HyperPaxos: Uma Versão Hierárquica do Algoritmo de Consenso Paxos

Fernando M. Kiotheka[1], **Djenifer R. Pereira**[1],
Edson T. Camargo[2], **Elias P. Duarte Jr.**[1]

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

[2]Universidade Tecnológica Federal do Paraná (UTFPR)
R. Cristo Rei, 19 – 85902-490 – Toledo/PR

{fmkiotheka,drpereira,elias}@inf.ufpr.br, edson@utfpr.edu.br

Resumo. *O consenso é um problema fundamental de sistemas distribuídos. Neste trabalho é proposto 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. O HyperPaxos organiza os* acceptors *em* clusters *e os* proposers *enviam suas mensagens para um* acceptor *dito difusor que faz a retransmissão para os demais* acceptors *usando difusão sobre o vCube. Inicialmente, o difusor envia a mensagem para o seu maior* cluster *na tentativa de conseguir uma maioria para a fase 1 ou 2. Caso não consiga, continua a difusão para seus próximos* clusters*, do maior para o menor. 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.*

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], bancos de dados distribuídos como o Google Spanner e o Cassandra [Brewer 2017, Google 2023, Ellis 2013], ferramentas de controle de acesso como o Chubby [Burrows 2006] 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 algoritmo empregar 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 fica 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 propõe o HyperPaxos, uma nova versão do algoritmo Paxos baseada na topologia distribuída hierárquica vCube [Duarte et al. 2022]. O vCube é um algoritmo de detecção de falhas, que foi inicialmente proposto [Duarte Jr and Nanya 1998, Duarte Jr et al. 2014], no contexto de diagnóstico distribuído. Os processos no vCube 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. Na medida em que os processos falham, o vCube mantém diversas propriedades logarítmicas, como a distância máxima entre os processos e o número de mensagens que necessitam para comunicar. Desta forma, devido às propriedades topológicas do vCube, o HyperPaxos pode ser visto como uma solução entre o Paxos e o RingPaxos.

O HyperPaxos utiliza o vCube para organizar os *acceptors*. A lógica original do Paxos é mantida, alterando somente a comunicação entre/de/para *acceptors*. Os *proposers* fazem as requisições para um *acceptor* denominado difusor, que 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 à mensagem original. As folhas da árvore de difusão encaminham as respostas para o *acceptor* responsável pela difusão. Quando este recebe uma maioria de respostas que confirmam a requisição, ele as reencaminha para o *proposer*. Caso a maioria não seja atingida, o *acceptor* difusor prossegue para o próximo *cluster*.

Em [Terra et al. 2020], foi feita a proposta que especifica uma única instância do Paxos baseada no vCube. Aquela solução difere em vários aspectos do HyperPaxos. Em primeiro lugar, o HyperPaxos é uma versão completa do Paxos, permitindo a execução de múltiplas instâncias (também chamado de "Multi-Paxos"). Além disso, no HyperPaxos, os papéis são completamente separados, de forma que o processo difusor é capaz de executar em um processo diferente, permitindo maior desempenho. No [Terra et al. 2020] os processos executam os três papéis, e um processo é escolhido como coordenador. Por fim, o trabalho de [Terra et al. 2020] implementa a instância do Paxos utilizando simulação, e diversos elementos do protocolo não são especificados, como mensagens por exemplo.

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 [Primi and Sciascia 2013]. 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

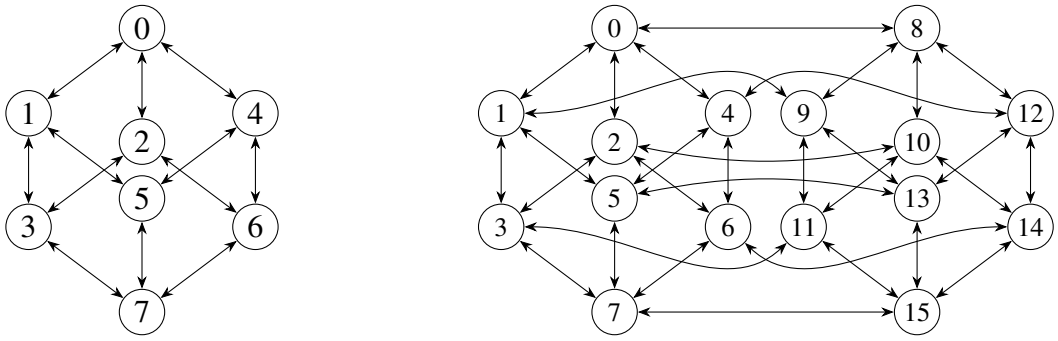
com uma das implementações do Ring Paxos, o U-Ring Paxos. O U-Ring Paxos foi escolhido para comparação pois utiliza uma estratégia de comunicação compatível, o *unicast* TCP. Além disso, o desempenho do U-Ring Paxos é similar ao da outra implementação, o M-Ring Paxos, que utiliza *multicast* UDP.

O restante desse trabalho está organizado da seguinte forma. A Seção 2 apresenta de forma resumida a topologia hierárquica virtual vCube. Já na Seção 3 é encontrada a descrição da difusão de melhor esforço sobre o vCube. Na Seção 4 é descrito o algoritmo HyperPaxos. Na Seção 5 é apresentada a implementação do HyperPaxos baseada na biblioteca libPaxos junto com os resultados experimentais obtidos nas comparações com a libPaxos e U-Ring Paxos. As conclusões seguem na Seção 6.

2. A Topologia Hierárquica vCube

O vCube é um detector de falhas distribuído que mantém uma topologia virtual hierárquica [Duarte et al. 2022]. O vCube foi originalmente proposto como um algoritmo de diagnóstico adaptativo e distribuído [Duarte Jr and Nanya 1998, Duarte Jr et al. 2014]. A topologia do vCube é versátil e já foi utilizada para diversas aplicações, como por exemplo a construção de redes overlay [Bona et al. 2006], a exclusão mútua distribuída [Rodrigues et al. 2013a] e sistemas *publish-subscribe* [De Araujo et al. 2017].

Quando todos os processos estão corretos e o número de processos é uma potência de dois, a topologia criada é de um hipercubo, como mostra a Figura 1. O hipercubo possui simetria e diâmetro logarítmico, o que garante escalabilidade. Conforme os processos vão falhando ou recuperando, o vCube se reorganiza, preservando suas propriedades logarítmicas.



(a) 8 processos **(b) 16 processos**

Figura 1. Hipercubo formado por 8 e 16 processos.

O vCube é um detector de falha do tipo *pull-based*, que se baseia em testes executados entre os processos. Quando todos os processos executam seus testes assinalados, acontece uma rodada de testes. O vCube assume o modelo de falha *crash*, mas pode ser trivialmente estendido para o modelo *crash-recovery* [Duarte et al. 2022]. O modelo temporal do vCube é assíncrono, e assim não é possível ter certeza que um processo está realmente falho, pois ele pode apenas estar muito lento. Logo, o estado de um processo monitorado pode ser ou *correto* ou *suspeito* de estar falho. Quando um processo falha ou recupera, o vCube garante que em um sistema com n processos, todos os processos corretos aprendem sobre essa nova mudança em no máximo $\log_2 n$ rodadas de teste.

O vCube define uma estratégia hierárquica de testes, na qual os processos são organizados em *clusters* cada vez maiores. Os *clusters* são definidos pela função $c(i, s)$ que retorna a sequência de processos do *cluster* s do processo i. Uma definição para a função $c(i, s)$ onde $\oplus$ denota a operação de ou exclusivo é dada por

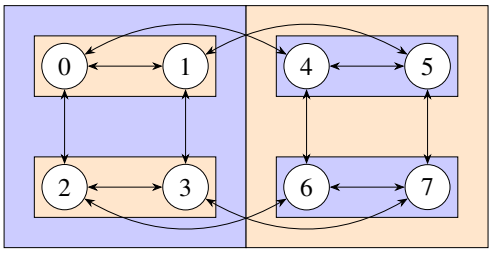
$$c(i, s) = (i \oplus 2^{s-1}, c(i \oplus 2^{s-1}, 1), c(i \oplus 2^{s-1}, 2), \ldots, c(i \oplus 2^{s-1}, s-1)).$$

Os processos corretos testam todos os seus *clusters* a cada rodada de teste, ou seja, do *cluster* com $s = 1$ até o *cluster* com $s = \log_2 n$. O testador de um processo j é i se i é o primeiro processo correto em $c(j, s)$. Assim, no teste do *cluster* s, o processo i faz testes em todos os processos da sequência $c(i, s)$ cujo testador é i. Por meio do teste, o processo i descobre se o processo testado está correto, e se estiver, obtém dele informações novas sobre o estado de todos os processos do sistema. Assim, os processos aprendem sobre mudanças dos estados de todos os processos, sem precisar testar todos diretamente.

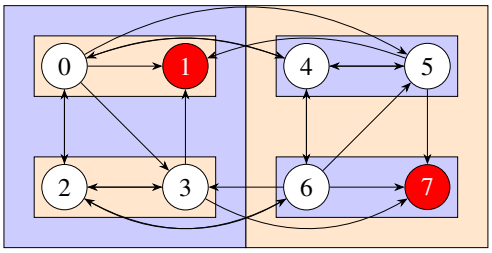
s	$c(0, s)$	$c(1, s)$	$c(2, s)$	$c(3, s)$	$c(4, s)$	$c(5, s)$	$c(6, s)$	$c(7, s)$
1	(1)	(0)	(3)	(2)	(5)	(4)	(7)	(6)
2	$(2, 3)$	$(3, 2)$	$(0, 1)$	$(1, 0)$	$(6, 7)$	$(7, 6)$	$(4, 5)$	$(5, 4)$
3	$(4, 5, 6, 7)$	$(5, 4, 7, 6)$	$(6, 7, 4, 5)$	$(7, 6, 5, 4)$	$(0, 1, 2, 3)$	$(1, 0, 3, 2)$	$(2, 3, 0, 1)$	$(3,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

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

Por exemplo, para um sistema com 8 processos, a Tabela 1 lista o $c(i, s)$ de cada um dos processos. Se o sistema não possui nenhuma falha, cada um dos processos testa 3 outros processos, um para cada *cluster* como mostra a Figura 2(a) que é uma nova representação da Figura 1(a). O $c(i, s)$ neste caso é simétrico: o primeiro processo correto j no cluster s tem como primeiro processo correto em $c(j, s)$ o mesmo processo i.



(a) vCube formado por 8 processos.



(b) vCube formado por 8 processos sendo 1 e 7 falhos.

Porém, se os processos 1 e 7 falham, por exemplo, as suas responsabilidades acumulam para os processos 6 e 0 que agora precisam testar 5 processos ao invés de 3 como mostra a Figura 2(b). No caso do processo 0, isso acontece porque o testador do processo 3 no *cluster* 2, dado pelo primeiro processo correto em $c(3, 2)$ agora é o processo 0, pois o processo 1 falhou. O mesmo ocorre no *cluster* 2 com o processo 5 que tem agora como testador o processo 6, já que o processo 7 também está falho. No *cluster* 3, os processos 5 e 3 são testados por 0 e 6 respectivamente.

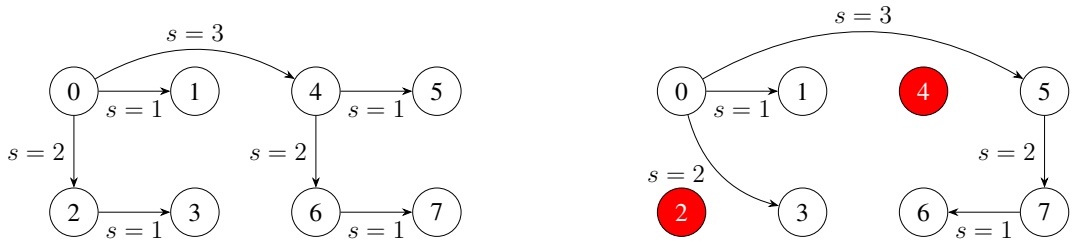
3. Difusão de melhor esforço sobre o vCube

A difusão (*broadcast*) é um tipo de transmissão de mensagens em que um processo origem, chamado neste trabalho de emissor, envia uma mesma mensagem para todos os

outros processos do sistema distribuído. Desta forma, todos os processos do sistema são destinatários da mensagem. Com o *broadcast* e suas variações, é possível implementar diversos algoritmos e serviços como notificação global, entrega de conteúdo, comunicação em grupo, replicação e exclusão mútua [de Araujo et al. 2017, Lamport 1978, Rodrigues et al. 2013b].

Existem diversos tipos de difusão, cada uma atendendo propriedades específicas. Definimos informalmente como difusão de melhor esforço aquela que garante que se o processo emissor da mensagem m não falha, então todo processo correto, após um intervalo de tempo finito, entrega m. Além disso, nenhuma mensagem é entregue mais de uma vez e os processos entregam apenas mensagens que foram previamente transmitidas [Cachin et al. 2011].

Em [Rodrigues et al. 2014] é proposta uma solução para a difusão de melhor-esforço baseada na topologia do vCube. A responsabilidade de fazer a difusão é dividida entre os processos de forma hierárquica, permitindo maior vazão. Para fazer a difusão de uma mensagem, o processo emissor envia a mensagem para o primeiro processo correto em cada um dos seus *clusters*, além de enviar a mensagem para si mesmo. Cada processo que recebe a mensagem, entrega a mensagem caso seja nova, e continua a difusão enviando a mensagem para todos os seus *clusters* de tamanho menor. Isto é, quando um processo i envia uma mensagem para o primeiro processo correto j de $c(i, s)$, j envia uma mensagem para todo primeiro processo correto dos seus *clusters* de $c(j, 1)$ até $c(j, s-1)$.



(a) Exemplo sem falhas.

(b) Exemplo com os processos 2 e 4 falhos.

Figura 3. Difusão de melhor esforço partindo do processo 0 em um sistema de 8 processos.

Em um sistema com 8 processos, uma difusão partindo do processo 0 acontece como mostra a Figura 3(a). Primeiramente, o processo 0 envia a mensagem para o processo 1 do *cluster* 1, o processo 2 do *cluster* 2 e o processo 4 do *cluster* 3. Ao receberem a mensagem, os processos 1, 2 e 4 continuam a difusão hierarquicamente. O processo 1 não encaminha mais mensagens, pois não tem mais *clusters* sob sua responsabilidade. Já os processos 2 e 4, continuam a difusão. O processo 2 envia a mensagem para o processo 3 do seu *cluster* 1, e o processo 4 envia a mensagem para os processos 5 e 6, dos *clusters* 1 e 2 respectivamente. Por fim, o processo 6 encaminha a mensagem para o processo 7 do seu *cluster* 1, completando o caminho mais longo no sistema de tamanho $\log_2 8 = 3$.

Em um cenário de falha, o sistema adquire a forma encontrada na Figura 3(b) que mostra os processos 2 e 4 falhos. Neste caso, o processo 0 possui como primeiro processo correto do *cluster* 2 o processo 3, e o primeiro processo correto do *cluster* 3 se torna o 5. Note que o 5 que previamente não tinha responsabilidade de envio, agora é responsável

por enviar mensagens para os *clusters* 1 e 2. Porém, não há processo correto em seu *cluster* 1, e o processo 5 então encaminha a mensagem para o processo 7 do *cluster* 2. O processo 7 que também não tinha responsabilidade de enviar uma mensagem agora precisa enviar uma mensagem para o processo 6 do seu *cluster* 1.

O algoritmo de difusão de melhor esforço sobre o vCube permite que falhas aconteçam durante a difusão, fazendo com que retransmissões aconteçam nos casos dos processos falharem. Note, porém, que o algoritmo pode não funcionar corretamente em um sistema assíncrono. Isto porque se um processo é julgado falho, mas não está falho, a difusão nem sequer o considera para envio. Isto é resolvido em [Rodrigues et al. 2017] com o envio de mensagens especiais para processos considerados falhos.

4. HyperPaxos: Paxos em múltiplas instâncias sobre o vCube

Nesta seção é proposto o algoritmo HyperPaxos. O algoritmo assume um modelo de tempo parcialmente síncrono com GST (*Global Stabilization Time*) [Dwork et al. 1988] e 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*. Para realizar as fases 1 e 2, os *proposers* então precisam se comunicar usando um *acceptor* intermediador que realiza a transmissão sobre o vCube para os demais *acceptors*. Esta transmissão é feita de maneira similar à difusão de melhor esforço sobre o vCube descrita na Seção 3.

Assim, na fase 1, o *proposer* envia um pedido de preparação para um *acceptor*, que será responsável por difundir a mensagem no vCube. Este *acceptor* faz a difusão para os seus *clusters*, do maior para o menor, um por vez. A resposta do *acceptor* difusor vai junto do pedido de preparação que ele difunde para seus *clusters*. E cada *acceptor*, ao receber um pedido de preparação, faz o mesmo, encaminhando o pedido para os primeiros *acceptors* de seus *clusters* menores no vCube, junto da sua própria resposta ao pedido de preparação. Caso o *acceptor* seja 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 mais *clusters*. Caso o *acceptor* esgote 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.

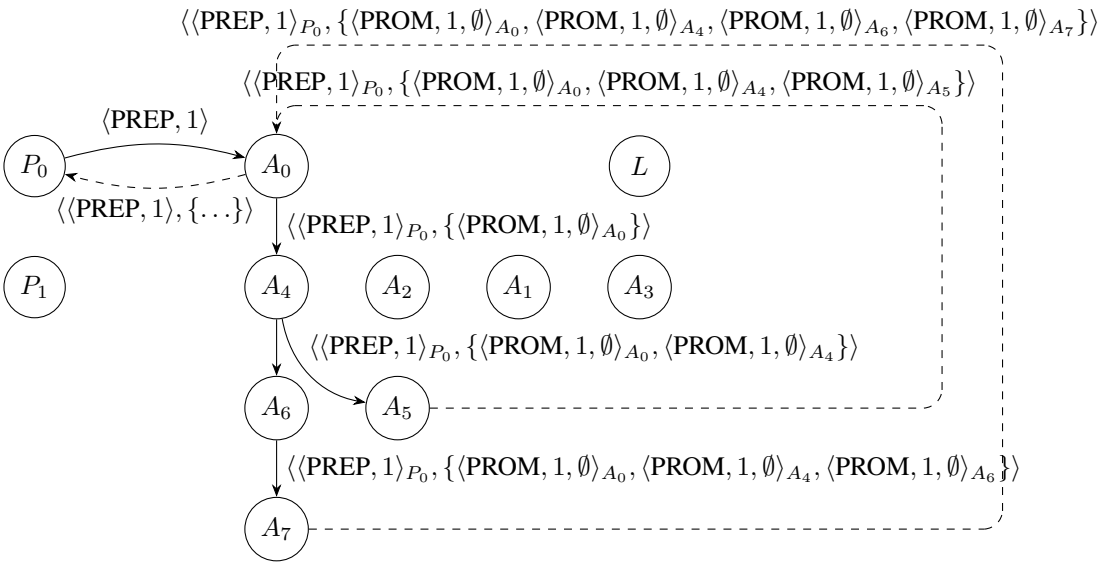


Figura 4. Difusão na fase 1 do maior *cluster* do *acceptor* 0 em um sistema com $n_a = 8$.

A Figura 4 mostra um sistema distribuído com 1 *proposer* e 8 *acceptors*. Neste exemplo, o *proposer* 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* 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*. 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 5. 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* 1 prossegue para o seu *cluster* 2,

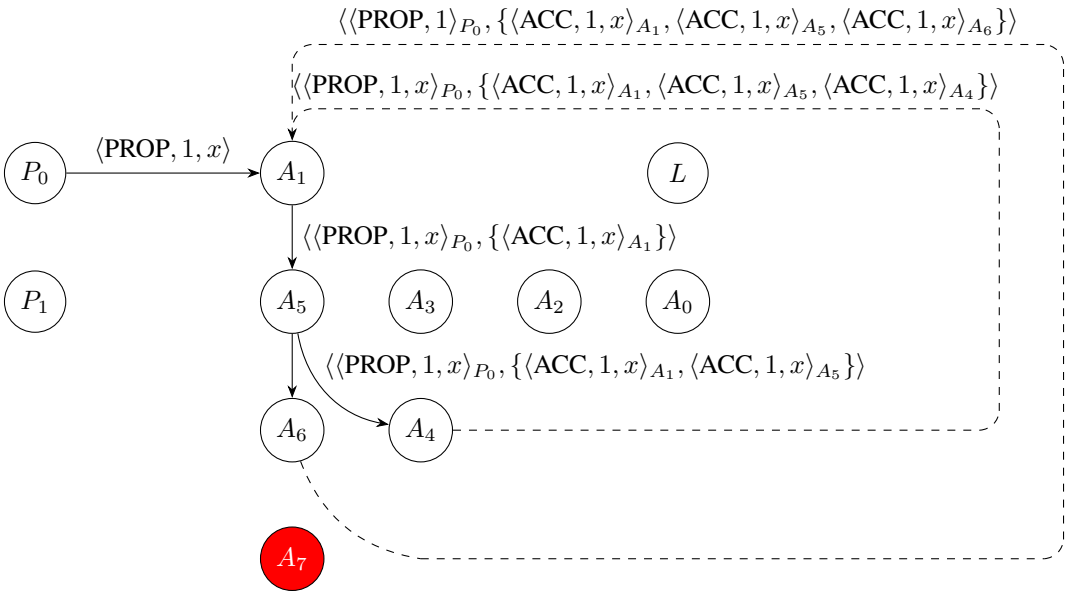


Figura 5. Difusão da fase 2 do maior *cluster* do *acceptor* 1 em um sistema com $n_a = 8$ e *acceptor* 7 falho.

enviando a proposta para o *acceptor* 3 como mostra a Figura 6. 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*.

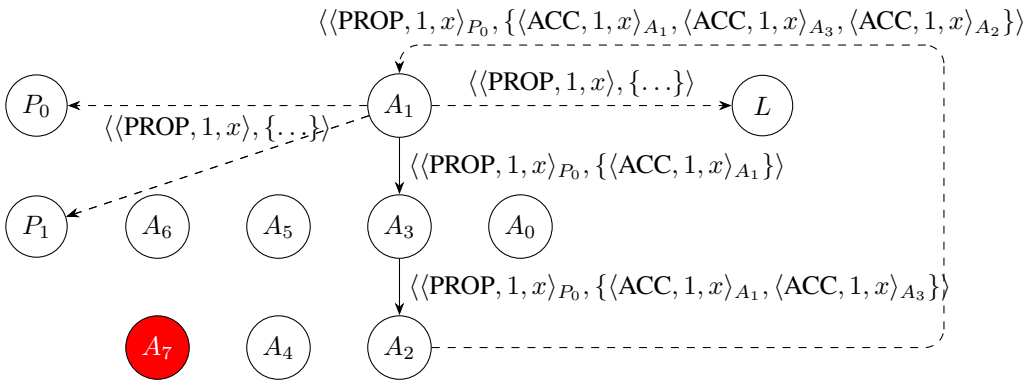


Figura 6. Difusão na fase 2 do segundo maior *cluster* do *acceptor* 1 em um sistema com $n_a = 8$ e *acceptor* 7 falho.

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

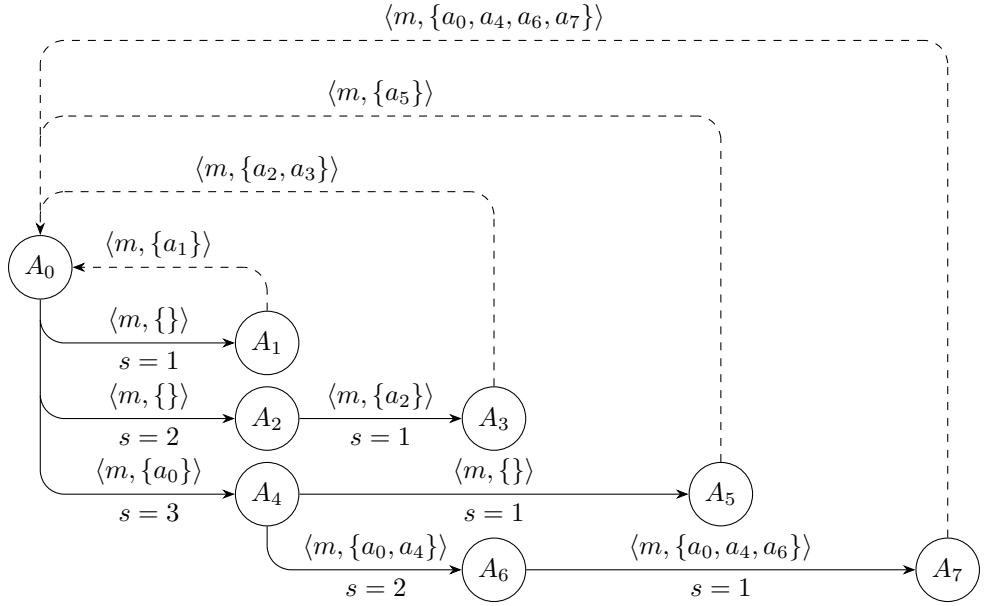


Figura 7. Difusão partindo do processo 0 em um sistema de 8 processos.

Para cada difusão, o *proposer* deve escolher um *acceptor* diferente do escolhido para difusão anterior. Isso serve para distribuir a responsabilidade de difusão entre os *acceptors* e evitar sobrecarga de um único *acceptor*. Como o *proposer* não faz parte do vCube, para evitar que o *proposer* envie mensagens para um *acceptor* falho, uma otimização possível é que o *proposer* obtenha alguma informação sobre o vCube ao conseguir se comunicar com um *acceptor* correto.

A difusão na fase 2 pode ser substituída por uma difusão mais simples, sem concatenação das respostas. No entanto, os *learners* precisam aprender sobre a decisão de outra maneira. Além disso, o *proposer* também precisa descobrir se sua proposta foi escolhida, para não precisar executar uma nova fase 1.

Como a única diferença do HyperPaxos em relação ao Paxos é a comunicação de/entre *acceptors*, o HyperPaxos herda a corretude do Paxos. Apenas um valor proposto por um *proposer* será decidido. Além disso, apenas um valor será decidido, e é aquele decidido por uma maioria de *acceptors*. E por fim, os *learners* aprendem apenas os valores decididos por uma maioria de *acceptors* na fase 2. A terminação do algoritmo é garantida apenas depois do GST e quando apenas um *proposer* está fazendo propostas [Renesse and Altinbuken 2015].

5. Implementação sobre a LibPaxos e Resultados

A LibPaxos [Primi and Sciascia 2013] é um projeto constituído de várias implementações do algoritmo Paxos. Dentre elas, se encontra a libPaxos propriamente dita, uma biblioteca escrita em C que permite a uma aplicação usar o Paxos tradicional para executar o consenso. A libPaxos conta com três versões, sendo que a terceira versão é chamada de libPaxos[3]. Nela, a comunicação é *unicast* com protocolo TCP. Nesta versão houve um claro progresso da arquitetura em relação às anteriores, em especial no sentido de que a comunicação é separada da lógica do Paxos. A biblioteca pode ser obtida em [Sciascia 2016].

Para implementar o algoritmo HyperPaxos, a terceira versão da libPaxos é usada como base. Devido a sua arquitetura, é possível acrescentar a lógica do HyperPaxos, incluindo a topologia do vCube, mas mantendo a lógica do Paxos original da libPaxos. A biblioteca implementada – chamada libHyperPaxos – está disponível em [Kiotheka and Pereira 2022].

Optou-se por fazer duas otimizações, que tiveram impacto positivo nos resultados obtidos. Além da eliminação das mensagens duplicadas na comunicação de um processo para um determinado *cluster*, mencionada na seção anterior, também é implementada uma codificação mais eficiente das respostas. Concretamente, esta codificação se dá através da inclusão de um conjunto separado de identificadores dos *acceptors* que validam o número de proposta na fase 1 ou aceitam a proposta na fase 2. 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 porque a difusão é feita hierarquicamente, sendo uma resposta sempre acompanhada da requisição correspondente.

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, o próprio *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.

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 dois parâmetros de configuração diferentes que potencializam a quantidade de valores decididos por segundo. O primeiro parâmetro se refere à quantidade de valores simultâneos enviados, chamado *outstanding*. O cliente inicialmente envia esta quantidade de valores, e ao receber a informação de decisão de um valor, submete outro. Já o segundo parâmetro se refere ao tamanho da janela de pré-execução, isto é, a quantidade de fases 1 executadas a frente, antes de ter um valor

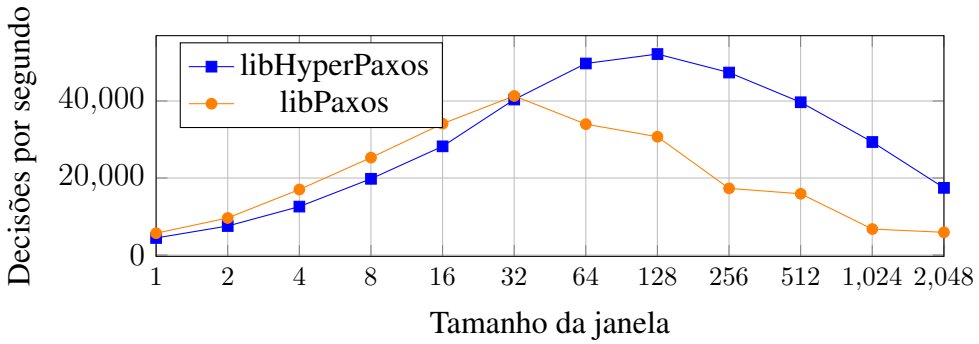


Figura 8. Tamanho da janela e decisões por segundo nas bibliotecas libHyperPaxos e libPaxos em um sistema de 4 réplicas e valores de 64 bytes.

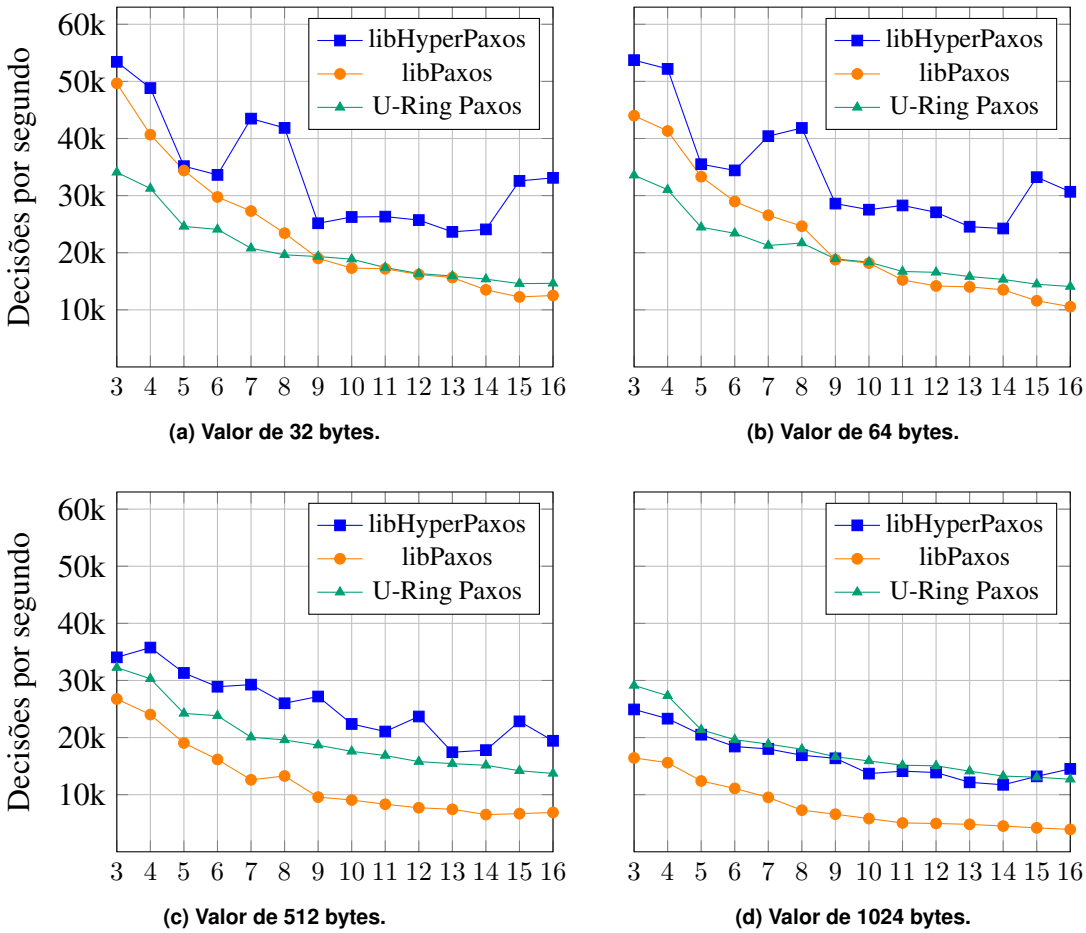
do cliente. A janela de pré-execução potencialmente acelera o tempo de execução das instâncias, já que os *proposers* estão prontos para executar a fase 2 quando recebem o valor do cliente.

No caso da libPaxos, o tamanho de janela de pré-execução ótimo encontrado foi de 32 instâncias. Na libHyperPaxos, esta janela foi de 128 instâncias. Como mostra a Figura 8, no caso da libPaxos, valores maiores ou menores que 32 para a janela de execução pioram o desempenho. É possível constatar na figura que, de forma geral, diminuir ou aumentar o tamanho da janela afeta drasticamente a quantidade de decisões por segundo obtidas. Os testes foram feitos em 4 réplicas, mas a mesma tendência se aplica a 8 e 16 réplicas. O parâmetro *outstanding* utilizado foi de 1024 valores simultâneos em todos os testes. Este parâmetro de 1024 é suficiente para saturar o sistema com valores, e aumentá-lo não influencia na quantidade de decisões por segundo.

Já na implementação do U-Ring Paxos, os parâmetros utilizados foram de uma janela de pré-execução de 32768, com a mesma quantidade de 1024 valores simultâneos em todos os testes. O U-Ring Paxos não aceita janelas de pré-execução pequenas para esta quantidade de valores simultâneos devido a questões de implementação, e nem apresenta variações nas decisões por segundo com variação do tamanho da janela. A estimativa de decisões por segundo nesta biblioteca é calculada através do envio de 1 500 000 valores do *proposer* da réplica 1 e medindo o tempo que o *learner* dessa mesma réplica demora para aprender o último valor.

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 9 apresenta os resultados dos experimentos executados. São quatro gráficos, cada com um tamanho de valor decidido diferente, variando 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 $2^k - 1$. Ainda, pode-se notar que a libHyperPaxos teve uma quantidade maior de valores decididos por



(a) Valor de 32 bytes. **(b) Valor de 64 bytes.**

(c) Valor de 512 bytes. **(d) Valor de 1024 bytes.**

Figura 9. Valores decididos por segundo em relação ao número de réplicas.

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.

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 libHyerPaxos, é esperado que numa próxima versão da biblioteca, o desempenho obtido seja igual em todos os tamanhos de valor.

6. Conclusão

Este trabalho apresentou o algoritmo de consenso HyperPaxos, uma versão hierárquica do algoritmo Multi-Paxos. O HyperPaxos organiza os *acceptors* em *clusters*, usando o vCube, uma topologia hierárquica virtual escalável e com propriedades logarítmicas. 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 e acelerando o consenso.

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 apresentaram bom desempenho da libHyperPaxos em todos os testes em relação à libPaxos, e em valores de tamanho menor que 1024 bytes quando comparado ao U-Ring Paxos.

Trabalhos futuros incluem, a curto prazo, otimizar a implementação da biblioteca, com destaque para suporte a valores maiores. A longo prazo, a implementação de um sistema de replicação máquina de estados baseado na libHyperPaxos.

Referências

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

Bona, L. C., Duarte Jr, E. P., Mello, S. L., and Fonseca, K. V. (2006). Hyperbone: Uma rede overlay baseada em hipercubo virtual sobre a internet. *XXIV SBRC*.

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

Burrows, M. (2006). The Chubby lock service for loosely-coupled distributed systems. In *7th USENIX*, pages 335–350.

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

De Araujo, J. P., Arantes, L., Duarte, E. P., Rodrigues, L. A., and Sens, P. (2017). A publish/subscribe system using causal broadcast over dynamically built spanning trees. In *2017 29th SBAC-PAD*, pages 161–168. IEEE.

de Araujo, J. P., Arantes, L., Duarte, E. P., Rodrigues, L. A., and Sens, P. (2017). A publish/subscribe system using causal broadcast over dynamically built spanning trees. In *29th SBAC-PAD*, pages 161–168.

Duarte, E. P., Rodrigues, L. A., Camargo, E. T., and Turchetti, R. (2022). A Distributed System-level Diagnosis Model for the Implementation of Unreliable Failure Detectors. *CoRR*, abs/2210.02847.

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.

Ellis, J. (2013). Lightweight transactions in Cassandra 2.0. `https://www.datastax.com/blog/lightweight-transactions-cassandra-20`.

Google (2023). Replication | Cloud Spanner | Google Cloud. https://cloud.google.com/spanner/docs/replication.

Hurfin, M., Mostefaoui, A., and Raynal, M. (1998). Consensus in asynchronous systems where processes can crash and recover. In *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. (2022). HyperPaxos / LibHyperPaxos - GitLab. https://gitlab.c3sl.ufpr.br/hyperpaxos/libhyperpaxos.

Lamport, L. (1978). The implementation of reliable distributed multiprocess systems. *Computer Networks (1976)*, 2(2):95–114.

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., Cohen, J., Arantes, L., and Duarte, E. P. (2013a). A robust permission-based hierarchical distributed k-mutual exclusion algorithm. In *2013 IEEE 12th International Symposium on Parallel and Distributed Computing*, pages 151–158. IEEE.

Rodrigues, L. A., Cohen, J., Arantes, L., and Duarte Jr, E. P. (2013b). A Robust Permission-Based Hierarchical Distributed k-Mutual Exclusion Algorithm. In *12th ISPDC*, pages 151–158.

Rodrigues, L. A., Duarte Jr, E. P., and Arantes, L. (2014). Árvores geradoras mínimas distribuídas e autonômicas. *XXXII SBRC*, 2014:1–14.

Rodrigues, L. A., Jeanneau, D., Duarte Jr, E. P., and Arantes, L. (2017). Uma Solução de Difusão Confiável Hierárquica em Sistemas Distribuídos Assíncronos. In *XXXV SBRC*.

Sciascia, D. (2016). sciascid / libpaxos — Bitbucket. https://bitbucket.org/sciascid/libpaxos/.

Terra, A. d. C., Camargo, E. T. d., and Duarte Jr, E. P. (2020). A Caminho de Uma Alternativa Hierárquica para Implementação do Algoritmo de Consenso Paxos. In *XXI WTF*, pages 15–28.

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 *2nd PDSW*, pages 35–44.