Sergio ANAGBLA Léo CHECK

Projet ARA: Multi-Paxos sur Peersim

M2 Informatique - Spécialité Systèmes et Applications Répartis

Sorbonne Université

Année universitaire 2020-2021

I. Contexte

Une architecture distribuée est un réseau constitué d'un ensemble de ressources autonomes et communicants ne se trouvant pas au même endroit. Ce type d'architecture ne possède ni mémoire partagée, ni horloge globale permettant d'assurer une cohérence globale au sein du système. Cette cohérence est d'autant plus difficile à maintenir en raison des défaillances liées à fiabilité du réseau et aux pannes des unités de calcul du système.

Néanmoins ce type d'architecture présente de gros avantages dans plusieurs domaines comme le traitement des gros volumes de données (Répartition de charge, résistance aux pannes, disponibilité des données). Afin de faire face à ces difficultés, des protocoles de communication ont été mis en place.

Paxos est un protocole permettant de résoudre le problème de consensus au sein d'un réseau de nœuds faillibles. Ainsi les différents nœuds d'un système réparti peuvent avoir une vision d'ensemble cohérente. Dans le cadre de notre projet, nous avons implémenté ce protocole et mené plusieurs études expérimentales via le simulateur PeerSim qui vous seront présentées dans ce rapport.

II. Étude expérimentale 1 : itérations d'élection

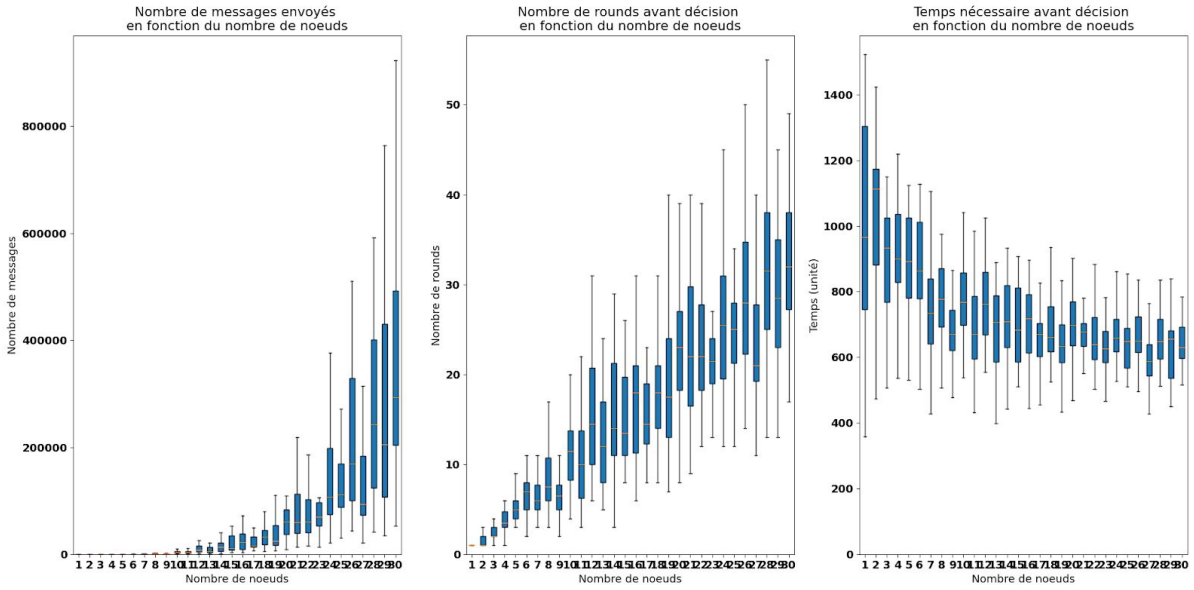
Scénario considéré : Tous les nœuds (en tant que Proposer) tentent d'élire un leader (via la requête findLeader) qui sera utilisé dans les itérations suivantes. Aucune faute ne sera injectée.

Critère de validation : Tous les nœuds doivent avoir la même valeur de leader à la fin du scénario.

Métriques considérées dans cette étude :

- le nombre de rounds nécessaires pour atteindre une majorité, c'est à dire pour que le consensus soit atteint
- le nombre de messages émis
- le temps de convergence: le temps nécessaire au protocole pour que tous les nœuds reconnaissent le même leader

[Expérience 1] Paramètre: le nombre de nœuds. Backoff désactivé, valeur du round initial égale à l'identifiant du noeud Proposer

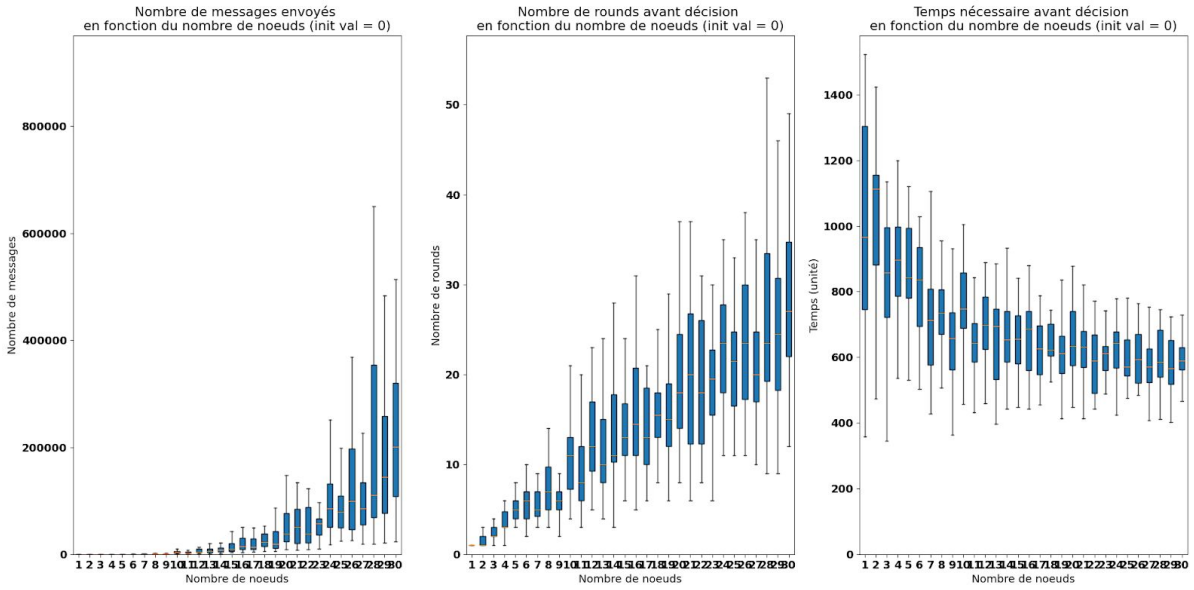


Dans cette expérience, nous avons testé l'élection sur un nombre de nœuds allant de 1 à 30. Pour chaque taille du système, 30 tests ont été réalisés (avec une graine aléatoire allant de 0 à 29). Au lieu de tracer une courbe des moyennes, nous avons choisi de tracer des boîtes à moustaches, nous permettant d'interpréter plus facilement les données.

Comme nous pouvions nous y attendre, le nombre de messages augmente exponentiellement avec la taille du système. Le nombre de rounds nécessaire pour l'élection semble évoluer de façon linéaire. Quant à la durée de l'élection, elle semble se stabiliser autour de 650 unités de temps lorsque le système atteint 10 nœuds, ce qui peut s'expliquer par la distribution de la charge de traitement.

Quelque soit la métrique, nous remarquons une grande disparité des valeurs dûe aux conflits des processus lors de l'élection. Il y a en effet, une part de "chance" pour qu'une élection soit validée par tous les nœuds.

[Expérience 2] Paramètre: le nombre de nœuds. Backoff désactivé, valeur du round initial égale à 0 (Paxos original)

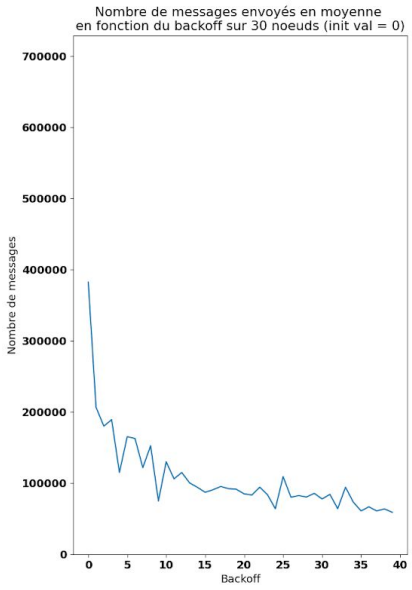


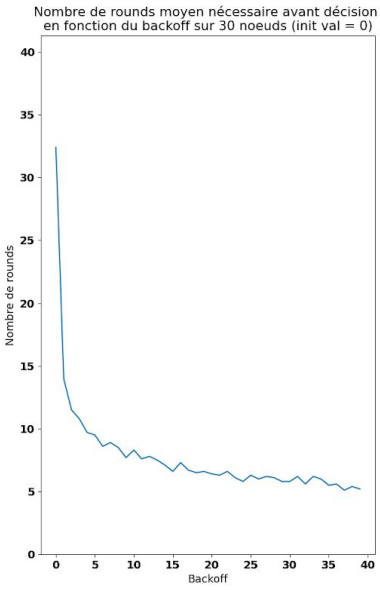
Les conditions de cette expérience sont identiques à la précédente. En comparant les données, nous pouvons affirmer qu'il y a un réel intérêt à proposer pour valeur initiale 0 comme dans le Paxos original. Il est clair que, en comparant avec la première expérience, nous avons réduit la croissance exponentielle en message envoyé. De plus, la disparité des données est bien moins grande (notamment les messages).

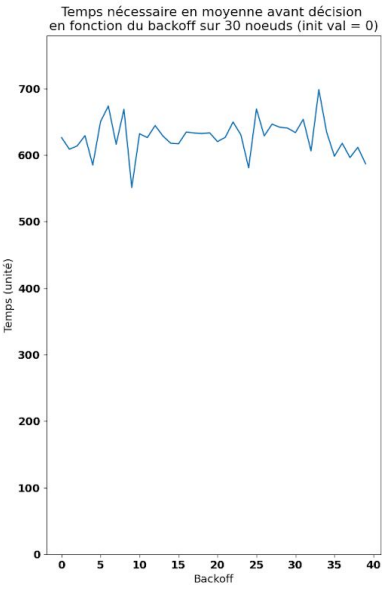
Cependant, nous n'avons pas un gain significatif en complexité de temps. Elle semble se stabiliser autour de 600 unités de temps au lieu de 650 (environ -8%). Quant au nombre de rounds, il semble ne pas être affecté. Dans la suite, nous fixons la valeur initiale proposée à 0.

Pour réduire davantage la complexité des 3 métriques étudiées, nous introduisons un mécanisme de backoff. Lorsqu'un nœud est entré en conflit pendant son élection, il attend pendant une certaine période avant de réitérer sa proposition. À chaque rejet, il augmente ce temps d'attente par la valeur du backoff.

[Expérience 3] Paramètre : valeur du backoff. Nombre de noeud fixé à 30, valeur du round initial égale à 0 (Paxos original)





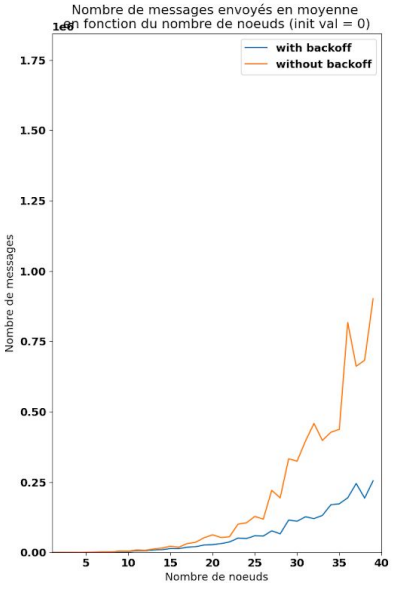


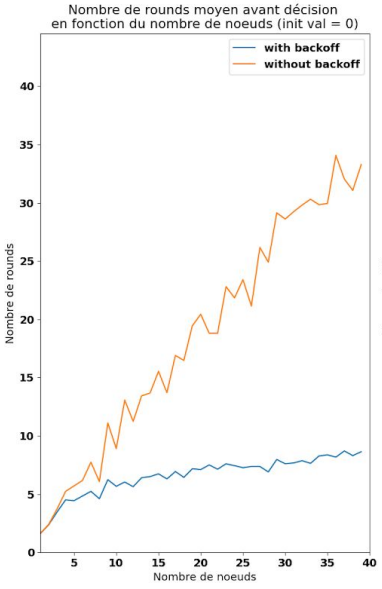
Pour tester l'impact du backoff, nous avons décidé de fixer le nombre de nœuds à 30, la taille maximale testée jusqu'à présent. Nous faisons ensuite varier la valeur du backoff de 0 à 39. De la même manière que les deux premières expériences, nous testons 10 graines aléatoires pour chaque valeur du backoff.

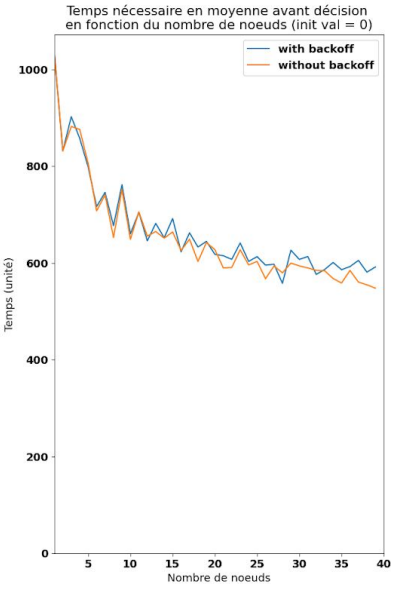
Le backoff se montre efficace dès ses premières valeurs. En effet, lorsqu'il vaut 10, nous réduisons le nombre de rounds par 4, et le nombre de messages est réduit de 75% en moyenne. Cependant le temps reste relativement constant quelle que soit la valeur du backoff car les délais du backoff contre-balancent les messages économisés.

Au-delà de 10, le gain en performance apporté par le mécanisme du backoff reste minime. Nous nous fixons donc à cette valeur.

[Expérience 4] Paramètre : le nombre de nœuds. Backoff activé fixé à 10, valeur du round initial égale à 0 (Paxos original)







Dans les mêmes conditions que les deux premières expériences (taille et graine aléatoires), nous observons qu'un backoff de 10 permet de réduire considérablement la croissance des 2 premières métriques étudiées. Le temps d'élection reste approximativement égal, bien qu'on puisse observer un temps d'élection légèrement supérieur lorsque le backoff est activé, dû aux délais.

Finalement, nous obtenons une configuration pour l'élection relativement optimisée pour cette tâche : backoff fixée à 10 et valeur initiale égale à 0.

III. Étude expérimentale 2 : Multi-Paxos séquentiel

Scénario considéré : En dehors d'une itération d'élection, le leader soumet séquentiellement des requêtes. Le leader attend qu'une itération soit validée (en tant que Learner) pour passer à la suivante. En fonction des résultats que vous avez eus dans l'étude précédente, vous prendrez la meilleure configuration en ce qui concerne les paramètres de backoff et de round initial dans les itérations findLeader.

Critère de validation : La propriété de sûreté décrite à la section 2 du sujet doit être vérifiée.

Métriques considérées dans cette étude :

- le débit moyen de requêtes applicatives validées qui est le rapport entre le nombre total de requêtes hors élection validées et le temps d'expérience
- la latence moyenne qui est le temps moyen pour qu'une requête applicative (hors élection) soit validée
- le nombre de messages émis (à l'exception des messages Ping et Pong)

D'après I, la meilleure configuration pour l'élection est : backoff = 10, valeur initiale = 0. Nous l'utiliserons dans la suite de nos expériences.

Pour l'implémentation du Multi-Paxos, nous nous sommes permis d'utiliser les messages Ping pour les soumissions, et les Pong pour les ack des requêtes entre les nœuds du système et le nœud leader. Pour simplifier, les requêtes seront représentées par un identifiant unique.

Lorsqu'un nœud souhaite soumettre une requête de sa pile **queueRequest** à son leader, il crée un message Ping contenant la requête et l'envoie au leader. De plus, il s'envoie à lui-même un message Timeout_Ping, contenant lui aussi la requête, jouant le rôle du timeout. On suppose donc que les nœuds ont connaissance du temps maximal que prend un message pour réaliser l'aller-retour entre eux-mêmes et le leader.

À la réception d'un Ping, le leader ajoute l'identifiant de la requête à sa pile de requêtes **queueRequest**, et répond au nœud source avec un message Pong contenant la requête nouvellement ajoutée.

Ainsi, si un nœud reçoit une réponse Pong du leader, il place la requête dans une pile **requestOk** et passe à la suivante dans sa pile. Mais, si ce même nœud reçoit son

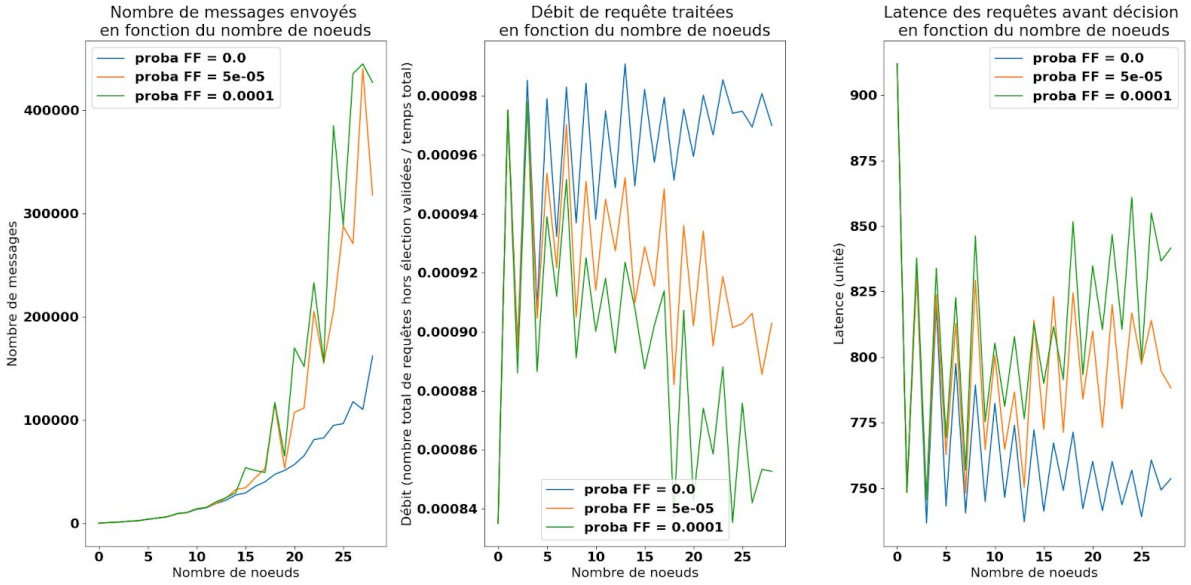
message Timeout_Ping avant le Pong, il considérera ce dernier comme défaillant et lancera une nouvelle élection.

Il est important de souligner que si le leader courant tombe en panne, toutes les requêtes qu'il a reçues (contenues dans **queueRequest**) seront perdues.

Nous simulons par ailleurs une réception "infinie" des requêtes par tous les nœuds tant que l'expérience n'est pas terminée. À chaque fois qu'un nœud reçoit un ack du leader, il crée artificiellement une nouvelle requête dans sa pile de requêtes **queueRequest**. Nous limitons les simulations à une date maximale de 5 000.

Dans cette étude expérimentale, les pannes transitoires nous ont posé beaucoup de problèmes. Nous avons donc décidé de nous focaliser sur les fautes franches.

[Expérience 5] Paramètres : nombre de nœuds et probabilité d'avoir une faute franche (FF).



Dans cette expérience, nous testons l'impact d'une faute franche dans le système. Elle se produit, selon une probabilité donnée, après l'envoie d'un message d'un nœud. Nous évaluons 3 cas : aucune faute, 0.00005% et 0.0001%. Nous avons besoin que ces probabilités restent très faibles pour que les fautes n'arrivent pas trop rapidement dans la simulation. Nous testons ces 3 probabilités sur un système de taille allant de 2 à 30, où, pour chaque taille, nous y testons 30 graines aléatoires.

On peut s'apercevoir que le nombre de messages en moyenne (hors ping et pong) augmente considérablement, dû aux pannes des leaders menant à réaliser des élections successivement, chargeant le réseau. En conséquence, le débit moyen de requêtes traitées par le Multi-Paxos se voit ralenti, et la latence moyenne du traitement d'une requête augmente.

IV. Étude expérimentale 3 : Réduire le coût en messages

A. Bufferisation

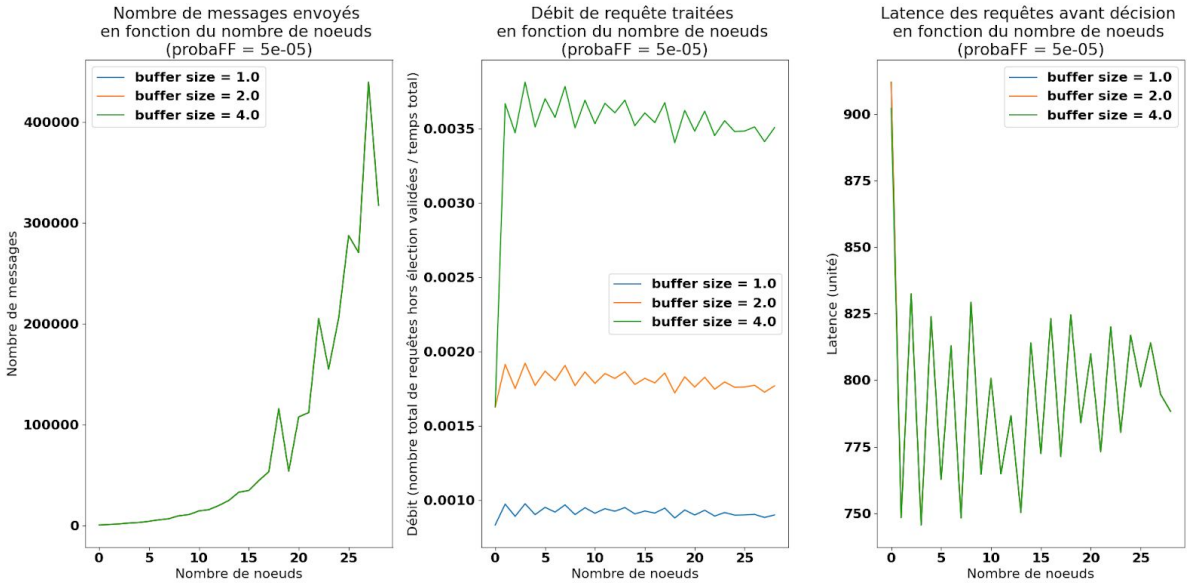
Afin de réduire le coût en messages, nous décidons de produire des messages avec des tailles de contenus applicatifs plus grands. Nous réalisons une bufferisation de **buffer_size** requêtes lors d'une itération de Multi-Paxos. Ainsi, le leader soumet séquentiellement **buffer_size** requête(s) dans le protocole pour décider les **buffer_size** prochain(s) index de l'historique des nœuds. Cela permettrait, en théorie, de réduire les messages envoyés dans le système lors du Multi-Paxos.

Nous n'appliquons pas la bufferisation dans les messages ping et pong servant de soumission de requêtes entre les nœuds et le leader afin de limiter la perte de requête lorsque le leader tombe en panne. Ils contiendront donc une unique requête.

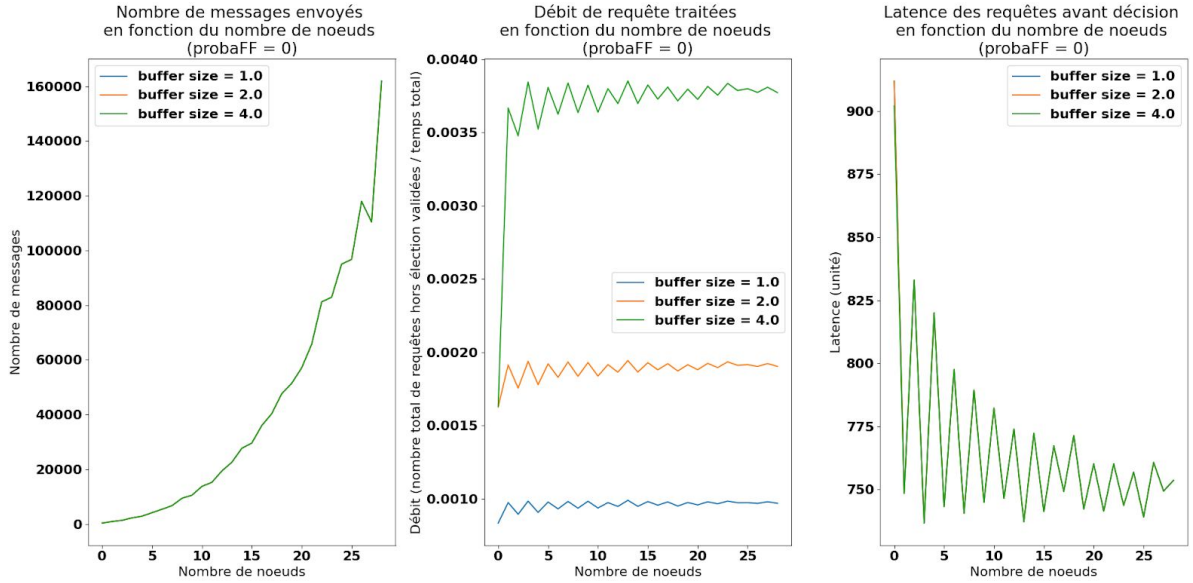
Nous identifierons l'impact des paramètres suivants sur le nombre de messages, le débit et la latence des traitements des requêtes :

- La taille du système N.
- La taille de bufferisation buffer_size.
- Système fautif ou non.

[Expérience 6] Paramètre : nombre de nœuds et taille du buffer. Probabilité d'avoir une faute franche fixée à 0.00005% (FF).



[Expérience 7] Paramètre : nombre de nœuds et taille du buffer. Système sans faute.



Malheureusement, cette solution de bufferisation a uniquement influencé le débit de requête traitées puisqu'au final, seul le contenu des messages échangés dans le Multi-Paxos ont été modifiés. De ce fait, en lançant les expériences avec les mêmes paramètres que dans la partie II, il s'y produit les mêmes fautes (et donc, les mêmes élections) et la même latence au niveau du traitement des messages.

Cette solution n'a donc d'intérêt que lorsque nous visons à améliorer le débit des décisions du protocole Multi-Paxos.

B. Élection cyclique du leader

Nous nous sommes aperçus dans la partie II que le Multi-Paxos devenait très coûteux en message lorsque nous introduisions les fautes franches. En effet, grâce à nos données de la partie I, la complexité d'une élection en messages augmente de façon exponentielle avec la taille du système. Ainsi, réaliser plusieurs élections en cas de pannes successives des leaders surcharge le réseau.

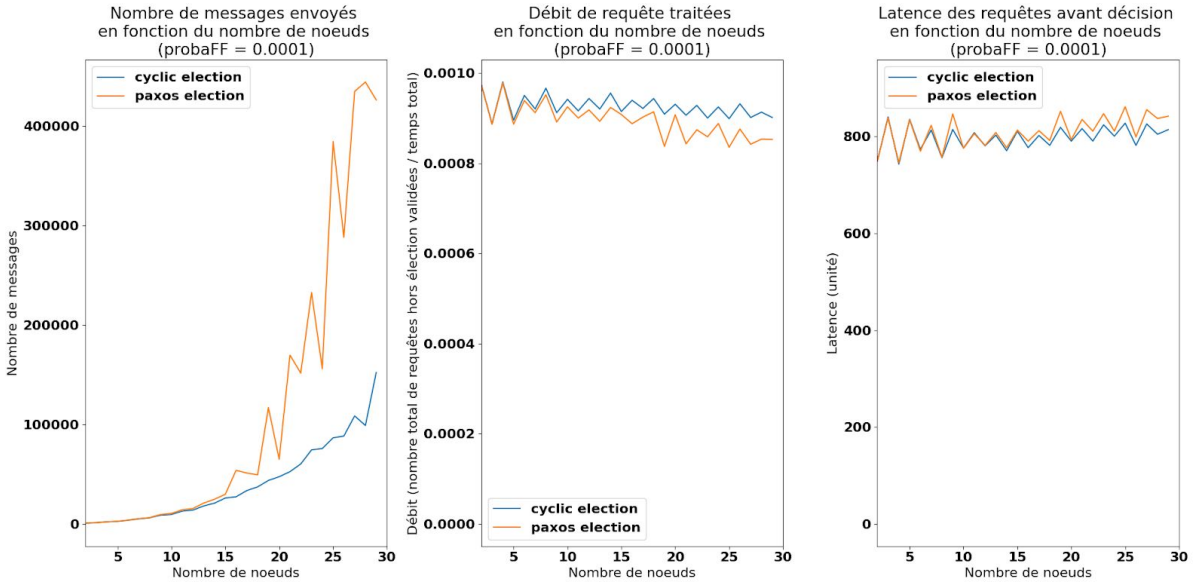
Une solution simple serait alors de réaliser non pas une élection par Paxos, mais une simple élection cyclique lors d'une détection de panne. Cela rajoute une hypothèse supplémentaire sur les nœuds : ils doivent connaître l'ensemble des identifiants des nœuds du système.

Ainsi, lorsqu'un nœud détecte la panne de son leader d'identifiant **id_leader**, il élit immédiatement le nœud d'identifiant **id_leader**+1 modulo N (où N est la taille du système). Si cette solution peut être difficile à mettre en place dans des conditions réelles, où il existe une grande dynamicité dans les systèmes distribués, elle est facile à mettre en place dans Peersim.

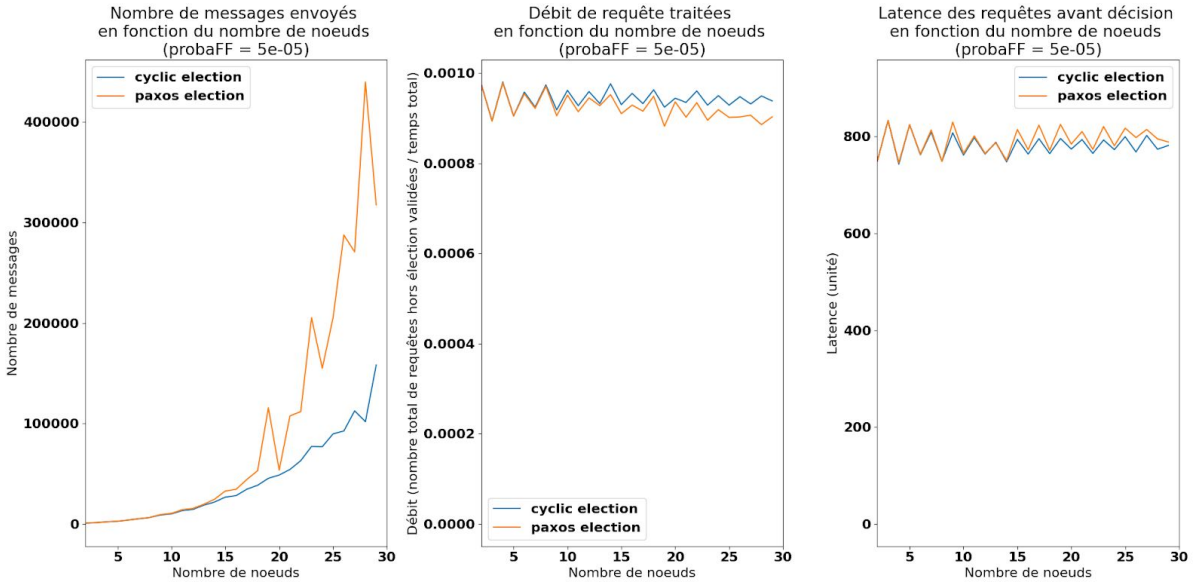
Nous identifierons l'impact des paramètres suivants sur le nombre de messages, le débit et la latence de traitement des requêtes :

- Nombre de nœuds du système N,
- Mode d'élection (paxos ou cyclique),
- Probabilité de panne franche (excepté 0, sinon l'élection cyclique ne se produira jamais)

[Expérience 8] Paramètres : nombre de nœuds et mode d'élection. Probabilité d'avoir une faute franche fixée à 0.0001% (FF).



[Expérience 9] Paramètres : nombre de nœuds et mode d'élection. Probabilité d'avoir une faute franche fixée à 0.00005% (FF).



Dans l'expérience 8 (resp. 9), nous exécutons le Multi-Paxos sur un système de taille allant de 2 à 30, où pour chaque taille nous y testons 30 graines aléatoires, avec une probabilité de faute franche de 0.0001% (resp. 0.00005%).

L'élection cyclique se montre très efficace lorsque le système grandit en taille. Elle réduit considérablement le nombre de messages, sans pour autant dégrader de façon significative les 2 autres métriques.

Cependant, dans un système de taille inférieure à 15, nous n'avons pas de réel gain en termes de messages. En effet, afin que le protocole de Paxos fonctionne, nous avons dû limiter le nombre de pannes de sorte à ce qu'au moins une majorité des nœuds soient corrects. Couplé à la faible probabilité de faute franche, il se produit moins de pannes, en particulier celle du leader. Ainsi, nous ne réalisons pas de réélection. Il faudrait alors directement optimiser le protocole de Paxos.