Réduire le coût de communication des algorithmes à phases par l'agrégation de messages : application à Paxos

Célia Mahamdi, Jonathan Lejeune, Julien Sopena, Pierre Sens, Mesaac Makpangou

Compas'2021 : Parallélisme/ Architecture/ Système / Temps Réel
MILC - Lyon, France, 5-9 juillet 2021

Réduire le coût de communication des algorithmes à phases par l'agrégation de messages : application à Paxos

Célia Mahamdi, Jonathan Lejeune, Julien Sopena, Pierre Sens et Mesaac Makpangou

Sorbonne Université - CNRS - Inria
4, place Jussieu,
F-75005, Paris, France
Email : firstname.lastname@lip6.fr

Résumé

Les algorithmes à phases ou à round comme l'algorithme de Paxos sont généralement très coûteux en messages. Si l'ajout d'un temporisateur et/ou de mécanisme système d'agrégation dans les couches basses permet d'en réduire le coût, cela se fait au prix d'une dégradation de leurs performances. Dans cet article, nous proposons un mécanisme générique d'agrégation des messages qui permet d'exploiter les propriétés de ces algorithmes pour réduire le nombre de message sans dégrader les performances de l'algorithme initial. En outre, notre mécanisme ne nécessite pas de modifications de l'algorithme ciblé et peut adresser le cas d'algorithmes s'exécutant en concurrence sur les mêmes nœuds. Nos expérimentations sur l'algorithme de Paxos ont ainsi montré qu'il est possible d'économiser jusqu'à environ 40% de messages sans dégrader significativement la latence des consensus.

Mots-clés : algorithmes distribués, Paxos, agrégation de messages, évaluation expérimentale

1 Introduction

Dans les systèmes informatiques contemporains (Cloud Computing, Fog Computing), les ressources de calcul, de stockage et de réseaux sont mutualisées entre plusieurs applications qui sont déployées sur une infrastructure distribuée et large échelle. Ces applications implantent des algorithmes qui offrent des solutions aux problèmes fondamentaux rencontrés dans les systèmes distribués (consensus [7], exclusion mutuelle [12][11], élection de leader [14] etc.). Beaucoup de ces algorithmes reposent sur une succession de phases ou de round. Ainsi, un algorithme à phases se définit par l'exécution d'une série de mécanismes (e.g. une communication de groupe), de communications, où le début de l'étape $i+1$ est conditionné par la réalisation de tout ou une partie de l'étape i. Parmi les algorithmes à phases, on trouve, notamment, des algorithmes résolvants le problème du consensus (Paxos [9], ZAB [8], two-phases commit [13]) permettant à un ensemble de participants de se mettre d'accord sur une valeur (donnée, action ...) dans un environnement non fiable. Bien que ces algorithmes soient indispensables au bon fonctionnement de certaines applications (SGBD, Google Spanner [4]), ils n'en restent pas moins très coûteux en envois de messages. Par exemple, l'algorithme de Paxos ([9]), qui est un des algorithmes les plus populaires [1], repose sur une série de diffusions à l'ensemble des participants synchronisé par l'attente d'un quorum de réponses. Il existe de très nombreuses versions du protocole de Paxos ([6],[10], [2]) mais dans sa version la plus répandue, la complexité en message est quadratique par rapport au nombre de participants ce qui complexifie le passage à l'échelle.

Une technique classique pour réduire la complexité en messages dans les systèmes distribués est l'agrégation (piggybacking). Elle repose sur le multiplexage au sein d'un seul message physique d'un ensemble de messages relatifs à plusieurs applications et/ou plusieurs phases d'un

algorithme. Cette technique peut d'ailleurs se faire de façon transparente au niveau de la couche réseau. Malheureusement pour être efficace, elle nécessite de bufferiser les envois, ce qui dégrade les performances du système. Cette bufferisation étant réalisée à l'aveugle, il est a priori impossible de savoir si le message retardé pourra être ou non multiplexé.

Pourtant dans le cas des algorithmes à phases, nous connaissons les différentes étapes de l'algorithme. Nous pouvons ainsi savoir si un participant s'adressera à un autre dans un futur proche, et donc, si il est pertinent d'attendre pour agréger un message. Partant de cette constatation, nous avons développé une solution générique et non-intrusive, qui permet d'exploiter cette propriété des algorithmes à phases pour réduire leur complexité en messages sans dégrader les performances. Celle-ci se présente comme une couche sous-jacente et une nouvelle API de communication introduisant le concept de promesse d'envoi de message (pledge).

Une première série d'expérimentations sur un simulateur, montre que l'utilisation de notre mécanisme sur un ensemble d'instances de Paxos permet d'économiser jusqu'à environ 40% de messages sans dégrader significativement la latence des consensus.

L'article est organisé de la façon suivante : dans la section 2 nous exposerons différentes solutions existantes relatives à nos travaux, dans la section 3 nous présenterons notre mécanisme d'agrégation, la section 4 décrira notre évaluation expérimentale et enfin nous conclurons et évoquerons nos perspectives de travaux futurs dans la section 5.

2 État de l'art

Dans un premier temps, nous discuterons des mécanismes d'agrégation et de l'utilité de prendre en compte les caractéristiques applicatives. Nous nous intéresserons ensuite à l'algorithme de Paxos.

2.1 Mécanismes d'agrégation

Il existe différents mécanismes d'agrégation. Le protocole TCP en propose un. Il repose sur l'algorithme de Nagle [16] qui reste aujourd'hui une fonctionnalité standard de TCP. Les paquets TCP/IP possèdent un en-tête de 40 octets. Lorsqu'un nombre conséquent de petits messages sont envoyés, TCP voit son efficacité limitée et peut entraîner une congestion du réseau. L'idée générale derrière l'algorithme de Nagle est la suivante : les données à envoyer sont agrégées tant que les données précédentes n'ont pas été acquittées ou que le tampon n'est pas plein.

Nous nous sommes basés sur l'hypothèse que les nœuds physiques d'une même infrastructure puissent héberger plusieurs applications distribuées s'exécutant concurremment (et utilisant des algorithmes à phases). À ce jour et à notre connaissance, nous n'avons pas trouvé de mécanismes d'agrégation qui prennent en compte les caractéristiques applicatives. Les mécanismes se situent donc au niveau de la couche réseau. Ainsi, si nous utilisions le mécanisme propre à TCP par exemple, d'autres applications pourraient pâtir de cette agrégation (chat, visioconférence, temps réel...). La prise en compte des caractéristiques des applications nous offre, quant à elle, la possibilité d'utiliser plus pertinemment l'agrégation pour des applications choisies et qui en auraient l'utilité.

2.2 Paxos

L'algorithme de Paxos est un des algorithmes distribués les plus utilisés (cf. annexe A.1 qui présente rapidement l'algorithme). Ainsi, de nombreux travaux ont été publiés pour optimiser son fonctionnement et notamment pour en réduire la complexité en messages (Fast Paxos [10], Single-Decree Paxos [6], Multi-Paxos [5], etc.). Mais ces travaux adressent uniquement l'optimisation d'une instance de Paxos sans considérer les autres instances ni le reste des applications s'exécutant sur les noeuds. Ces travaux sont donc complémentaires à notre mécanisme qui cherche à optimiser globalement le nombre de messages. Dans la suite de l'article, nous nous concentrerons sur la version classique de Paxos, mais des travaux en cours étudient le comportement de notre mécanisme sur des versions plus optimisées.

3 Mécanisme d'agrégation

Nos ensembles d'acteurs peuvent être amenés à lancer différentes instances de l'algorithme de Paxos de manière aléatoire et simultanément. L'algorithme de Paxos, se révélant très vite coûteux en nombre de messages, nous proposons une manière de réduire ce coût tout en préservant une latence raisonnable.

Une première approche naïve serait de faire une agrégation systématique des messages durant un certain temps arbitraire ou jusqu'à ce que le tampon soit complètement rempli. Cette méthode permet de réduire drastiquement le nombre de messages, au détriment, cependant de la latence qui se retrouve grandement impactée. Notre idée permet, selon nos besoins, d'instaurer un entre-deux.

Notre mécanisme d'agrégation découle d'une observation que nous avons faîtes en étudiant l'algorithme de Paxos dans un contexte d'applications multiples.

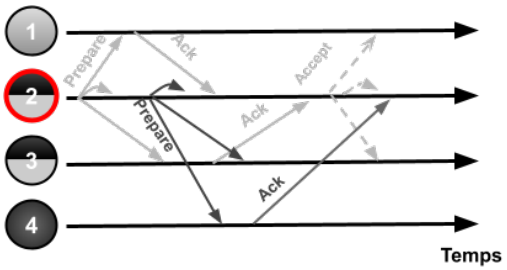

FIGURE 1 – Début avec deux ensembles

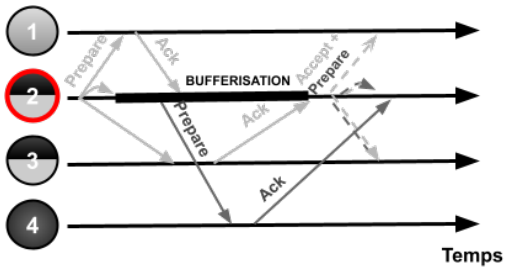

FIGURE 2 – Mécanisme d'agrégation

Supposons que nous ayons deux ensembles représentant chacun une application en cours d'exécution. L'ensemble gris et l'ensemble noir sont composés respectivement des nœuds 1,2,3 et 2,3,4. Ces deux ensembles sont donc en intersection via les nœuds 2 et 3. Chaque ensemble, selon ses besoins, peut lancer des instances de l'algorithme de Paxos au cours du temps. Dans la Figure 1, le leader (représenté par l'anneau rouge, ici le nœud 2) démarre une instance pour l'ensemble gris puis pour l'ensemble noir.

L'idée générale est de rassembler certains messages en fonction des phases de l'algorithme. En effet, en fonction des ensembles auxquels un acteur appartient ce derniers va être amené à s'adresser très souvent aux mêmes autres acteurs.

Grâce à la connaissance des phases de l'algorithme, nous sommes en mesure de prédire quand un nœud A s'adressera à un nœud B.Ainsi, pour une instance donnée, lorsque l'on attend le début d'une phase suivante, il est possible d'agréger tous les messages allant de A vers B de n'importe quelle autre instance. Puisque cette phase d'attente est inhérente à l'algorithme (et donc inévitable), elle permet de compenser (et donc limiter) la latence engendrée par l'agrégation.

3.1 Mise en place des pledges

Il est nécessaire dans un premier temps de savoir si il est pertinent de retarder un message. En effet, l'agrégation systématique peut être contre-productive car elle peut bloquer l'avancement de l'exécution de l'algorithme ce qui affectera grandement la latence. Pour passer d'une phase à une autre, il est nécessaire de recevoir un quorum de réponses venant d'autres participants (phase de préparation, phase d'acceptation). La connaissance de l'algorithme nous indique qu'une fois ce quorum atteint un certain type de messages sera envoyé (ici le message *accept* clôturant la phase de préparation et le message *decide* indiquant la fin de la phase d'acceptation). Nous pouvons donc assurer, qu'à court terme, les acteurs concernés par ces messages seront de nouveau contactés. Ainsi, dans l'attente de ces quorums, les messages à destination de ces acteurs peuvent être retardés. Nous appelons ce mécanisme un **pledge** : un acteur peut retarder l'envoi d'un message s'il s'engage à l'envoyer à terme.

Reprenons l'exemple de la Figure 1, et illustrons notre mécanisme d'agrégation. Dans la Figure 2, le leader démarre une instance de consensus sur l'ensemble gris. Dès lors, nous savons qu'il s'adressera de nouveau aux acteurs 1,2 et 3 pour l'ensemble gris une fois qu'il aura reçu une majorité de *ack* gris. Nous pouvons donc utiliser ce temps d'attente fourni par l'algorithme pour agréger les messages. Ainsi, lorsque l'acteur 2 démarre une instance pour l'ensemble noir, l'acteur 4 n'étant pas concerné par le consensus gris va recevoir le message de type *prepare* normalement. En revanche, pour les acteurs 2 et 3, le *prepare* peut être retardé, car nous pouvons garantir que le leader s'adressera à termes à ces dernier via un message de type *accept*. Le message *prepare* de l'instance noire peut ainsi être agrégé avec le message *accept* de l'instance grise.

Modifier et s'introduire dans l'algorithme de Paxos pour intégrer directement ce mécanisme n'est pas trivial. En effet il est nécessaire de prévoir toute la combinatoire des cas d'agrégation ce qui rend notre mécanisme difficilement extensible à d'autres algorithmes que Paxos. Nous avons donc introduit une nouvelle API réseau afin d'être le moins intrusif possible.

3.2 Nouvelle API

Nous avons enrichi la primitive *send(msg,dest)* de deux nouveaux champs :

— $probaPledge$: la probabilité d'agréger le message. Une valeur 0 signifie que le message sera envoyé immédiatement (cas du *send* original) alors que la valeur 1 signifie que le message sera mis en attente dans le tampon d'envoi. Le fait d'être retardé avec une certaine probabilité permet de paramétrer l'importance du message dans l'avancement de l'exécution de l'algorithme et donc de contrôler la dégradation de la latence.

— $timeout$: le temps maximal durant lequel le message restera dans le tampon d'envoi. Ceci assure qu'un message retardé suite à un pledge sera à terme envoyé assurant ainsi les propriétés de sûreté et de vivacité de l'algorithme.

Une nouvelle couche intermédiaire entre le réseau et l'application a été introduite. Dans cette couche, chaque acteur maintient un tampon pour chaque nœud destinataire auquel il est susceptible de s'adresser. Dès que la primitive *send* est appelée, deux cas de figures se présentent. Si la $probaPledge$ est nulle alors le message est envoyé directement accompagné des messages éventuellement présents dans le tampon associé au même destinataire. Dans le cas contraire, un tirage aléatoire est effectué selon la probabilité $probaPledge$ afin de déterminer si le message sera retardé. Le cas échéant, le message est placé dans le tampon de son destinataire et un timeout de valeur $timeout$ est armé. Au déclenchement d'un timeout, si le message associé est toujours présent dans le tampon, alors tout son contenu est vidé et envoyé dans un seul et même message.

3.3 Modification de Paxos

Au niveau de la couche applicative exécutant Paxos, il est nécessaire de transformer toutes les variables propres à Paxos en vecteur ayant pour taille le nombre d'ensembles existants. Ainsi, chaque case d'un vecteur correspond aux variables nécessaires pour un ensemble. Cette enrichissement de connaissance permet de savoir pour n'importe quel ensemble de participants si il existe une instance de Paxos qui serait en attente de démarrer une nouvelle phase et donc de pouvoir honorer un pledge. Pour cela, nous ajoutons une liste $future_{comm}$ qui maintient la liste des participants pour lesquels il est prévu de s'adresser ultérieurement. Avant chaque envoi, nous vérifions si le destinataire fait partie d'un ensemble de $future_{comm}$. Dans ce cas l'envoi est réalisé avec une $probaPledge$ non nulle. Le pseudo-code de cette modification se trouve dans l'annexe A.2.

4 Experimentation

Nous détaillerons dans un premier temps, le protocole expérimental puis les résultats obtenus.

4.1 Protocole expérimental

Toutes les expériences ont été réalisées avec le simulateur Peersim[15]. Dans nos simulations, nous avons considéré 15 nœuds parmi lesquels nous avons sélectionné arbitrairement un leader. La latence moyenne de transmission de message de 30 ms (selon une gaussienne ayant pour écart-type

10% de la moyenne). Nous avons stressé le système en démarrant périodiquement une nouvelle instance de Paxos selon un débit d'entrée donné et nous avons stoppé l'expérience une fois que 1000 instances se sont terminées. Une instance de Paxos démarre indépendamment des autres instances qui seraient susceptibles d'être toujours en cours d'exécution. Ainsi, ceci est équivalent à considérer que toute instance de Paxos est un ensemble à part entière, que tous ces ensembles sont en intersection totale et qu'ils ont tous le même nœud leader.

Nous avons étudié l'impact du débit d'entrée, et de la valeur des paramètres $probaPledge$ et $timeout$ de la primitive *send* modifiée (cf. section 3.2) sur les deux métriques suivantes :

— la latence moyenne pour exécuter une instance de Paxos
— le nombre moyen de messages pour exécuter une instance de Paxos (un agrégat contenant N messages compte juste pour un seul message).

Nous avons comparé nos performances à deux algorithmes :

— l'algorithme classique de Paxos, c-à-d. sans mécanisme d'agrégation et sans connaissance des ensembles multiples
— une version modifiée de Paxos ayant la connaissance des ensembles multiples, mais qui place systématiquement tous les messages dans un tampon. Les envois réels de messages sont ainsi cadencés par le déclenchement des timeout.

4.2 Résultats

Dans un premier temps, nous nous sommes intéressés à l'impact du débit et de la probabilité de pledger un message pour un timeout fixe.

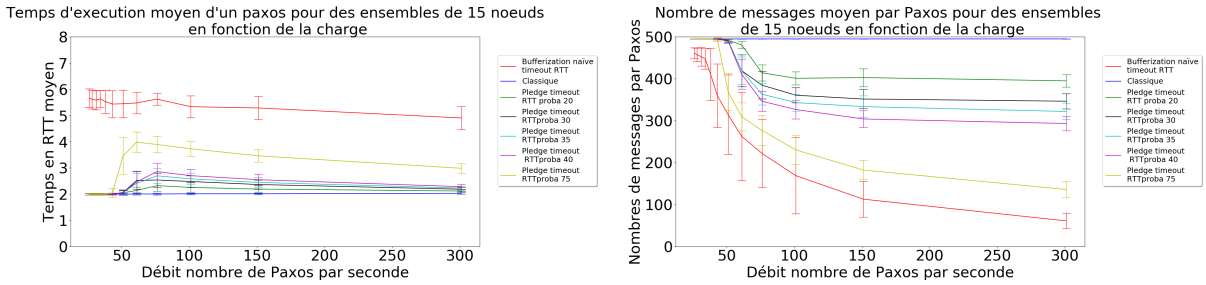

FIGURE 3 – Impact du débit et de la probabilité avec un timeout fixé à un RTT moyen

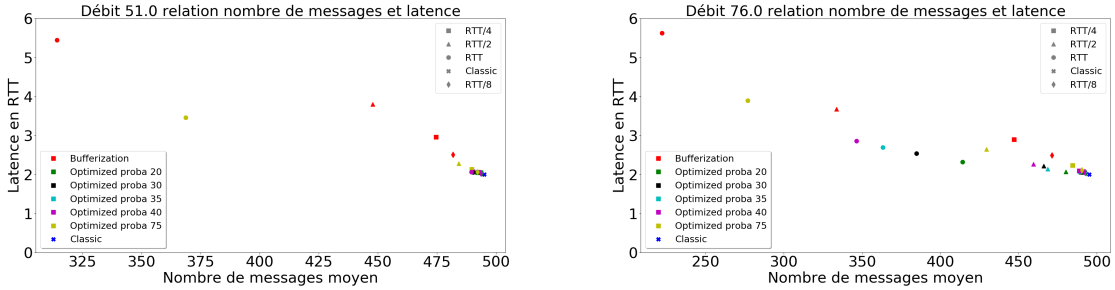

FIGURE 4 – Impact de la probabilité et du timeout pour un débit fixe (51 et 76)

Dans la figure 3, la courbe bleue représente la version de Paxos classique, la courbe rouge représente la version de Paxos avec la mise en tampon systématique et les autres courbes représentent les résultats de notre mécanisme d'agrégation pour différentes probabilités de pledge.

Nous pouvons remarquer que toutes les courbes relatives à notre mécanisme se trouvent bien entre les deux algorithmes extrêmes. Plus la probabilité de pledger augmente, plus la latence augmente mais plus le nombre de messages diminue. En effet, une probabilité importante implique

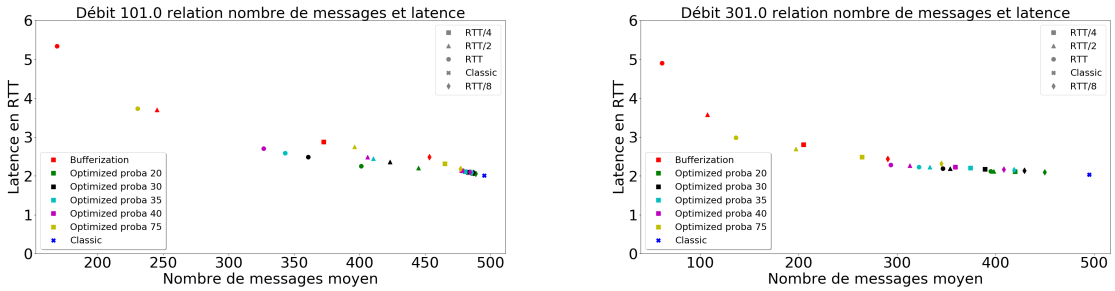

FIGURE 5 – Impact de la probabilité et du timeout pour un débit fixe (101 et 301)

d'avoir plus de chance de retarder un message pour l'agréger avec d'autres mais ce retard se répercute sur la latence. À fort débit, le potentiel d'émission de messages augmente considérablement sur l'ensemble des instances de Paxos ce qui permet d'agréger davantage et donc d'économiser des envois réels de messages.

À faible débit, il est très peu probable que des instances de Paxos s'exécutent au même moment. Prévoir les futures communications est donc très difficile et notre mécanisme s'active peu souvent. C'est la raison pour laquelle à faible débit le nombre moyen de messages est maximal.

Nous nous sommes ensuite penchés sur l'impact du timeout et de la probabilité de réaliser un pledge pour un débit donné. Cette représentation sous forme de front de Pareto permet de visualiser la meilleure configuration pour une application. Par exemple, si nous voulons réduire de 10% le nombre de messages, nous pouvons visualiser les répercussions des différentes configurations sur la latence et donc choisir la configuration la plus pertinente.

Sur la Figure 4, nous pouvons constater qu'à faible débit, les points sont concentrés autour du nombre maximal de messages par Paxos (sauf pour la mise en tampon systématique). Comme évoqué précédemment, l'activation de notre mécanisme dans ce cas de figure est rare. Avec l'augmentation du débit, nous pouvons visualiser qu'il est possible de réaliser davantage de pledges, permettant ainsi de réduire le coût en messages. Pour un fort débit et à condition de paramétrer correctement la primitive *send*, il est possible de réduire le nombre de messages sans pour autant dégrader la latence que l'on aurait dans la version classique de Paxos. En effet, on peut remarquer dans la Figure 5, qu'avec un débit de 301, la plupart de nos points sont alignés horizontalement avec le point du Paxos classique. Ainsi, en paramétrant la primitive *send* avec une probabilité de 40% et un timeout égal à un RTT moyen, il est possible de réduire de 40% le nombre de messages sans dégradation significative de la latence.

5 Conclusion

Nous avons proposé un mécanisme d'agrégation de message que l'on a appliqué à l'algorithme de Paxos. Notre mécanisme permet de trouver un compromis entre la dégradation de latence et le gain en nombre d'envois de messages. Nous exploitons la connaissance des différentes phases de l'algorithme pour savoir si un nœud s'adressera à un autre dans un futur proche lors d'un changement de phase. Nos expérimentations montrent que notre mécanisme peut nous faire économiser jusqu'à environ 40% de messages sans dégrader significativement la latence.

En ce qui concerne le choix de la probabilité, nous aimerions adapter sa valeur dynamiquement afin de fournir la meilleure configuration possible tout en prenant en compte l'état du réseau.

Nous avons choisi dans cet article de nous intéresser à l'algorithme de Paxos. Ses propriétés intéressantes et son utilisation très répandue en ont fait un très bon candidat. En réalité, ce mécanisme pourrait être utilisé par tous les algorithmes à phases. En effet, dès lors qu'un algorithme repose sur une succession de phases, il est possible d'anticiper les futures communications et donc d'agréger les messages plus pertinemment. Nous envisageons donc de rendre notre mécanisme suffisamment générique pour qu'il s'adapte à n'importe quel algorithme à phase. Nous gardons également comme objectif d'avoir une intrusivité nulle vis-à-vis des algorithmes que l'on multiplexe.

Bibliographie

1. Burrows (M.). – The chubby lock service for loosely-coupled distributed systems. – In *Proceedings of the 7th symposium on Operating systems design and implementation*, pp. 335–350, 2006.
2. Chand (S.), Liu (Y. A.) et Stoller (S. D.). – Formal verification of multi-paxos for distributed consensus. – In *International Symposium on Formal Methods*, pp. 119–136. Springer, 2016.
3. Chandra (T.), Griesemer (R.) et Redstone (J.). – Paxos made live - an engineering perspective. – In *PODC '07 : Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing*, 2006.
4. Corbett (J. C.), Dean (J.), Epstein (M.), Fikes (A.), Frost (C.), Furman (J. J.), Ghemawat (S.), Gubarev (A.), Heiser (C.), Hochschild (P.) et al. – Spanner : Google's globally distributed database. *ACM Transactions on Computer Systems (TOCS)*, vol. 31, n3, 2013, pp. 1–22.
5. Du (H.) et Hilaire (D. J. S.). – Multi-paxos : An implementation and evaluation. *Department of Computer Science and Engineering, University of Washington, Tech. Rep. UW-CSE-09-09-02*, 2009.
6. García-Pérez (Á.), Gotsman (A.), Meshman (Y.) et Sergey (I.). – Paxos consensus, deconstructed and abstracted. – In *European Symposium on Programming*, pp. 912–939. Springer, Cham, 2018.
7. Guerraoui (R.), Hurfinn (M.), Mostéfaoui (A.), Oliveira (R.), Raynal (M.) et Schiper (A.). – Consensus in asynchronous distributed systems : A concise guided tour. *In : Advances in Distributed Systems*, pp. 33–47. – Springer, 2000.
8. Junqueira (F.), Reed (B.) et Serafini (M.). – Zab : High-performance broadcast for primary-backup systems. *2011 IEEE/IFIP 41st International Conference on Dependable Systems & Networks (DSN)*, 2011, pp. 245–256.
9. Lamport (L.). – The part-time parliament, may 1998, 1998.
10. Lamport (L.). – Fast paxos. *Distributed Computing 19, pages79–103(2006)*, 2006.
11. Lamport (L.). – The mutual exclusion problem : part i—a theory of interprocess communication. *In : Concurrency : the Works of Leslie Lamport*, pp. 227–245. – 2019.
12. Lamport (L.). – The mutual exclusion problem : part ii—statement and solutions. *In : Concurrency : the Works of Leslie Lamport*, pp. 247–276. – 2019.
13. Mahmoud (H.), Nawab (F.), Pucher (A.), Agrawal (D.) et El Abbadi (A.). – Low-latency multi-datacenter databases using replicated commit. *Proceedings of the VLDB Endowment*, vol. 6, n9, 2013, pp. 661–672.
14. Malpani (N.), Welch (J. L.) et Vaidya (N.). – Leader election algorithms for mobile ad hoc networks. – In *DIALM '00 : Proceedings of the 4th international workshop on Discrete algorithms and methods for mobile computing and communications*, p. Pages 96–103, 2000.
15. Montresor (A.) et Jelasity (M.). – PeerSim : A scalable P2P simulator. – In *Proc. of the 9th Int. Conference on Peer-to-Peer (P2P'09)*, pp. 99–100, Seattle, WA, septembre 2009.
16. Nagle (J.). – Congestion control in IP/TCP internetworks. *RFC*, vol. 896, 1984, pp. 1–9.
17. Rao (J.), Shekita (E. J.) et Tata (S.). – Using paxos to build a scalable, consistent, and highly available datastore. *CoRR*, vol. abs/1103.2408, 2011.

A Annexes

A.1 Paxos

Paxos est un algorithme reposant sur un leader, tolérant aux fautes, permettant de résoudre le problème du consensus. L'algorithme original a été proposé par Lamport en 1998 [9]. Les communications sont asynchrones et la perte de messages tolérée. Du côté des acteurs, leur nombre est fixe et ils peuvent subir des fautes franches (*crash-recovery* possible). Une instance de Paxos débute lorsque le leader démarre un nouveau scrutin et se décompose en trois phases :

- Phase de préparation : le leader envoie des messages de type *prepare* à tous les participants accompagné d'un numéro de scrutin. Sur réception de ce message, les participants acceptent de rejoindre le scrutin si et seulement si il est plus récent qu'un éventuel scrutin auquel ils participent déjà.
- Phase d'acceptation : lorsque le leader apprend qu'une majorité de participants a accepté de rejoindre son scrutin, il envoie un message de type *accept* à tous les acteurs. Les autres participants vont alors (s'ils ne participent pas à un autre scrutin plus récent) retransmettre ce message via un message de type *accepted*.
- Phase de décision : lorsqu'un participant a reçu une majorité de message *accepted* pour un même numéro de ballot, il décide et transmet un message de type *decide* à tous les autres participants.

Paxos est un algorithme très utilisé notamment pour la gestion des réplicas de bases de données [3],[17]. En effet, il nécessite peu d'hypothèses et tolère la perte de messages. En revanche, la terminaison de l'algorithme n'est pas assurée (impossible d'arriver à une majorité, scrutins démarrés infiniment souvent).

A.2 Modification Paxos

Variables locales à l'acteur i

1 n_{set}
2 Number of sets to which p_i belongs

3 Val_i initially [($\perp$,set_1), ..., ($\perp$, $set_{n_{set}}$)]
4 Vector of size n_{set} representing the current value for the set set_k with $k\ in[1, n_{set}]$

5 $Ballot_i$ initially [(0, i, set_1), ..., (0, i, $set_{n_{set}}$)]
6 Number of the last ballot in which p_i took part for the set set_k with $k\ in[1, n_{set}]$

7 $AcceptNum_i$ initially [(0, i, set_1), ..., (0, i, $set_{n_{set}}$)]
8 Number of the ballot associated with the last value accepted by p_i for the set set_k with $k\ in[1, n_{set}]$

9 $AcceptVal_i$ initially [($perp$, set_1), ..., ($perp$, $set_{n_{set}}$)]
10 Last value accepted by p_i for the set set_k with $k\ in[1, n_{set}]$

11 $futur_{comm}$
12 set of waiting sets for a quorum

13 $proba_{pledge}$
14 probability to pledge a message

15 $timeout$
16 delay maximum of a message

Fonction additionnelle : checkPledge

checkPledge($dests$,set_k,$message$) : **for** *all members* p_k *belonging to* $dests$ **do**
  if p_k *belongs to another set of* $futur_{comm}$ *different from* set_k **then**
    Send($timeout$,$proba_{pledge}$, $message$) at $\{p_k\}$
  else
    Send(0, 0, $dest$, $message$) at $\{p_k\}$
end

Paxos Multi-set : phase de préparation

On p_i propose ((new_val, set_k)) :
$Ballot_i$ = <$Ballot_i.num++$, p_i, set_k >
checkPledge(set_k,set_k, <Prepare, $Ballot_i[set_k]$ >)
$futur_{comm} = futur_{comm} \cup \{set_k\}$

On p_i <Prepare, bal >from p_j :
if $Ballot_i[bal.set] \leq bal$ **then**
  $Ballot_i[bal.set] = bal$
  checkPledge($\{p_j\}$,set_k, <Ack, bal, $AcceptNum_i[bal.set]$, $AcceptVal_i[bal.set]$ >)
end

Paxos Multi-set : phase d'acceptation

On p_i <Ack, $Ballot$, b, val >from n-f actors :
$futur_{comm} = futur_{comm} \setminus \{set_k\}$
if *all vals at* ($\perp$,$Ballot.set$) **then**
  $Val_i[Ballot.set] = (new\_val, Ballot.set)$

else
  $Val_i[Ballot.set]$ = the value associated with the biggest b for $Ballot.set$

checkPledge($Ballot.set$,$Ballot.set$, <Accept, $Ballot$, $Val_i[Ballot.set]$ >)
$futur_{comm} = futur_{comm} \cup \{set_k\}$

On p_i <Accept, b, v, >from p_j :
if $b \geq Ballot_i[b.set]$ **then**
  $Ballot_i[b.set] = b$
  $AcceptNum_i[b.set] = b$
  $AcceptVal_i[b.set] = v$
  checkPledge($b.set$,$b.set$, <Accepted, b, v >)
end

Paxos Multi-set : phase de décision

On p_i <Accepted, b, v, >from $n - f$ processus :
Decide v
$futur_{comm} = futur_{comm} \setminus \{set_k\}$
checkPledge($b.set$,$b.set$, <Decide, val >)

On p_i <Decide, v > :
$futur_{comm} = futur_{comm} \setminus \{set_k\}$
checkPledge($v.set$,$v.set$, <Decide, v >)