Uma Estratégia de Difusão Agressiva para Aumentar a Vazão do Algoritmo de Consenso HyperPaxos

Djenifer R. Pereira, Fernando M. Kiotheka, Elias P. Duarte Jr.

Departamento de Informática, Universidade Federal do Paraná (UFPR)
Caixa Postal 19018 – 81531-990 – Curitiba/PR

{drpereira,fmkiotheka,elias}@inf.ufpr.br

Resumo. *Algoritmos de consenso distribuído são essenciais para sistemas de armazenamento, bancos de dados, controle de acesso e orquestração de aplicações em nuvem. Este trabalho apresenta uma estratégia para melhorar a vazão do algoritmo HyperPaxos em termos de decisões por segundo. O algoritmo HyperPaxos é uma versão hierárquica de um dos principais algoritmos de consenso, o Paxos. O HyperPaxos é baseado na topologia virtual hierárquica vCube, que apresenta diversas propriedades logarítmicas. Os* acceptors *são organizados em* clusters *e os* proposers *executam as duas fases do Paxos escolhendo um* acceptor *dito difusor. O difusor é responsável por retransmitir as mensagens para os demais* acceptors *sobre o vCube. Neste trabalho, propomos que o difusor adote uma estratégia de difusão agressiva para transmitir, de uma só vez, as mensagens para uma maioria de* acceptors *paralelamente. A estratégia proposta foi implementada e comparada à versão original. Resultados obtidos mostram o desempenho superior da estratégia proposta em todos os cenários testados.*

1. Introdução

O acordo distribuído, ou consenso, é possivelmente o problema central da área de sistemas distribuídos. Informalmente, no problema do consenso, os processos propõem valores e, ao final, todos os processos decidem por um mesmo valor entre os propostos. Exemplos de aplicações diversas que utilizam consenso incluem bancos de dados distribuídos como o Google Spanner [Brewer 2017] e sistemas para orquestração de aplicações em nuvem como o Kubernetes [Red Hat 2019].

Um dos principais algoritmos que resolve o problema do consenso é o Paxos [Lamport 1998, Lamport 2001, Renesse and Altinbuken 2015]. No Paxos, os processos assumem papéis, que podem ser: *proposer*, *acceptor* e *learner*. Um *proposer* propõe valores, os *acceptors* decidem por um valor e os *learners* aprendem o valor decidido. O processo de decisão ocorre em duas fases, descritas a seguir de maneira bastante resumida. Na primeira fase, o *proposer* valida um número de proposta com os *acceptors* para propor um valor. Com um número de proposta, na segunda fase, o *proposer* faz uma proposta com valor para os *acceptors*. O consenso é atingido quando uma maioria de *acceptors* aceita um mesmo valor.

Devido à importância do Paxos, e ao fato de ser um algoritmo custoso, diversas variantes têm sido desenvolvidas [Regis and Mendizabal 2022]. Em particular, o Ring Paxos [Jalili Marandi et al. 2017] é uma versão que utiliza uma topologia em anel com a proposta de aumentar a vazão em termos do número de decisões por segundo. No Ring

Paxos os processos são organizados em anel, de forma que cada processo se comunica com apenas um outro processo, até atingir uma maioria. Porém, a estrutura em anel leva a um potencial aumento na latência do algoritmo. Neste contexto, foi proposto o algoritmo HyperPaxos [Kiotheka et al. 2023], que se baseia na topologia escalável vCube [Duarte Jr et al. 2014] que apresenta diversas propriedades logarítmicas.

No HyperPaxos, a lógica original do Paxos é mantida, alterando somente a comunicação entre os papéis para criar a topologia virtual do vCube. Os *proposers* fazem as requisições para um *acceptor* denominado difusor e esse se torna responsável em repassar para os demais *acceptors* as requisições feitas pelo *proposer* usando a topologia do vCube. As mensagens são enviadas do maior ao menor *cluster* até atingir uma maioria de *acceptors*. Conforme a árvore de difusão é percorrida, as respostas dos *acceptors* vão sendo concatenadas junto da mensagem original. As respostas são encaminhadas para o *acceptor* responsável pela difusão e caso receba uma maioria de respostas confirmando a requisição, ele reencaminha as respostas para o *proposer*. Caso a maioria não seja atingida, o *acceptor* difusor prossegue para o próximo *cluster*.

Em [Kiotheka et al. 2023], a libHyperPaxos, uma implementação do HyperPaxos baseada na libPaxos [Primi and Sciascia 2013], é comparada com a implementação U-Ring Paxos [Benz 2017] em termos de decisão por segundo. Para valores com tamanho menor que 1024 bytes, a libHyperPaxos tem melhores resultados que o U-Ring Paxos. No entanto, ao investigar cenários com vários números de *acceptors*, o desempenho da libHyperPaxos em termos da vazão, isto é, o número de decisões por segundo, apresenta grande variação. Assim, neste trabalho, propomos uma estratégia agressiva de difusão em que o *acceptor* difusor transmite, de uma só vez, as mensagens para uma maioria de *acceptors*. A estratégia proposta foi implementada e resultados de comparação mostram seu desempenho superior em todos os cenários testados.

O restante deste trabalho está organizado da seguinte forma. A Seção 2 apresenta o algoritmo HyperPaxos, explicando como é feita a organização dos papéis no algoritmo e como é feita a comunicação entre eles. Na Seção 3 descrevemos as alterações propostas para o HyperPaxos. Em seguida, na Seção 4 apresentamos a implementação e os resultados obtidos da comparação com a libHyperPaxos. E por fim, encerramos o trabalho com as conclusões na Seção 5.

2. O Algoritmo HyperPaxos: Uma Visão Geral

O HyperPaxos é um algoritmo de consenso distribuído tolerante a falhas. O algoritmo assume modelo temporal parcialmente síncrono com GST (*Global Stabilization Time*) [Dwork et al. 1988] e o modelo de falhas *crash-recovery* [Hurfin et al. 1998]. Além disso, os enlaces de comunicação são perfeitos [Cachin et al. 2011].

O HyperPaxos define para os processos os mesmos papéis do algoritmo Paxos. Apenas a comunicação entre os papéis é alterada, de forma que os *acceptors* formam uma topologia hierárquica vCube de n_a *acceptors*. Como os *proposers* estão fora da topologia, para realizar as fases 1 e 2, eles se comunicam usando um *acceptor* intermediador que realiza a transmissão sobre o vCube para os demais *acceptors*.

A topologia vCube vem do algoritmo de detecção de falhas vCube, que foi inicialmente proposto em [Duarte Jr and Nanya 1998], no contexto de diagnóstico distribuído.

Nele, os processos são organizados em *clusters* de forma hierárquica, formando um hipercubo quando o número de processos é uma potência de dois e todos os processos estão corretos. O hipercubo apresenta simetria e diâmetro logarítmico. Na falta ou na falha de processos, essa topologia se reorganiza mantendo as propriedades logarítmicas.

No HyperPaxos, os *acceptors* formam uma topologia do vCube, e cada *acceptor* se comunica com no máximo $\lceil \log_2 n \rceil$ *clusters* de *acceptors* utilizando um algoritmo de difusão inspirado em [Rodrigues et al. 2018]. Os *clusters* são definidos pela função $c(i, s)$ que retorna a sequência de *acceptors* do *cluster* s para o *acceptor* i. Uma definição para a função $c(i, s)$ onde $\oplus$ denota a operação de ou exclusivo é dada por

$$c(i, s) = (i \oplus 2^{s-1}, c(i \oplus 2^{s-1}, 1), c(i \oplus 2^{s-1}, 2), \ldots, c(i \oplus 2^{s-1}, s-1)).$$

A Tabela 1 lista o $c(i, s)$ de todos os *acceptors* em um sistema com número de *acceptors* $n_a = 8$. Cada linha indica o número do *cluster* s e cada coluna indica o *acceptor* i de $c(i, s)$. Cada *acceptor* se comunica com o primeiro *acceptor* correto de cada *cluster*. Assim, num sistema sem falha e quando o número de *acceptors* for uma potência de dois, a topologia forma um hipercubo perfeito.

s	$c(0,s)$	$c(1,s)$	$c(2,s)$	$c(3,s)$	$c(4,s)$	$c(5,s)$	$c(6,s)$	$c(7,s)$
1	(1)	(0)	(3)	(2)	(5)	(4)	(7)	(6)
2	(2,3)	(3,2)	(0,1)	(1,0)	(6,7)	(7,6)	(4,5)	(5,4)
3	(4,5,6,7)	(5,4,7,6)	(6,7,4,5)	(7,6,5,4)	(0,1,2,3)	(1,0,3,2)	(2,3,0,1)	(3,2,1,0)

Tabela 1. Valores de $c(i, s)$ para 8 *acceptors*.

Na fase 1, o *proposer* envia um pedido de preparação para um *acceptor* dito difusor, que será responsável por difundir a mensagem no vCube. O difusor faz a difusão para os seus *clusters*, do maior para o menor, um por vez. Isto é, o *acceptor* difusor i envia uma mensagem para o primeiro *acceptor* correto j de $c(i, s)$, começando com $s = \lceil \log_2 n_a \rceil$, aguardando as respostas desse *cluster*. A resposta do próprio *acceptor* difusor vai junto do pedido de preparação que ele difunde para seus *clusters*.

Cada *acceptor* i, ao receber um pedido de preparação de um *acceptor*, encaminha o pedido para os todos os primeiros *acceptors* corretos dos seus *clusters* de $c(i, 1)$ até $c(i, s-1)$, junto da sua própria resposta ao pedido de preparação. Caso o *acceptor* não tenha a quem transmitir a mensagem, isto é, é uma folha na árvore de difusão, então esse *acceptor* encaminha uma mensagem com todas as respostas de volta ao *acceptor* difusor.

Quando o *acceptor* difusor recebe as respostas de todo um *cluster* para o pedido de preparação, ele avalia se existe uma maioria de promessas que validam o número de proposta do *proposer*. Em caso afirmativo, o *acceptor* encaminha as respostas recebidas de volta para o *proposer*. Se não, o *acceptor* deve continuar a difundir a mensagem para um *cluster* de tamanho s menor. Caso o *acceptor* esgote todos os *clusters*, sem validação do número de proposta, o *proposer* é instruído a reiniciar a fase 1 com um número de proposta maior.

No melhor caso, a difusão para o maior *cluster* é suficiente, particularmente quando o número de *acceptors* é uma potência de 2 e todos estão corretos, pois neste caso, o maior *cluster* tem tamanho $n_a/2$. Como a difusão é inicializada com a resposta do difusor para o pedido de preparação, são $n_a/2 + 1$ respostas. Assim, se todas as res-

postas forem promessas que validam o número de proposta, a maioria necessária para a validação é atingida.

A execução da fase 2 é semelhante à fase 1. O *proposer* envia a proposta com valor para outro *acceptor*, que será responsável por difundir a mensagem. A difusão ocorre da mesma maneira que na fase 1, de *cluster* em *cluster*. Ao atingir a maioria, o difusor encaminha a decisão para todos os *proposers* e todos os *learners*. Caso a maioria não seja atingida, o *proposer* é instruído a iniciar uma nova fase 1 com um novo número de proposta.

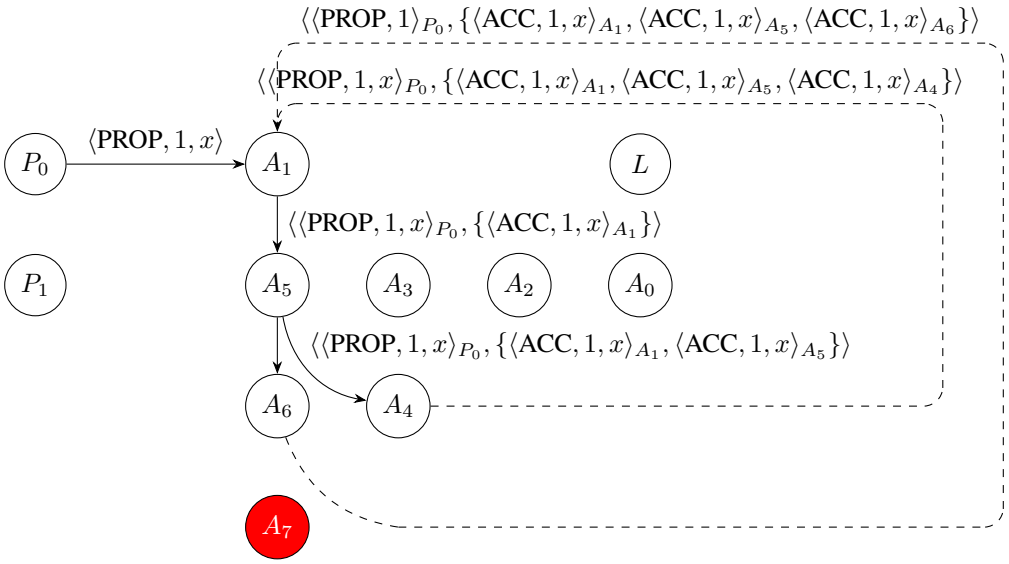

Figura 1. Difusão na fase 2 do maior *cluster* do *acceptor* 1 em um sistema com $n_a = 8$ e *acceptor* 7 falho.

A Figura 1 mostra um cenário com 8 *acceptors* onde o *proposer* 0 executa a fase 2 e o *acceptor* 7 falhou. O *proposer* escolhe o *acceptor* 1 para a difusão e envia sua proposta PROP com número 1 e valor x. O *acceptor* 1, ao receber a proposta, a aceita e a encaminha com sua resposta ACC para o *acceptor* 5 do seu maior *cluster*, o *cluster* 3. O *acceptor* 5 faz o mesmo, aceitando a proposta e a encaminhando com as respostas para o *acceptor* 4 do *cluster* 1 e o *acceptor* 6 do *cluster* 2, já que o *acceptor* 7 está falho. Como os *acceptors* 4 e 6 são folhas na árvore de difusão, eles retornam as respostas para o *acceptor* difusor 1. Ao receber todas as respostas do *cluster*, o *acceptor* 1 verifica se tem uma maioria de aceites para a proposta.

Como neste caso não há maioria, o *acceptor* difusor 1 prossegue para o seu *cluster* 2, enviando a proposta para o *acceptor* 3 como mostra a Figura 2. O *acceptor* 3 aceita a proposta e repassa para o *acceptor* 2 do seu *cluster* 1. O *acceptor* 2 é uma folha na árvore, então retorna a resposta para o *acceptor* difusor 1. Agora o *acceptor* 1 conseguiu a maioria de aceites atingindo o consenso e pode enviar as respostas para todos os *proposers* e todos os *learners*.

3. Uma Estratégia de Difusão Agressiva para o HyperPaxos

Na versão original do HyperPaxos, a difusão para o primeiro *cluster* é suficiente apenas no caso do número de *acceptors* ser uma potência de 2, e o sistema não apresentar falhas.

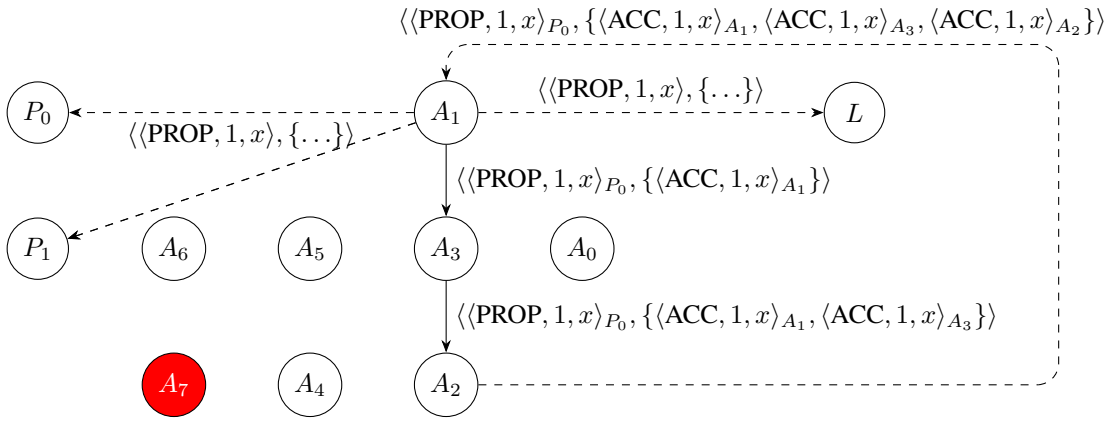

Figura 2. Difusão na fase 2 do segundo maior *cluster* do *acceptor* 1 em um sistema com $n_a = 8$ e *acceptor* 7 falho.

Caso contrário, a primeira difusão pode não ser suficiente para atingir a maioria necessária para a fase 1 e 2. Isso é exemplificado nas Figuras 1 e 2, em um sistema com 8 *acceptors* e o *acceptor* 7 falho, a primeira difusão feita pelo *acceptor* difusor 1 é para o *cluster* 3, que não forma uma maioria. O mesmo ocorre se a difusão fosse iniciada nos *acceptors* 0, 2 e 3. Nesse caso, a difusão de *cluster* em *cluster* aguarda o retorno do *cluster* 3 antes de transmitir para o *cluster* 2, aumentando a latência.

Assim, propomos uma estratégia agressiva para o difusor enviar, de uma só vez, a mensagem para diversos *clusters* de *acceptors* ao mesmo tempo, paralelamente. Isso é possível pois o HyperPaxos depende de um detector de falhas, que informa quais processos estão falhos. Considerando quantos *acceptors* estão falhos em cada *cluster*, a estratégia seleciona os *clusters* necessários para formar uma maioria. Duas variações são propostas. A primeira variação busca os maiores *clusters* que formam a maioria, utilizando a menor quantidade possível de *clusters*. Desta forma, nas Figuras 1 e 2, o difusor 1 faz a difusão em paralelo para os *clusters* 2 e 3. Como o *cluster* 3 possui 3 *acceptors* (4, 5 e 6), e o *cluster* 2 possui dois *acceptors* (2 e 3) e a difusão começa pelo *acceptor* 1, então 6 *acceptors* são atingidos: 1, 2, 3, 4, 5 e 6, formando a maioria necessária.

A segunda variação proposta utiliza a força bruta para selecionar o melhor conjunto de *clusters* que juntos formam uma maioria. Assim, o número de *acceptors* fica mais próximo do mínimo necessário. Esta variação demanda examinar todas as alternativas de combinação de *clusters*, verificando quantos *acceptors* corretos há em cada um, para escolher o conjunto de *clusters* para os quais fazer a difusão em paralelo. A complexidade da segunda variação é, portanto, exponencial no número de *clusters*, ou seja, $\mathcal{O}(2^{\log_2 n_a})$, onde n_a é o número de *acceptors*. Como o número de *clusters* é logarítmico, a complexidade se torna linear no número de *acceptors*.

No mesmo cenário das Figuras 1 e 2, a segunda variação faz a primeira difusão em paralelo para os *clusters* 1 e 3. Desta maneira, a quantidade de *acceptors* atingidos é 5 (0, 1, 4, 5 e 6), o mínimo necessário para maioria. Considere agora outro cenário com 14 *acceptors* sem falha, como mostra a Figura 3. Considerando que o difusor é o *acceptor* 0, na primeira variação o difusor comunica em paralelo com os dois maiores *clusters*, i.e. *clusters* 4 e 3), ou seja, com 11 *acceptors*. Na segunda variação, são selecionados os *clusters* 4 e 1, portanto a comunicação ocorre com apenas 8 *acceptors*.

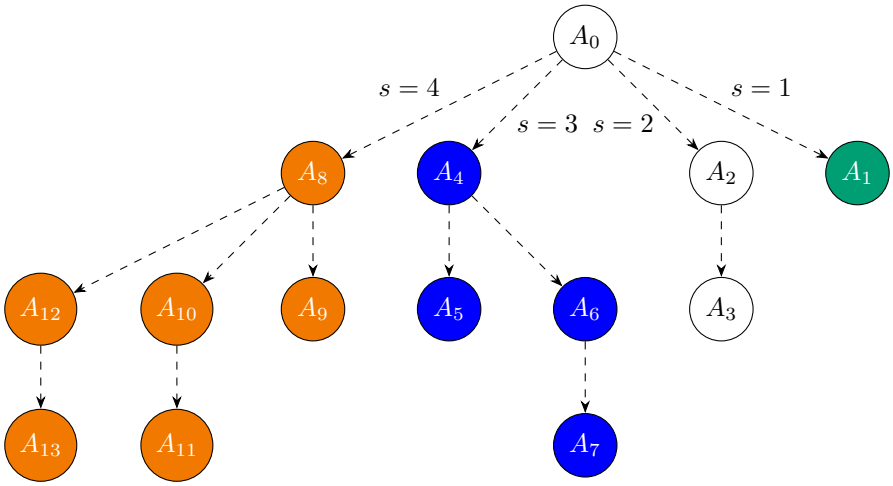

Figura 3. Difusão partindo do *acceptor* 0 em um sistema com $n_a = 14$.

Note que é possível que haja mais de um subconjunto de *clusters* que formam a maioria na estratégia na segunda variação. Neste caso, é escolhido o subconjunto com a menor quantidade de *clusters*. Vale observar que ainda outras estratégias podem ser adotadas.

É importante destacar que qualquer variação da estratégia proposta utiliza apenas um único *cluster* nos casos em que o sistema forma uma hipercubo perfeito, i.e. o número de *acceptors* é uma potência de 2 e todos os *acceptors* estão corretos. Neste caso, é mantido o desempenho original. Nos casos em que o sistema não forma um hipercubo perfeito, vários *clusters* recebem as mensagens de forma simultânea, diminuindo a latência do algoritmo.

4. Resultados Experimentais

Modificamos a libHyperPaxos [Kiotheka and Pereira 2022] de acordo com a Seção 3 para testar a estratégia proposta. Foram realizados experimentos comparando a quantidade de decisões por segundo na libHyperPaxos original, e em versões modificadas. Nos testes utilizamos um cliente e várias réplicas. O cliente é o responsável por submeter valores a um *proposer* e medir a quantidade de valores decididos por segundo.

Uma réplica é um processo que implementa os três papéis do Paxos: *proposer*, *acceptor* e *learner*. Os processos foram executados em núcleos distintos de uma mesma máquina física. A máquina possui processador AMD EPYC 7401 que possui 32 núcleos, e executa sistema operacional Linux Mint LMDE 5. Cada processo foi assinalado a um núcleo de processamento diferente.

Os sistemas testados variam de 3 a 32 réplicas, mais o cliente. As réplicas são inicializadas com o cliente. Foi medido o tempo de relógio necessário para o que o sistema decida 600 000 valores de 64 bytes, usando uma janela de pré-execução de 128 instâncias e 1024 valores simultâneos. Cada experimento foi realizado 20 vezes, sendo utilizados os maiores valores de decisões por segundo dos testes realizados. A Figura 4 apresenta os resultados de todos os experimentos executados.

Assim como em [Kiotheka et al. 2023], a quantidade de decisões por segundo tende a cair conforme o número de réplicas aumenta. Isso não é diferente na estratégia

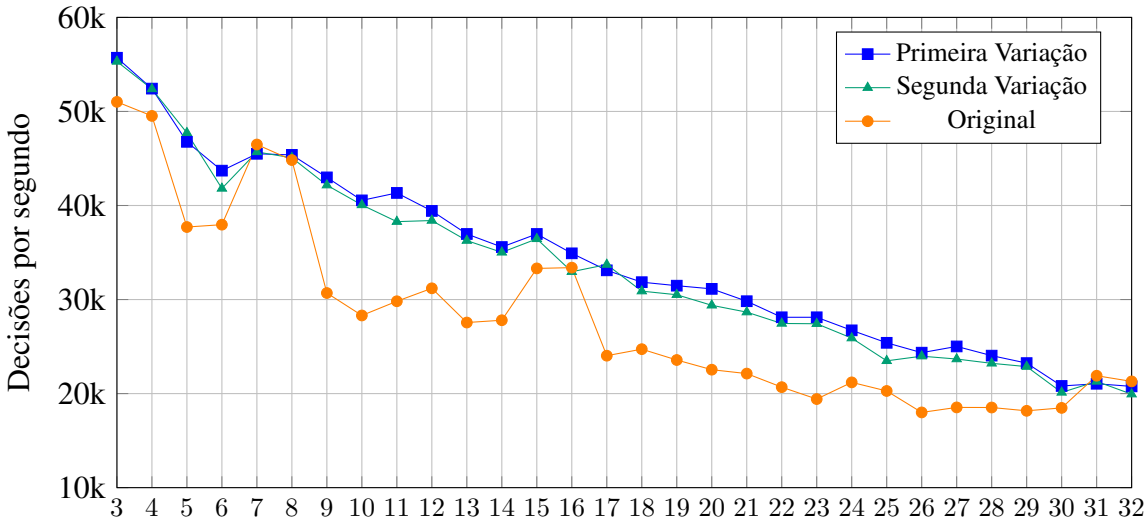

Figura 4. Valores decididos por segundo em relação ao número de réplicas.

proposta, já que o número de mensagens necessário para obter maioria cresce linearmente. Porém, a libHyperPaxos apresenta picos quando o número de réplicas é uma potência de dois, 2^k e $2^k - 1$. Isso acontece devido ao fato que o maior *cluster* possui a maioria necessária para fazer uma decisão. Já a estratégia proposta nesse trabalho apresenta curvas mais uniformes, com algumas ondulações resultantes do tamanhos dos *clusters* utilizados do vCube.

É possível verificar que as duas curvas para as duas variações da estratégia proposta possuem comportamento muito parecido, pois as duas variações são idênticas em várias ocasiões. Porém, a segunda variação perde em alguns cenários. A primeira variação é mais agressiva, fazendo a difusão de uma vez para os maiores *clusters*. A segunda busca a combinação de *clusters* com o menor número de *acceptors* necessário. Desta forma, um tempo de resposta mais longo de um dos *acceptors* pode impactar o resultado final.

5. Conclusão

Neste trabalho apresentamos uma estratégia para melhorar a vazão em termos de decisões por segundo para o algoritmo HyperPaxos, uma versão hierárquica do algoritmo Paxos sobre a topologia vCube. No algoritmo original, a difusão acontece de *cluster* em *cluster*, aguardando as respostas antes de prosseguir para o próximo *cluster*. A nossa proposta é que o difusor faça a transmissão em paralelo para o conjunto de *clusters* necessário para formar uma maioria. Duas variações foram implementadas e comparadas com a implementação original do HyperPaxos.

A primeira variação utiliza os primeiros maiores *clusters* com uma maioria de *acceptors* corretos, enquanto a segunda procura o melhor conjunto de *clusters* que juntos possuem uma maioria de *acceptors*. As duas variações apresentam desempenho mais uniforme que a estratégia originalmente proposta.

Trabalhos futuros incluem investigar mais estratégias para melhorar ainda mais o desempenho do HyperPaxos. Esta investigação inclui fazer modificações na função $c(i, s)$ para cenários de hipercubos imperfeitos na topologia vCube, e o uso de outras estratégias para seleção de *clusters*.

Referências

Benz, S. (2017). sambenz/URingPaxos: URingPaxos - A high throughput atomic multicast protocol. https://github.com/sambenz/URingPaxos.

Brewer, E. (2017). Spanner, TrueTime and the CAP Theorem. Technical report, Google.

Cachin, C., Guerraoui, R., and Rodrigues, L. (2011). *Introduction to reliable and secure distributed programming*. Springer Science & Business Media, 2nd edition.

Duarte Jr, E. P., Bona, L. C. E., and Ruoso, V. K. (2014). VCube: A Provably Scalable Distributed Diagnosis Algorithm. In *5th ScalA*, pages 17–22.

Duarte Jr, E. P. and Nanya, T. (1998). A hierarchical adaptive distributed system-level diagnosis algorithm. *IEEE Transactions on Computers*, 47(1):34–45.

Dwork, C., Lynch, N., and Stockmeyer, L. (1988). Consensus in the presence of partial synchrony. *Journal of the ACM (JACM)*, 35(2):288–323.

Hurfin, M., Mostefaoui, A., and Raynal, M. (1998). Consensus in asynchronous systems where processes can crash and recover. In *Proceedings of the 17th IEEE Symposium on Reliable Distributed Systems*, pages 280–286.

Jalili Marandi, P., Primi, M., Schiper, N., and Pedone, F. (2017). Ring Paxos: High-throughput atomic broadcast. *The Computer Journal*, 60(6):866–882.

Kiotheka, F. M. and Pereira, D. R. (2022). HyperPaxos / LibHyperPaxos - GitLab. https://gitlab.c3sl.ufpr.br/hyperpaxos/libhyperpaxos.

Kiotheka, F. M., Pereira, D. R., Camargo, E. T., and Duarte Jr, E. P. (2023). HyperPaxos: Uma Versão Hierárquica do Algoritmo de Consenso Paxos. *41o Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos (SBRC)*, pages 1–14.

Lamport, L. (1998). The Part-Time Parliament. *ACM Transactions on Computer Systems*, 16(2):133–169.

Lamport, L. (2001). Paxos made simple. *ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121, December 2001)*, pages 51–58.

Primi, M. and Sciascia, D. (2013). LibPaxos: Open-source Paxos. http://libpaxos.sourceforge.net/.

Red Hat (2019). What is etcd? https://www.redhat.com/en/topics/containers/what-is-etcd.

Regis, S. and Mendizabal, O. M. (2022). Análise comparativa do algoritmo Paxos e suas variações. In *XXIII WTF*, pages 71–84.

Renesse, R. v. and Altinbuken, D. (2015). Paxos Made Moderately Complex. *ACM Computing Surveys (CSUR)*, 47(3):1–36.

Rodrigues, L. A., Duarte Jr, E. P., and Arantes, L. (2018). A distributed k-mutual exclusion algorithm based on autonomic spanning trees. *JPDC*, 115:41–55.