HyperPaxos em Múltiplas Instâncias sobre a LibPaxos: Uma Versão Hierárquica do Algoritmo de Consenso Paxos

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 o algoritmo HyperPaxos, 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. A difusão ocorre do maior* cluster *para o menor, retornando ao* proposer *quando uma maioria é atingida. O HyperPaxos foi implementado como a biblioteca libHyperPaxos. Resultados obtidos mostram o bom desempenho da libHyperPaxos, que inclusive supera a libPaxos e, em alguns casos, a implementação do U-Ring Paxos em decisões por segundo.*

Abstract. *Distributed consensus algorithms are essential components of storage systems and databases, and have multiple other applications, such as in access control and orchestration of cloud applications. This paper presents the HyperPaxos algorithm, a hierarchical version of one of the most important consensus algorithms, Paxos. HyperPaxos is based on the vCube hierarchical virtual topology, which has several logarithmic properties. Acceptors are organized in clusters and proposers execute the two Paxos phases by choosing an acceptor that is responsible for relaying the messages to the other acceptors across the vCube. Broadcast starts from the largest cluster to the smallest, returning to the proposer as soon as a majority is reached. HyperPaxos was implemented as the libHyperPaxos library. Results obtained highlight the performance of libHyperPaxos, which surpasses libPaxos and, in some cases, the implementation of U-Ring Paxos in decisions per second.*

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 o sistema de armazenamento distribuído Ceph [Weil et al. 2007] e os 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. Uma das características do Paxos que implica no seu alto custo é o fato de ser baseado em comunicação 1-para-todos. No Ring Paxos este problema é resolvido organizando os processos em anel, de forma que cada processo se comunica com apenas um outro processo, até atingir uma maioria. Ao mesmo tempo, nenhum processo é sobrecarregado com uma quantidade grande de mensagens. Porém, apesar da comunicação em anel necessitar do número mínimo de mensagens, a estrutura sequencial do anel leva a um potencial aumento da latência do algoritmo.

O presente trabalho apresenta o HyperPaxos [Kiotheka and Pereira 2022b, Kiotheka et al. 2023, Kiotheka and Pereira 2022a], uma versão do algoritmo Paxos que utiliza a topologia distribuída hierárquica vCube [Duarte Jr et al. 2014] para organizar os *acceptors*. 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*.

Para avaliar o algoritmo HyperPaxos, implementamos a biblioteca de código aberto libHyperPaxos. A libHyperPaxos é uma implementação do algoritmo HyperPaxos feita com base na terceira versão da libPaxos. A libHyperPaxos altera a comunicação entre os papéis, sem alterar a lógica do algoritmo Paxos já implementada. São apresentados resultados de comparação da libHyperPaxos com a libPaxos e com uma das implementações do Ring Paxos, o U-Ring Paxos. O U-Ring Paxos foi escolhido para comparação pois tem um padrão de comunicação similar com a utilização de *unicast* TCP, e seu desempenho é similar ao da outra implementação, o M-Ring Paxos, que é implementado com *multicast* UDP.

O trabalho tem, desta forma, três contribuições principais: (i) a especificação do algoritmo hierárquico HyperPaxos, baseado no Paxos e no vCube; (ii) a implementação do HyperPaxos utilizando a biblioteca libPaxos; (iii) a comparação experimental do Hy-

perPaxos tanto com o Paxos original como com o Ring Paxos.

O restante deste trabalho está organizado da seguinte forma. A Seção 2 apresenta o algoritmo HyperPaxos, destacando como é feita a organização dos papéis no algoritmo, bem como a comunicação entre eles. Na Seção 3 a implementação da biblioteca libHyperPaxos é descrita. Em seguida, a Seção 4 apresenta os resultados experimentais de comparação da libHyperPaxos com a terceira versão da libPaxos e com o U-Ring Paxos. E por fim, as conclusões seguem na Seção 5.

2. HyperPaxos

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 organiza os processos em *clusters* hierárquicos, 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.

Como no HyperPaxos os *acceptors* que formam a topologia do vCube, cada *acceptor* se comunica com no máximo $\lceil \log_2 n \rceil$ *clusters* de *acceptors*. 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

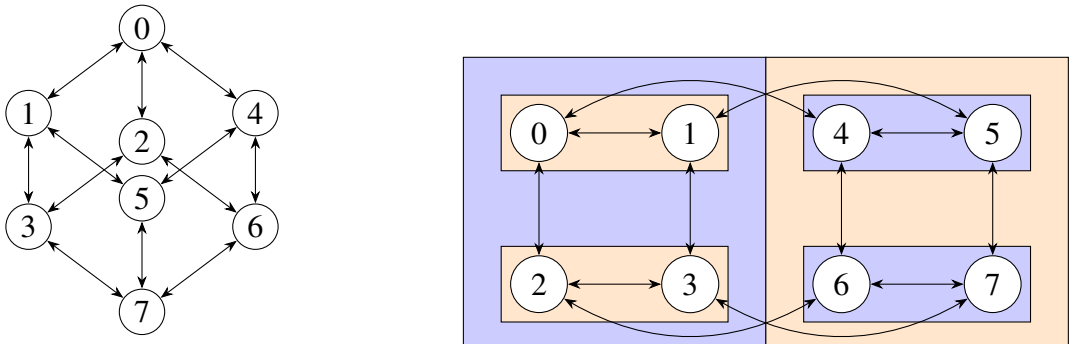
$$c(i, s) = (i \oplus 2^{s-1}, c(i \oplus 2^{s-1}, 1), c(i \oplus 2^{s-1}, 2), \ldots, c(i \oplus 2^{s-1}, s-1)).$$

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. A Figura 1 mostra como fica a organização dos *acceptors* num sistema com 8 *acceptors*.

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



(a) Representação em hipercubo

(b) Representação planificada

Figura 1. 8 *acceptors* organizados na topologia vCube.

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, encaminha o pedido para os todos os primeiros processos 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 respostas 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.

A Figura 2 mostra um sistema distribuído com 2 *proposers* e 8 *acceptors*. Neste exemplo, o *proposer* 0 escolhe o *acceptor* 0 como difusor e envia seu pedido de preparação PREP com número de proposta 1. Ao receber o pedido de preparação, o *acceptor* 0 encaminha o pedido de preparação com a sua resposta PROM para o *acceptor*

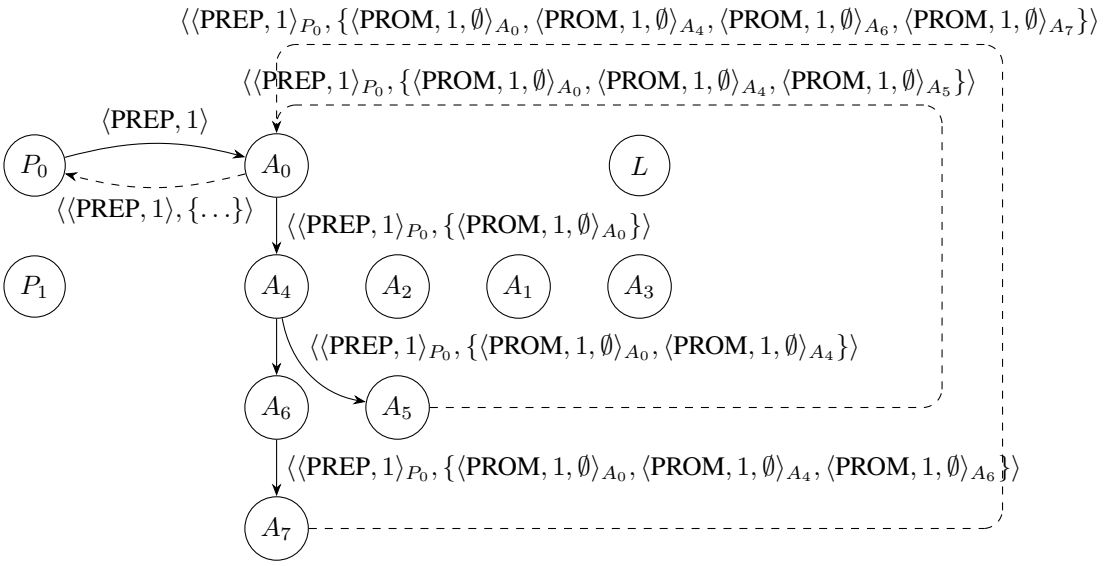


Figura 2. Difusão na fase 1 do maior *cluster* do *acceptor* 0 em um sistema com $n_a = 8$.

4 do *cluster* 3. O *acceptor* 4 faz o mesmo, encaminhando o pedido de preparação com a resposta anterior e sua resposta para os *acceptors* 5 e 6, dos *clusters* 1 e 2. Por fim, o *acceptor* 6 envia o pedido de preparação com as respostas para o *acceptor* 7.

Os *acceptors* 5 e 7 são folhas na árvore de difusão, e portanto encaminham as respostas para o *acceptor* difusor 0. Ao receber todas as respostas, o *acceptor* 0 verifica se existe uma maioria de promessas que validam o número de proposta para poder enviar para o *proposer* 0. Como existe uma maioria nesse caso, o *acceptor* 0 pode enviar essas respostas para o *proposer*, validando o número de proposta 1 e permite que ele prossiga para a próxima fase.

Suponha agora que o *proposer* executa a fase 2 e o *acceptor* 7 falhou como mostra a Figura 3. O *proposer* 0 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* 1 prossegue para o seu *cluster* 2, enviando a proposta para o *acceptor* 3 como mostra a Figura 4. 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*.

A difusão adotada segue o algoritmo de árvore geradora mínima encontrada em [Rodrigues et al. 2014]. Assim, é garantido que em um sistema síncrono, na difusão de

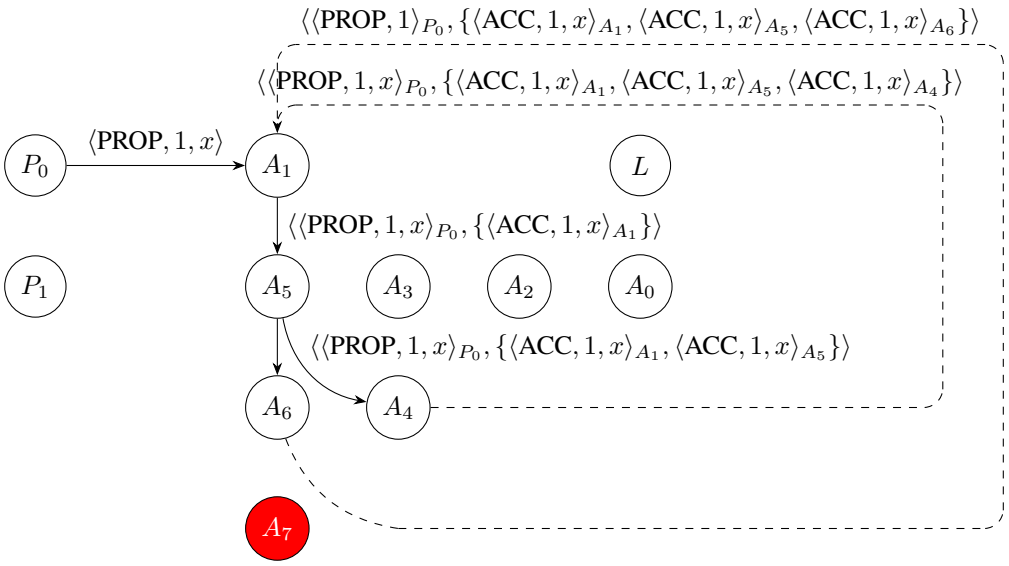


Figura 3. Difusão da fase 2 do maior *cluster* do *acceptor* 1 em um sistema com $n_a = 8$ e *acceptor* 7 falho.

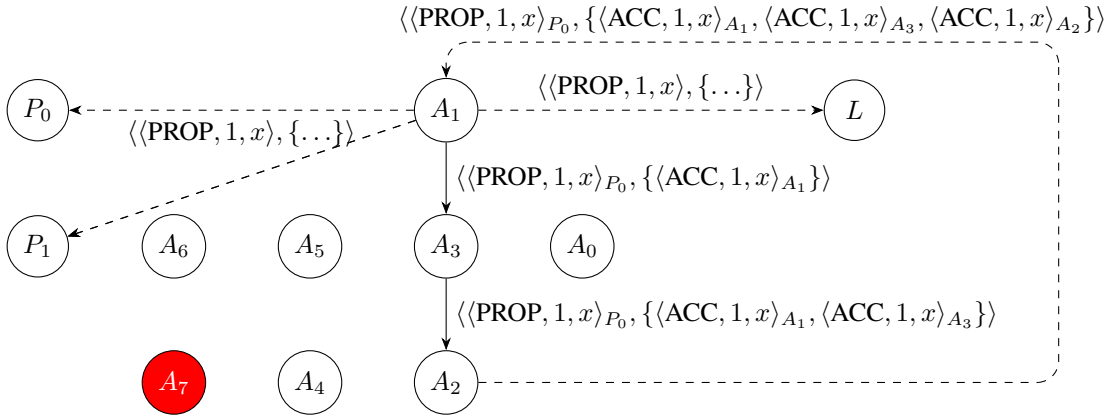


Figura 4. Difusão na fase 2 do segundo maior *cluster* do *acceptor* 1 em um sistema com $n_a = 8$ e *acceptor* 7 falho.

cada *cluster*, o difusor recebe respostas de todos os processos corretos do *cluster*. Quando há falhas, retransmissões podem ser empregadas ou o difusor pode instruir o *proposer* a fazer uma nova difusão. Note que antes do GST, processos corretos podem ser considerados falhos pelo detector de falhas, o que implica que nem todos os processos corretos recebem a mensagem. Porém, isso não compromete a corretude do Paxos, mas a sua terminação não é garantida [Lamport 2001].

Para eliminar respostas duplicadas, uma estratégia foi adotada de só difundir respostas no maior *cluster* com processos corretos na hierarquia. O restante dos *clusters* recebe um conjunto de respostas vazio. A Figura 5 mostra um exemplo de difusão completa com esta otimização, onde o *acceptor* 0 inicia a difusão de uma mensagem m para todos os *acceptors* para todos os *clusters*. O *acceptor* 4 do *cluster* 3 recebe a resposta do *acceptor* 0, mas os outros *clusters* recebem um conjunto de respostas vazio. O *acceptor* 4 faz o mesmo e envia o conjunto de respostas do *acceptor* 0 e 4 apenas para o *acceptor* 6 do *cluster* 2, enquanto o *cluster* 1 recebe um conjunto de respostas vazio. O resultado

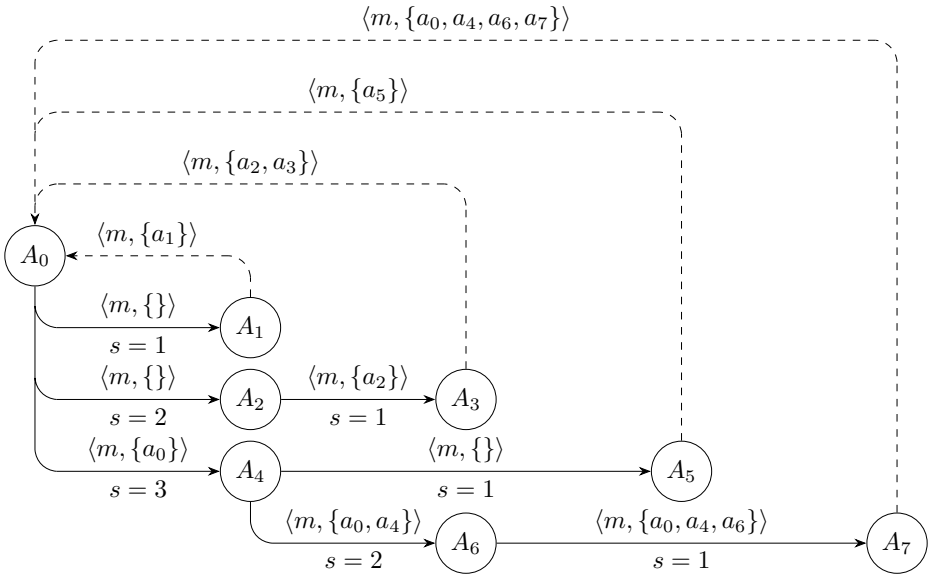


Figura 5. Difusão partindo do processo 0 em um sistema de 8 processos.

é que a união das respostas dos *acceptors* que são folhas 1, 3, 5 e 7 não contém nenhuma duplicação.

3. Implementação libHyperPaxos

A libHyperPaxos [Kiotheka and Pereira 2022a] implementa o algoritmo HyperPaxos seguindo como base a implementação da libPaxos [Primi and Sciascia 2013]. A biblioteca é divida em duas partes: *libhyperpaxos* e *libevhyperpaxos*. A *libhyperpaxos* contém a lógica do HyperPaxos, que encapsula a lógica do Paxos da libPaxos, sem código de protocolos de rede. Já a *libevhyperpaxos* contém a implementação dos protocolos de rede usando a *libevent* e a *libhyperpaxos*.

Para usar a libHyperPaxos é necessário um detector de falha, pois a mesma não possui este mecanismo. O detector de falhas deve notificar os eventos de falha e recuperação, indicando o processo e qual foi o evento.

Os papéis na libHyperPaxos funcionam de maneira semelhante à libPaxos, e também são bem separados em *proposer*, *acceptor* e *learner*. O papel de *replica* também existe, sendo a junção dos três papéis do Paxos. Os clientes submetem seus valores aos *proposers* e para aprenderem sobre as decisões do sistema, precisam executar o papel de *learner*.

Também usamos TCP na libHyperPaxos, porém a responsabilidade dos papéis de iniciar a conexão se altera devido à topologia do vCube. Os clientes ainda se conectam a um ou mais *proposers* e os *proposers* e *learners* se conectam a todos os *acceptors*, como na libPaxos. No entanto, os *acceptors* precisam comunicar entre si devido à hierarquia da topologia. Para tal, cada *acceptor* se conecta a outros *acceptors* de identificador maior, isto é, para cada *acceptor* i, i inicia a conexão com o *acceptor* j se $i < j$.

Para diminuir o tamanho das mensagens na rede e aumentar a vazão, adotamos a estratégia de eliminar as respostas duplicadas. Para tal, basta garantir que as respostas são difundidas apenas no maior cluster. Além disso, as respostas são codificadas de uma

forma eficiente. Um conjunto separado com os identificadores dos *acceptors* que validam o número de proposta na fase 1 ou aceitam a proposta na fase 2 é utilizado. Isso faz com que o caso mais comum tenha mensagens menores, que é aquele caso no qual as fases 1 e 2 são aceitas por todos. Essa otimização só é possível por a difusão ser feita hierarquicamente, e isso permite que o tamanho das mensagens que trafegam na rede seja reduzido, também permitindo maior vazão.

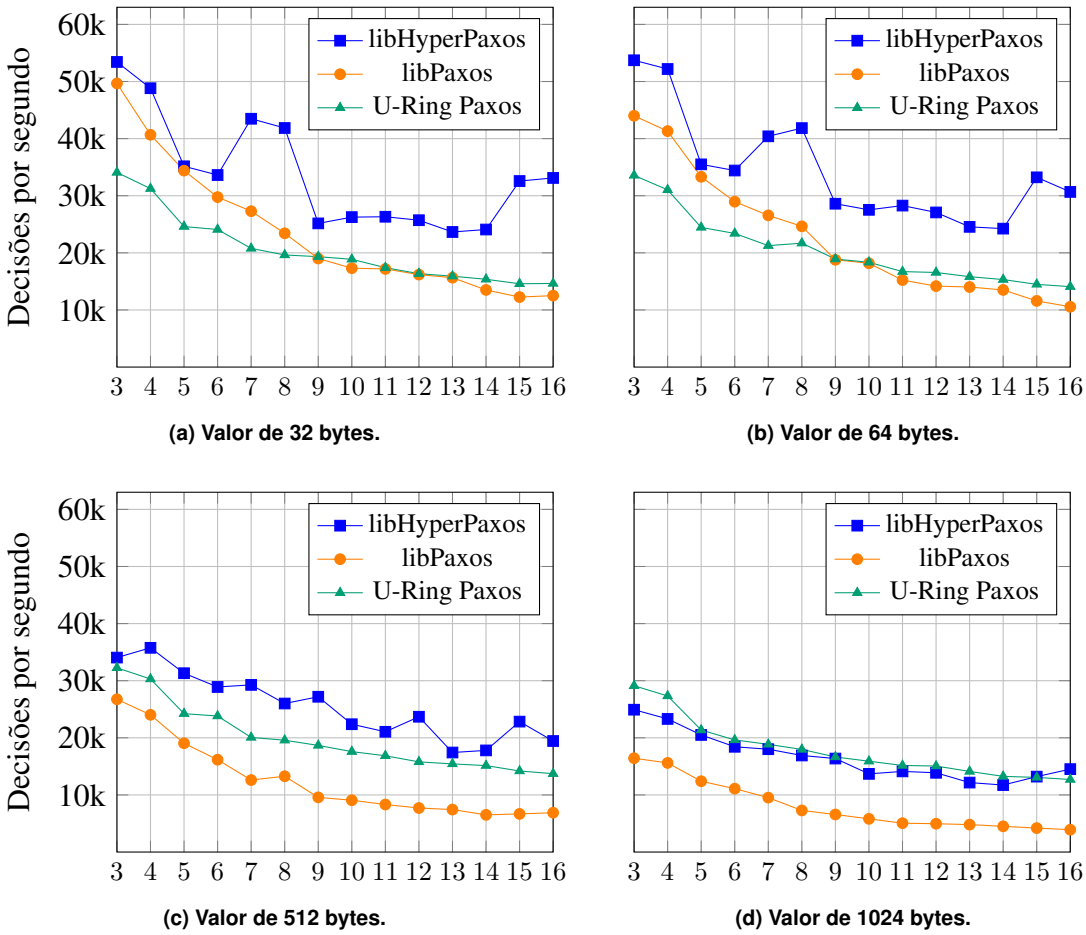
4. Resultados experimentais

Foram realizados experimentos comparando a quantidade de decisões por segundo na libHyperPaxos, na libPaxos e na implementação do U-Ring Paxos [Benz 2017] em sistemas com todos os processos corretos. Nos testes da libHyperPaxos e libPaxos, utilizamos um cliente e várias réplicas. No U-Ring Paxos, um *proposer* é quem age como cliente. O cliente é um processo que manda valores fixos para um *proposer* selecionado e aprende os valores decididos executando o papel de *learner*. Sabendo disso, o cliente mede quantos valores são decididos por segundo. A réplica é um processo que executa todos os papéis do Paxos e pode propor, decidir e aprender valores.

O primeiro experimento foi realizado em máquinas físicas distintas comparando apenas a libPaxos e a libHyperPaxos em [Kiotheka and Pereira 2022b]. Já em [Kiotheka et al. 2023], a comparação foi com as três implementações, no qual os processos foram executados em núcleos distintos de uma mesma máquina física. A máquina possui processador AMD EPYC 7401 que possui 24 núcleos, e executa sistema operacional Linux Mint LMDE 5. Cada processo foi assinalado a um núcleo de processamento diferente. Para executar a implementação do U-Ring Paxos, também foi executado o ZooKeeper versão 3.8.0 sobre OpenJDK 17.0.4, mesma versão do Java utilizada para execução do U-Ring Paxos. O ZooKeeper é utilizado pelo U-Ring Paxos apenas para configuração dos processos.

As bibliotecas foram executadas com parâmetros de configuração que potencializam a quantidade de valores decididos por segundo. Os sistemas testados variam de 3 a 16 réplicas, mais o cliente. As réplicas são inicializadas com o cliente. No caso da libPaxos e da libHyperPaxos, são coletadas informações de decisões por segundo, de segundo em segundo. Após 10 segundos, a amostra do último segundo de decisões por segundo é usada, quando se encontra mais estável. A Figura 6 apresenta os resultados de todos os experimentos executados. São quatro gráficos com vários tamanhos de valores decididos, de 32 a 1024 bytes. O eixo y representa a quantidade de valores decididos por segundo e o eixo x representa a quantidade de réplicas que estão presentes no sistema testado.

Em todas as implementações, a quantidade de decisões por segundo tende a cair conforme o número de réplicas aumenta. No entanto, a libHyperPaxos apresenta picos de mais decisões quando o número de réplicas é uma potência de dois, 2^k ou $2^k - 1$. Neste caso, apenas o primeiro *cluster* é necessário para alcançar uma maioria. Ainda, pode-se notar que a libHyperPaxos teve uma quantidade maior de valores decididos por segundo, quando comparada à libPaxos em todos os números de réplica. Isso acontece devido ao HyperPaxos enviar mensagens apenas para uma quantidade necessária de *acceptors*, ou seja, até obter uma maioria. Além disso, as mensagens são melhor distribuídas na rede, pois o *acceptor* difusor sempre muda, em contraste à libPaxos em que *proposers* ficam responsáveis pelo envio de mensagens a todos os *acceptors* o tempo todo.



(a) Valor de 32 bytes.

(b) Valor de 64 bytes.

(c) Valor de 512 bytes.

(d) Valor de 1024 bytes.

Figura 6. Valores decididos por segundo em relação ao número de réplicas.

Quando comparada a U-Ring Paxos, a libHyperPaxos a supera em decisões por segundo em valores inferiores a 1024 bytes. Na implementação do U-Ring Paxos, a quantidade de valores decididos permanece constante pois apenas os identificadores dos valores são propagados na rede. Assim, implementando esta otimização na própria libHyperPaxos, é esperado que o desempenho obtido seja igual em todos os tamanhos de valor.

5. Conclusão

Neste trabalho apresentamos o HyperPaxos, uma versão hierárquica do algoritmo Paxos sobre a topologia vCube. O HyperPaxos organiza os *acceptors* em *clusters* para formar a topologia virtual. Os *proposers* enviam suas mensagens para um *acceptor* chamado difusor e este é responsável por transmitir as mensagens para os demais *acceptors* no vCube. A difusão para os *acceptors* ocorre hierarquicamente em *clusters*, sendo as respostas concatenadas à mensagem original conforme a árvore de difusão é percorrida. No melhor caso, apenas a maioria necessária de *acceptors* receberá mensagens para executar o algoritmo, diminuindo a quantidade de mensagens enviadas.

O algoritmo HyperPaxos foi implementado como a biblioteca libHyperPaxos e comparado com a libPaxos e a implementação do U-Ring Paxos. A métrica para comparação do desempenho das bibliotecas foi a quantidade de valores decididos por segundo, variando o tamanho dos valores e o número de réplicas. Foram utilizados para

as respectivas bibliotecas seus melhores parâmetros. Os resultados mostram que a libHyperPaxos supera a libPaxos em todos os testes, e para valores de tamanho menor que 1024 bytes supera o U-Ring Paxos.

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.

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 *The 17th SRDS*, 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. (2022a). HyperPaxos / LibHyperPaxos - GitLab. `https://gitlab.c3sl.ufpr.br/hyperpaxos/libhyperpaxos`.

Kiotheka, F. M. and Pereira, D. R. (2022b). *HyperPaxos em múltiplas instâncias sobre a LibPaxos: Uma versão hierárquica do algoritmo de consenso Paxos*. TCC, UFPR.

Kiotheka, F. M., Pereira, D. R., Camargo, E. T., and Duarte Jr, E. P. (2023). HyperPaxos: Uma Versão Hierárquica do Algoritmo de Consenso Paxos. *XLI 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. (2014). Árvores geradoras mínimas distribuídas e autonômicas. *XXXII SBRC*, 2014:1–14.

Weil, S. A., Leung, A. W., Brandt, S. A., and Maltzahn, C. (2007). Rados: a scalable, reliable storage service for petabyte-scale storage clusters. In *Proceedings of the 2nd international workshop on Petascale data storage*, pages 35–44.