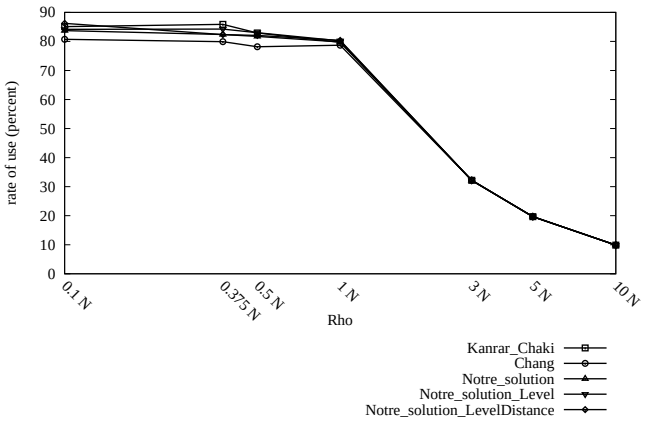
(e) Nombre moyen de requêtes favorisées par requête pénalisée

(f) Nombre moyen de requêtes pénalisées par requête favorisée

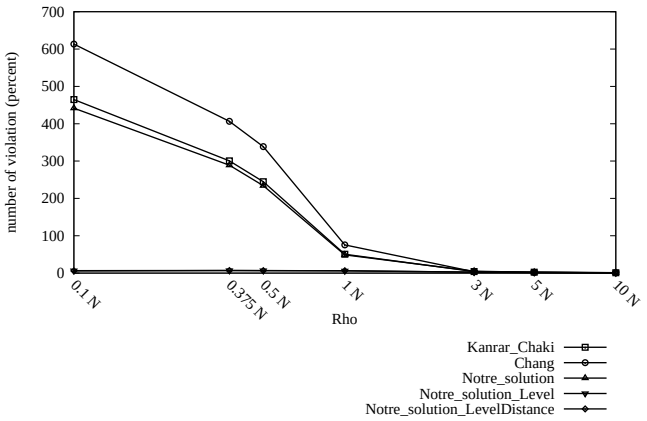
FIGURE 3 – Analyse de la violation de priorité



(a) Nombre de messages envoyés sur le réseau par requête



(b) Taux d'utilisation de la section critique

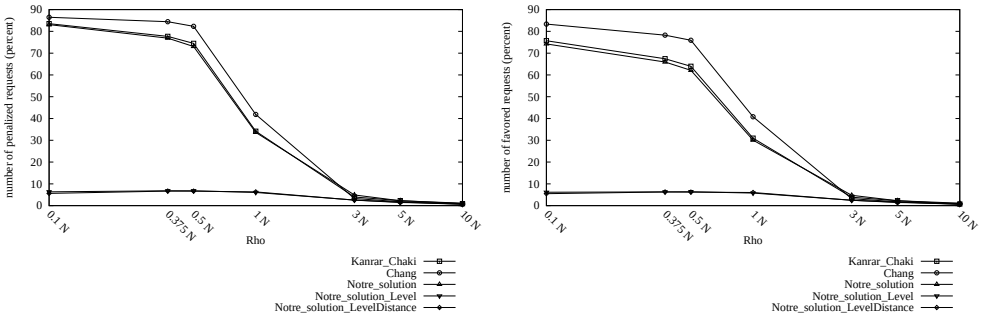


(c) Nombre total de violations en pourcentage de requêtes

FIGURE 4 – Etude de l'impact de la charge

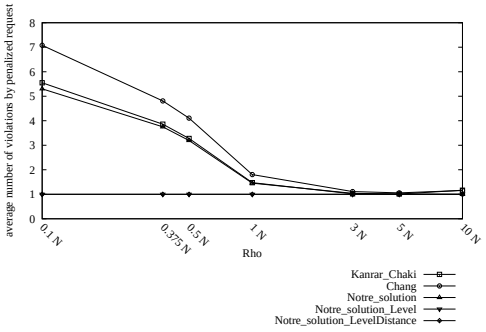
de violations.
D'autre part, nous observons sur la figure 4(a) que le nombre de messages diminue lorsque la charge augmente quel que soit l'algorithme. Ce phénomène est une conséquence directe de l'algorithme de Raymond : un nœud ne retransmet pas de requête si il est déjà demandeur de la section critique. On retrouve aussi sur cette figure, comme observé dans la section 4.2.1, le surcoût en messages engendré par le mécanisme *"level"*, ainsi que le gain apporté par le mécanisme *"distance"*. Il est cependant intéressant de remarquer que l'efficacité du mécanisme *"distance"* est d'autant plus importante que la charge augmente. L'algorithme devenant plus économe pour une charge de $0,1N$ (86,7 %). Ceci est particulièrement intéressant pour des applications présentant des pics de charges.
Concernant le taux d'utilisation de la section critique, nous observons dans la figure 4(b) que tous les algorithmes ont le même comportement, i.e., pour une valeur de ρ donnée, tout algorithme satisfait le même nombre de requêtes.
Il est important de souligner, que les trois graphiques de la figure 4 confirment qu'en cas de faible charge, les algorithmes *"no-level"* et *"level"* se comportent de la même manière.

4.3.2. Étude des différents types d'inversion

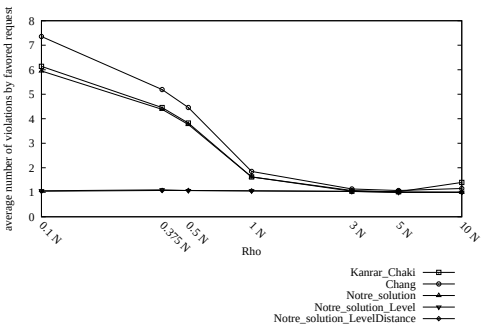


(a) Nombre de requêtes pénalisées en pourcentage

(b) Nombre de requêtes favorisées en pourcentage



(c) Nombre de requêtes favorisées par requête pénalisée



(d) Nombre moyen de requêtes pénalisées par requête favorisée

FIGURE 5 – Etude de l'impact de la charge sur la violation de priorité

Nous représentons respectivement dans les figures 5(a), 5(b), 5(c) et 5(d), le nombre de requêtes pénalisées, le nombre de requêtes favorisées, le nombre moyen de fois où une requête est pénalisée et le nombre moyen de fois où une requête est favorisée.
Dans la figure 5(a), nous remarquons, qu'en cas de faible charge ($10N$), il n'y a aucune requête pénalisée quel que soit l'algorithme. Lorsque la charge commence à augmenter (de $10N$ à $3N$), quelques requêtes pénalisées commencent à apparaître.

Avec une charge moyenne (de 3N à 0.5N), nous remarquons que seuls les algorithmes *"no-level"*, augmentent de manière significative les requêtes pénalisées (jusqu'à 80%) tandis que les algorithmes *"level"* continuent d'augmenter très légèrement.
Finalement lorsque la charge est forte, le pourcentage de requêtes pénalisées augmente légèrement pour atteindre 85% pour les algorithmes *"no-level"*. Ceci représente la proportion des requêtes ayant une priorité initiale strictement supérieure à zéro. En effet, les requêtes avec une priorité initiale de 0 ne peuvent pas être pénalisées. Autrement dit 100% des requêtes pénalisables sont pénalisées. On retrouve le même type de comportement pour les requêtes favorisées (figure 5(b)).
Pour terminer, nous nous intéressons au nombre moyen de requêtes doublant une requête pénalisée (figure 5(c)). En comparant ces résultats avec la figure 5(a), on remarque que si le nombre de requêtes pénalisées devient important (40%) pour une charge intermédiaire (1N), elles ne sont doublées en moyenne que deux fois. Une fois passé le seuil de $0,5$N, presque toutes les requêtes pénalisables sont pénalisées, mais le nombre de requêtes les dépassant continue d'augmenter fortement. Ceci explique pourquoi le nombre total de violations continue d'augmenter en forte charge (entre $0,5$N et $0,1$N) dans la figure 4(c). On comprend alors les mauvaises performances en forte charge : toutes les requêtes pénalisables sont doublées environ 6 fois.

5. Conclusion

Dans cet article nous avons présenté plusieurs mécanismes basés sur l'algorithme de Kanrar-Chaki qui utilisent des priorités dynamiques afin d'assurer la vivacité. Ces mécanismes permettent de réduire considérablement le nombre de violations de l'algorithme original. En retardant l'incrémentation des priorités, ils offrent un très bon compromis entre la famine et l'inversion de priorité. Les évaluations de performances confirment que la prise en compte de la localité des requêtes réduit sensiblement le nombre de messages. Elles montrent aussi que les gains obtenus par rapport aux algorithmes de la littérature sont d'autant plus importants que la charge est haute (avec un facteur 100 pour les violations). Ainsi nos mécanismes s'adaptent très facilement aux applications présentant des pics de charges. Pour aller plus loin dans l'analyse, il serait alors intéressant d'étudier ces mécanismes lorsque que la charge varie dynamiquement pendant l'exécution de l'expérience.

Bibliographie

1. Chang (Y.-I.). – Design of mutual exclusion algorithms for real-time distributed systems. *J. Inf. Sci. Eng.*, vol. 11, n4, 1994, pp. 527–548.
2. Goscinski (A. M.). – Two algorithms for mutual exclusion in real-time distributed computer systems. *J. Parallel Distrib. Comput.*, vol. 9, n1, 1990, pp. 77–82.
3. Housni (A.) et Trehel (M.). – Distributed mutual exclusion token-permission based by prioritized groups. *In : AICCSA*, pp. 253–259.
4. Kanrar (S.) et Chaki (N.). – Fapp : A new fairness algorithm for priority process mutual exclusion in distributed systems. *JNW*, vol. 5, n1, 2010, pp. 11–18.
5. Lamport (L.). – Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, vol. 21, July 1978, pp. 558–565.
6. Lejeune (J.), Arantes (L.), Sopena (J.) et Sens (P.). – Service level agreement for distributed mutual exclusion in cloud computing. *In : 12th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing (CCGRID'12)*. pp. 180–187. – IEEE Computer Society Press.
7. Maekawa (M.). – A $\sqrt{N}$ algorithm for mutual exclusion in decentralized systems. *ACM Trans. Comput. Syst.*, vol. 3, May 1985, pp. 145–159.
8. Mueller (F.). – Prioritized token-based mutual exclusion for distributed systems. *In : IPPS/SPDP*, pp. 791–795.
9. Mueller (F.). – Priority inheritance and ceilings for distributed mutual exclusion. *Real-Time Systems Symposium, IEEE International*, vol. 0, 1999, p. 340.
10. Naimi (M.) et Trehel (M.). – An improvement of the log(n) distributed algorithm for mutual exclusion. *In : ICDCS*, pp. 371–377.

11. Raymond (K.). – A tree-based algorithm for distributed mutual exclusion. *ACM Trans. Comput. Syst.*, vol. 7, n1, 1989, pp. 61–77.
12. Ricart (G.) et Agrawala (A. K.). – An optimal algorithm for mutual exclusion in computer networks. *Commun. ACM*, vol. 24, January 1981, pp. 9–17.
13. Suzuki (I.) et Kasami (T.). – A distributed mutual exclusion algorithm. *ACM Trans. Comput. Syst.*, vol. 3, n4, 1985, pp. 344–349.
14. Velazquez (M. G.). – *A Survey of Distributed Mutual Exclusion Algorithms*. – Rapport technique, Colorado state university, 1993.