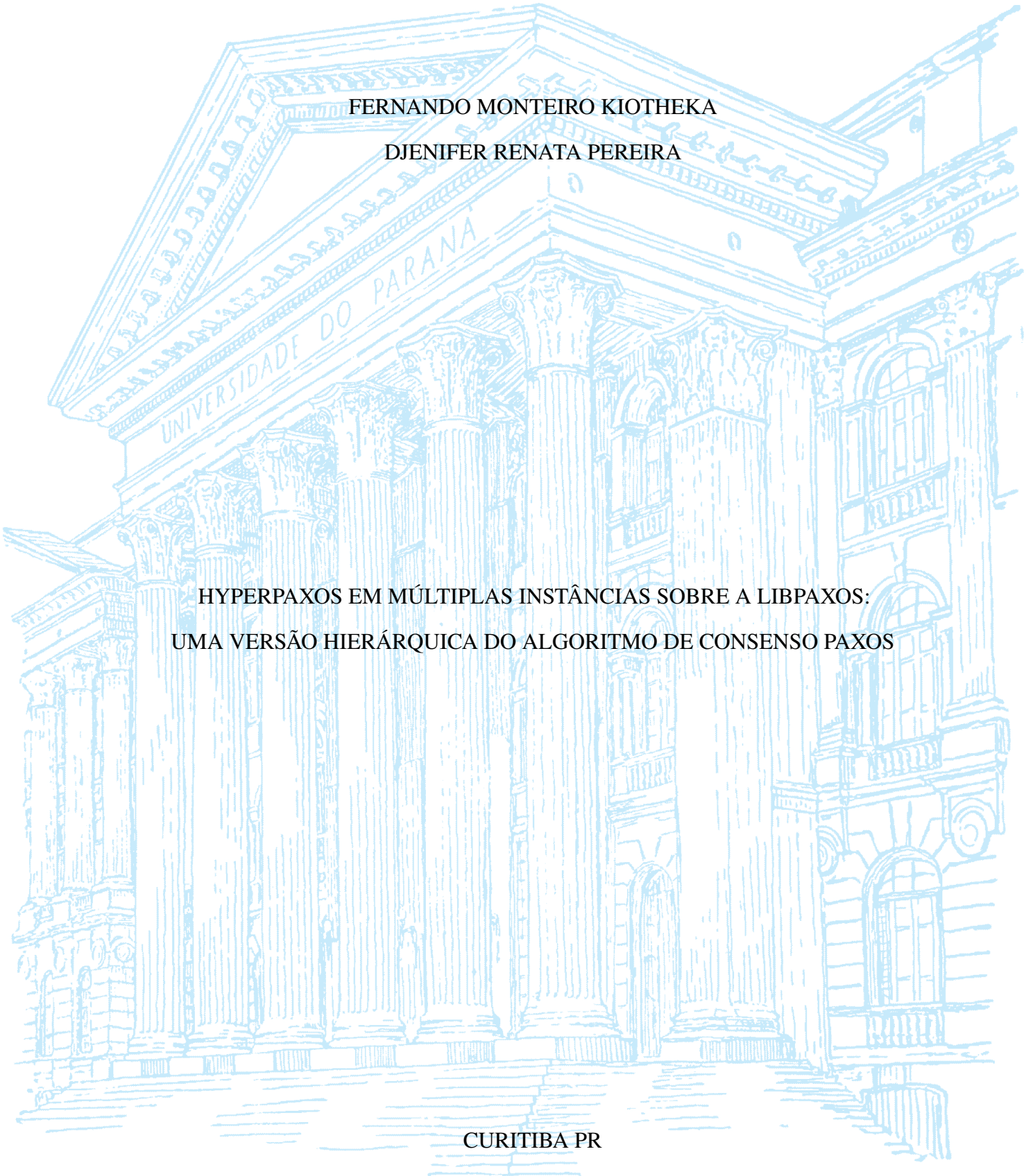
UNIVERSIDADE FEDERAL DO PARANÁ

FERNANDO MONTEIRO KIOTHEKA

DJENIFER RENATA PEREIRA



HYPERPAXOS EM MÚLTIPLAS INSTÂNCIAS SOBRE A LIBPAXOS: UMA VERSÃO HIERÁRQUICA DO ALGORITMO DE CONSENSO PAXOS

CURITIBA PR

2022

FERNANDO MONTEIRO KIOTHEKA
DJENIFER RENATA PEREIRA

HYPERPAXOS EM MÚLTIPLAS INSTÂNCIAS SOBRE A LIBPAXOS: UMA VERSÃO HIERÁRQUICA DO ALGORITMO DE CONSENSO PAXOS

Trabalho apresentado como requisito parcial à conclusão do Curso de Bacharelado em Ciência da Computação, Setor de Ciências Exatas, da Universidade Federal do Paraná.

Área de concentração: *Ciência da Computação*.

Orientador: Elias Procópio Duarte Júnior.

CURITIBA PR

2022

RESUMO

O consenso é um problema fundamental de sistemas distribuídos. Informalmente, o objetivo é fazer com que os processos corretos de um sistema decidam por um mesmo valor, a partir de um ou mais valores propostos inicialmente. Além disso, o consenso deve ser tolerante a falhas, permitindo que alguns processos falhem sem afetar a decisão. Um dos algoritmos que resolve o problema do consenso, sendo tolerante a falhas, é o Paxos. No Paxos, os processos assumem papéis que podem ser *proposer*, *acceptor* e *learner*. Um *proposer* propõe valores para o consenso, os *acceptors* decidem por um valor e os *learners* aprendem o valor decidido. O Paxos consiste em duas fases, sendo a primeira fase para preparar um número de proposta e a segunda para propor um valor com o número obtido na fase anterior. O Paxos assume o modelo de falhas *crash-recovery*, isto é, os processos podem parar e voltar ao sistema com informações de seu estado antes da falha. Neste trabalho é proposto o algoritmo HyperPaxos, uma versão do algoritmo Paxos sobre o vCube para múltiplas instâncias. O vCube é uma topologia virtual que organiza os processos em *clusters*, formando um hipercubo quando o número de processos é uma potência de dois e todos os processos estão corretos. Na falha de processos, essa topologia se reorganiza mantendo diversas propriedades logarítmicas. O algoritmo HyperPaxos organiza os *acceptors* em *clusters* e os *proposers* enviam suas mensagens para um *acceptor* dito difusor que fará a retransmissão para os demais *acceptors* usando a difusão sobre o vCube. A difusão das mensagens ocorre também em duas fases, como no Paxos, porém ocorrendo de forma hierárquica. Cada *acceptor* ao receber uma mensagem, a encaminha para o seu próximo *cluster*, até chegar nas folhas da árvore de difusão. Conforme um *acceptor* encaminha a mensagem para outros *acceptors*, ele concatena sua resposta ao *proposer* na mensagem. Os *acceptors* folhas retornam a mensagem para o *acceptor* que começou a retransmissão no vCube. Inicialmente, o *acceptor* difusor envia a mensagem para o seu maior *cluster* na tentativa de conseguir uma maioria para a fase 1 ou 2. Caso consiga, o *acceptor* responde ao *proposer*. Se não conseguir, continua a difusão para seus próximos *clusters*, do maior para o menor. O algoritmo HyperPaxos foi implementado como a biblioteca libHyperPaxos e comparada com a terceira versão da libPaxos. Resultados obtidos mostram o bom desempenho da libHyperPaxos, que supera inclusive a libPaxos em termos do número de decisões por segundo.

Palavras-chave: Consenso. Paxos. vCube. LibPaxos.

LISTA DE FIGURAS

SUMÁRIO

1 INTRODUÇÃO

O uso de sistemas computacionais se tornou parte integral da sociedade. Desta forma, falhas nesses sistemas causam prejuízos em vários aspectos (Lerner, 2014; Baker e Gates, 2019). Uma das técnicas para manter um serviço em funcionamento mesmo em caso de falhas de componentes é a replicação: ter várias cópias de um mesmo recurso disponíveis, que podem ser usadas caso alguns componentes falhem. A replicação também permite que vários componentes possam funcionar simultaneamente, aumentando o desempenho e a quantidade de usuários que o serviço suporta. O problema então se torna garantir a consistência do sistema, isto é, coordenar os componentes replicados para que todos se mantenham idênticos (Charron-Bost et al., 2010). É neste contexto que se apresenta o problema do consenso em sistemas distribuídos.

Sistemas distribuídos são aqueles compostos por dois ou mais processos que se comunicam e colaboram para realizar uma tarefa (Raynal, 2005). No problema do consenso, os processos propõem valores, e todos os processos devem chegar em um acordo para escolher um determinado valor proposto. Isso se torna desafiador quando o sistema deve ser tolerante a falhas, no qual processos podem falhar a qualquer momento e o sistema precisa continuar funcionando. Assumimos neste caso as falhas por parada (*crash*), em que processos param de se comunicar quando falham.

Um dos algoritmos que resolve o problema do consenso é o Paxos. O Paxos é um algoritmo proposto por Lamport (1998) que possibilita o consenso. Ele se encontra hoje em uso em múltiplas aplicações relevantes, incluindo serviços do Google (Burrows, 2006; Brewer, 2017; Google, 2022), no módulo RADOS do sistema de arquivos distribuído Ceph (Weil et al., 2007), entre vários outros. No Paxos, os processos assumem papéis, que podem ser: *proposer*, *acceptor* e *learner*. Um *proposer* propõe valores para o consenso, 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, 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, várias variantes e otimizações foram desenvolvidas para esse algoritmo (Regis e Mendizabal, 2022). Entre as variantes podemos encontrar, por exemplo, o Fast Paxos (Lamport, 2006) que foca em reduzir a latência e o Cheap Paxos (Lamport e Massa, 2004) que reconfigura o sistema em caso de falhas para reduzir a quantidade de processos corretos necessários. Já com a proposta de aumentar a vazão, podemos encontrar o Ring Paxos (Jalili Marandi et al., 2017), que utiliza uma topologia em anel. E Charapko et al. (2021) propõem utilizar processos retransmissores escolhidos aleatoriamente para aliviar o gargalo da transmissão de várias mensagens do *proposer*.

O presente trabalho propõe uma nova versão do algoritmo Paxos baseada em uma topologia distribuída hierárquica, o vCube (Duarte Jr. et al., 2022). O vCube é um algoritmo de detecção de falhas, que foi inicialmente proposto em Duarte Jr e Nanya (1998), no contexto de diagnóstico distribuído. 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, o que permite que o algoritmo seja escalável. Na falta ou na falha de processos, essa topologia se reorganiza mantendo as propriedades logarítmicas.

Usando o vCube enquanto topologia virtual, Rodrigues et al. (2014) propuseram alguns algoritmos de difusão de mensagens, sendo eles uma difusão de melhor esforço e uma difusão confiável. Os algoritmos criam uma árvore geradora autonômica, de forma que a origem manda mensagem para o primeiro processo correto de cada *cluster*, e os processos subsequentes mandam mensagens para os primeiros processos corretos dos seus *clusters* de tamanho menor. Quando processos falham, a árvore muda e as mensagens enviadas a processo falhos são retransmitidas para processos corretos.

Em Terra (2020), é proposta uma variante do Paxos para uma instância usando a topologia do vCube. No algoritmo, os *acceptors* são organizados em *clusters* e o coordenador, um processo que desempenha papel de *proposer* e de *acceptor*, utiliza a difusão de melhor esforço sobre o vCube com n processos. O coordenador transmite suas requisições para o maior *cluster* de *acceptors* e espera pelas suas respostas. Sem falhas e com o número de *acceptors* sendo uma potência de dois, uma maioria recebe a mensagem, pois o coordenador é um *acceptor* e o maior *cluster* tem tamanho $n/2$. Em caso de falhas, o coordenador continua enviando para *clusters* menores até atingir uma maioria. Ao receber uma requisição, o *acceptor* concatena sua resposta à mensagem e a encaminha para o próximo *acceptor* na árvore. Caso o *acceptor* seja uma folha na árvore, as respostas são devolvidas ao coordenador. A partir das respostas, o coordenador verifica se conseguiu uma maioria de propostas aceitas para atingir o consenso.

Neste trabalho, expandimos a variante do Paxos sobre o vCube de Terra (2020) propondo o algoritmo de consenso HyperPaxos, e apresentando sua implementação sobre a LibPaxos (Primi e Sciascia, 2013). O HyperPaxos inclui a execução para múltiplas instâncias enquanto Terra (2020) descreve a execução de uma única instância. Assim como no trabalho original, os *acceptors* são organizados em *clusters*, no entanto, o processo coordenador não existe. O *proposer* faz as requisições para um *acceptor* 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*.

Além do algoritmo HyperPaxos, esse trabalho apresenta a implementação de duas bibliotecas, a libvCube e a libHyperPaxos. A libvCube é uma implementação do algoritmo de detecção de falhas vCube e a libHyperPaxos é uma implementação do algoritmo HyperPaxos, ambas baseadas na biblioteca libPaxos. Ao final, a libHyperPaxos é comparada com a libPaxos em relação à quantidade de valores decididos por segundo, obtendo resultados positivos para esta métrica.

O restante desse trabalho está organizado da seguinte forma. O Capítulo 2 apresenta o algoritmo Paxos, que resolve o problema do consenso com tolerância a falhas. Já o Capítulo 3 apresenta a topologia hierárquica virtual vCube. Segue no Capítulo 4 a descrição da difusão de melhor esforço sobre o vCube. No Capítulo 5 encontramos a definição do algoritmo HyperPaxos. E no Capítulo 6 é apresentada a biblioteca LibPaxos, bem como as implementações do vCube e do HyperPaxos na forma das bibliotecas libvCube e libHyperPaxos. As conclusões seguem no Capítulo 7.

2 PAXOS: CONSENSO TOLERANTE A FALHAS

O consenso é um problema fundamental de sistemas distribuídos em que processos propõem valores, e dentre eles, todos os processos corretos chegam em um acordo e escolhem o mesmo valor. Em geral, este valor representa um comando ou uma transição de máquina de estado que pode ser usado para sincronizar processos. Mais especificamente, um algoritmo que resolve o consenso precisa garantir que apenas um valor que foi efetivamente proposto anteriormente seja escolhido. Além disso, o consenso também garante que apenas um único valor é escolhido, sempre garantindo o acordo. E que após um intervalo de tempo finito, um valor é sempre escolhido (Cachin et al., 2011).

O Paxos é um algoritmo para o problema do consenso proposto em Lamport (1998). Na proposta original, o Paxos é proposto dentro de uma analogia a um sistema legislativo fictício. No entanto, aquele artigo foi considerado de compreensão difícil, o que motivou Lamport a lançar uma descrição simplificada anos mais tarde (Lamport, 2001).

Os processos que executam o algoritmo desempenham três papéis: *proposers*, *acceptors* e *learners*. *Proposers* propõem valores ao sistema, sendo os valores enviados por clientes externos. *Acceptors* aceitam ou rejeitam propostas feitas pelos *proposers*. *Learners* aprendem os valores decididos pelos *acceptors*. Um processo pode exercer mais de um papel, sendo o mapeamento entre papéis e processos irrelevante para a corretude do algoritmo. Porém, o número de processos e seus papéis deve ser estático, isto é, eles não mudam durante a execução do algoritmo. Variações do Paxos como o Cheap Paxos (Lamport e Massa, 2004) permitem que processos entrem e saiam durante a execução do algoritmo conforme necessário.

O algoritmo Paxos assume o modelo de falha *crash-recovery*, isto é, um processo pode falhar e voltar ao sistema. Um processo falho é aquele que para de enviar mensagens, e pode voltar ao sistema, recuperando o estado anterior à falha (Hurfin et al., 1998). Além disso, o Paxos assume um modelo temporal assíncrono, ou seja, não existem garantias quanto ao tempo de processamento ou de entrega de mensagens.

O problema do consenso pode ser resolvido usando apenas um processo que escolhe valores. No entanto, esse processo se torna um ponto crítico que não pode falhar. Outra solução simples é sempre escolher o mesmo valor, porém isso não é interessante para aplicações reais. Então, se faz necessário o uso de um algoritmo mais robusto, tolerante a falhas.

Diferente dos algoritmos triviais descritos no parágrafo anterior para resolver o consenso, o Paxos é tolerante a f falhas em um sistema com $2f + 1$ *acceptors*. Ao menos um *proposer* também é necessário para que novos valores sejam propostos. Além disso, vários *proposers* podem propor valores simultaneamente sem comprometer a corretude.

Para decidir cada valor, devem ser executadas duas fases. Na primeira fase, o *proposer* prepara um número de proposta, e caso obtenha sucesso, prossegue para a segunda fase fazendo uma proposta com um valor. Estas duas fases são descritas com mais detalhes a seguir.

2.1 FASE 1: PREPARAÇÃO

O *proposer* procura nesta fase escolher um número de proposta válido. Um valor só é realmente proposto na próxima fase. Para tal, o *proposer* escolhe um número de proposta b (de *ballot*, como em Lamport (1998)) que não pode ser usado por nenhum outro *proposer*, e envia um pedido de preparação para um conjunto de *acceptors*. Cada *acceptor* pode:

- Responder, prometendo que vai ignorar números de propostas menores que b. Caso ele já tenha aceitado outra proposta na fase 2, tanto o número quanto o valor da proposta já aceita também são enviados como resposta.

- Ignorar o pedido, pois o número de proposta enviado é obsoleto, e ele já prometeu para algum *proposer* que não iria aceitar tais propostas.

Desta forma, o *proposer* espera nesta fase 1 que uma maioria de *acceptors* responda com uma promessa, validando o seu número de proposta. Caso isso aconteça, o *proposer* prossegue para a fase 2 com este número de proposta.

O valor da proposta depende do *proposer* ter recebido algum valor nas respostas da fase 1. Se sim, então o valor da proposta será aquele da proposta de maior número. Caso contrário, o *proposer* está livre para propor um valor qualquer.

Se não houver uma maioria de *acceptors* prometendo que o número da proposta é válido, o *proposer* precisa tentar novamente, com um número de proposta maior. Assim, para agilizar o processo de troca de número de proposta, o *acceptor* pode avisar ao *proposer* que o número de proposta recebido é obsoleto.

Note que obter um número de proposta que é único entre todos os *proposers* não precisa de comunicação. Basta que cada *proposer* use números de proposta da forma $kn_p + p$, onde n_p é o número de *proposers*, p é o identificador do *proposer* e k um inteiro que pode ser incrementado.

2.2 FASE 2: PROPOSTA

Nesta fase, o *proposer* faz a proposta com valor. Então, ele manda a sua proposta para um conjunto de *acceptors*. Cada *acceptor* pode:

- Aceitar a proposta e prometer que vai ignorar números de propostas menores que o número da proposta enviada. Isto é necessário, pois o Paxos permite que as mensagens cheguem em qualquer ordem (Renesse e Altinbuken, 2015) e o conjunto de *acceptors* utilizado na fase 1 pode não ser o mesmo conjunto da fase 2 (Lamport, 2001).

- Ignorar a proposta, pois o número de proposta enviado é obsoleto, e ele já prometeu para algum *proposer* que não iria aceitar tais propostas.

Quando uma maioria de *acceptors* aceita uma proposta, então há consenso sobre a escolha do valor. Este valor não muda, mesmo que outro *proposer* comece a fase 1 com um número de proposta maior, pois o valor escolhido aparecerá na resposta de pelo menos um *acceptor*, porém o número da proposta poderá aumentar.

Para saber que valor foi decidido, os *learners* precisam descobrir que uma proposta foi aceita por uma maioria de *acceptors*. Isso pode ser feito de várias maneiras como, por exemplo, cada *acceptor* informar todos os *learners* que aceitou uma proposta.

Note que os *proposers* não sabem se o seu valor foi escolhido. Um jeito de descobrirem é executando a fase 1 novamente. Obtendo sucesso, recebem como resultado o seu próprio valor em pelo menos uma resposta. Assim, outra otimização válida é o *acceptor* avisar o *proposer* que a sua proposta foi aceita ou ignorada.

2.3 EXEMPLO DE EXECUÇÃO

A Figura 2.1 mostra um sistema distribuído com dois *proposers*, P_1 e P_2 e três *acceptors*, A_1, A_2 e A_3. Inicialmente, o *proposer* P_1 propõe o valor x e P_2 propõe o valor y. Então P_1

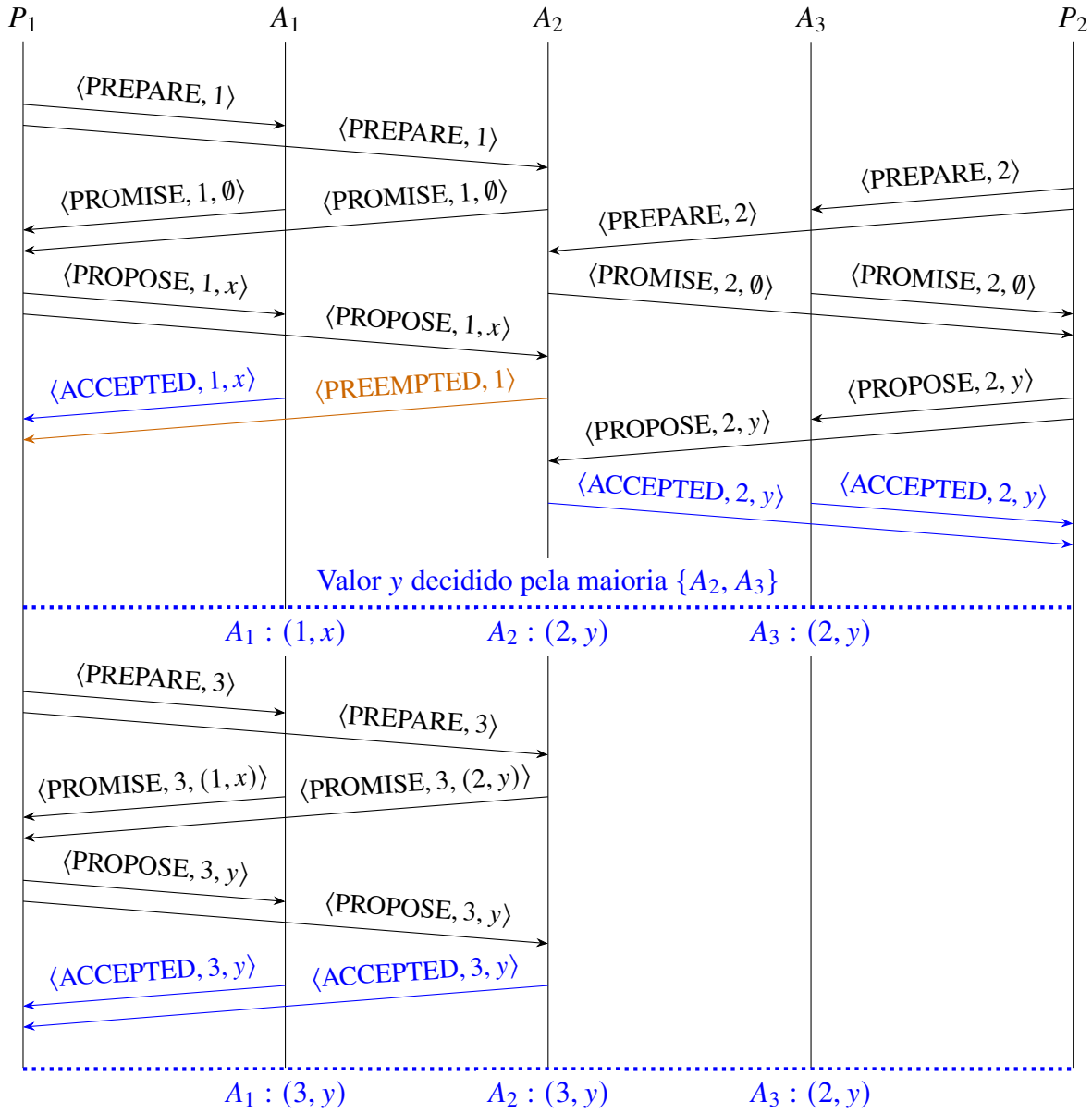


Figura 2.1: Exemplo de execução do Paxos.

inicia a fase 1 com número de proposta 1 e manda pedidos de preparação PREPARE para A_1 e A_2. Estes *acceptors* não fizeram promessas até o momento, então eles prometem que não irão aceitar propostas de número menor que 1 com um PROMISE.

Em seguida, P_2 inicia a fase 1 com número de proposta 2, e manda pedidos de preparação para A_2 e A_3. Ambos os *acceptors* prometem para P_2 que não irão aceitar propostas de número menor que 2. Em seguida, P_1 decide iniciar sua fase 2, e manda sua proposta PROPOSE de número 1 e valor x para os *acceptors* A_1 e A_2. A_1 aceita a proposta com um ACCEPTED, porém, A_2 já havia prometido para P_2 que não aceitaria propostas com número inferior a 2, e rejeita a proposta enviada por P_1 com um PREEMPTED. Sem maioria, P_1 precisa voltar para a fase 1 com um número maior de proposta.

Enquanto isso, P_2 executa a sua fase 2 com número de proposta 2 enviando propostas para A_2 e A_3. Ambos aceitam, resultando no consenso, com a escolha do valor y, pois há uma maioria de *acceptors* que aceitou propostas de valor y.

O *proposer* P_1 aumenta o seu número de proposta para 3 e inicia a fase 1 novamente, enviando pedidos de preparação para A_1 e A_2. Como resposta, ele recebe a promessa de ambos de não aceitar proposta de número menor que 3, além das propostas que eles já aceitaram. Entre as propostas aceitas, P_1 verifica que a proposta de maior valor, a proposta de número 2, possui valor y, então P_1 deve propor este valor na fase 2. Por fim, P_1 propõe para A_1 e A_2 uma proposta com número 3 e valor y. Ambos os *acceptors* aceitam a proposta, e todos os *acceptors* agora aceitaram uma proposta com o mesmo valor y, ainda que com números de proposta diferentes.

2.4 MÚLTIPLOS VALORES

Até o momento, descrevemos o algoritmo do Paxos que permite a decisão de apenas um único valor. No entanto, o Paxos consegue decidir múltiplos valores de forma simultânea, executados em múltiplas instâncias consecutivas. Cada instância decide um valor diferente, sendo composta por um identificador próprio e o estado do algoritmo. O estado do algoritmo é composto de informações necessárias para o funcionamento de uma instância do Paxos como as promessas feitas e as propostas aceitas no caso de um *acceptor*. Todas as mensagens trocadas devem também incluir a identificação da instância.

Cada instância executa paralelamente em relação às demais. Por não existir uma coordenação, é possível que uma instância recém-criada atinja consenso antes de instâncias que já existiam. Assim, a aplicação deve ordenar os valores decididos, o que é feito normalmente conforme o identificador da instância.

2.5 RECUPERAÇÃO DE PROCESSOS

Como o modelo de falhas do Paxos é *crash-recovery*, os processos utilizam memória secundária para recuperar seus estados. Mas nem todos os processos precisam recuperar o estado anterior à falha. Para satisfazer as propriedades do consenso, somente é preciso que os *acceptors* mantenham as propostas aceitas. Isto porque, os *learners* e os *proposers* podem aprender o que já foi escolhido executando o Paxos novamente.

2.6 PROGRESSO

Considere um cenário como mostra a Figura 2.2 onde dois *proposers* se bloqueiam, se alternando na execução das suas fases: um *proposer* P_1 completa a fase 1 com um número de proposta n_1, e o outro *proposer* P_2 também completa a fase 1 com um número de proposta n_2, onde $n_2 > n_1$. O *proposer* P_1 então tenta executar a fase 2, mas não obtém uma maioria de respostas, pois os *acceptors* prometeram ao *proposer* P_2 que não aceitariam números de proposta menores que n_2. Assim, o *proposer* P_1 inicia e completa uma nova fase 1 para um novo número de proposta n_3, onde $n_3 > n_2$. Isto faz com que a fase 2 do *proposer* P_2 também falhe, e assim por diante.

Esse cenário de "pingue-pongue" pode ser evitado elegendo um único *proposer* como sendo o único responsável por fazer propostas. Este *proposer* é comumente chamado de coordenador ou líder. O processo de eleição de líder não faz parte do Paxos, mas é comum nas implementações.

Particularmente, em Renesse e Altinbuken (2015) se propõe que quando um *proposer* descobre que existe outro *proposer* com um número de proposta maior (por meio de uma resposta na fase 1 ou na fase 2), ele pare de enviar propostas e fique monitorando este *proposer*. Se este *proposer* monitorado falhar, o primeiro *proposer* pode começar a enviar propostas novamente.

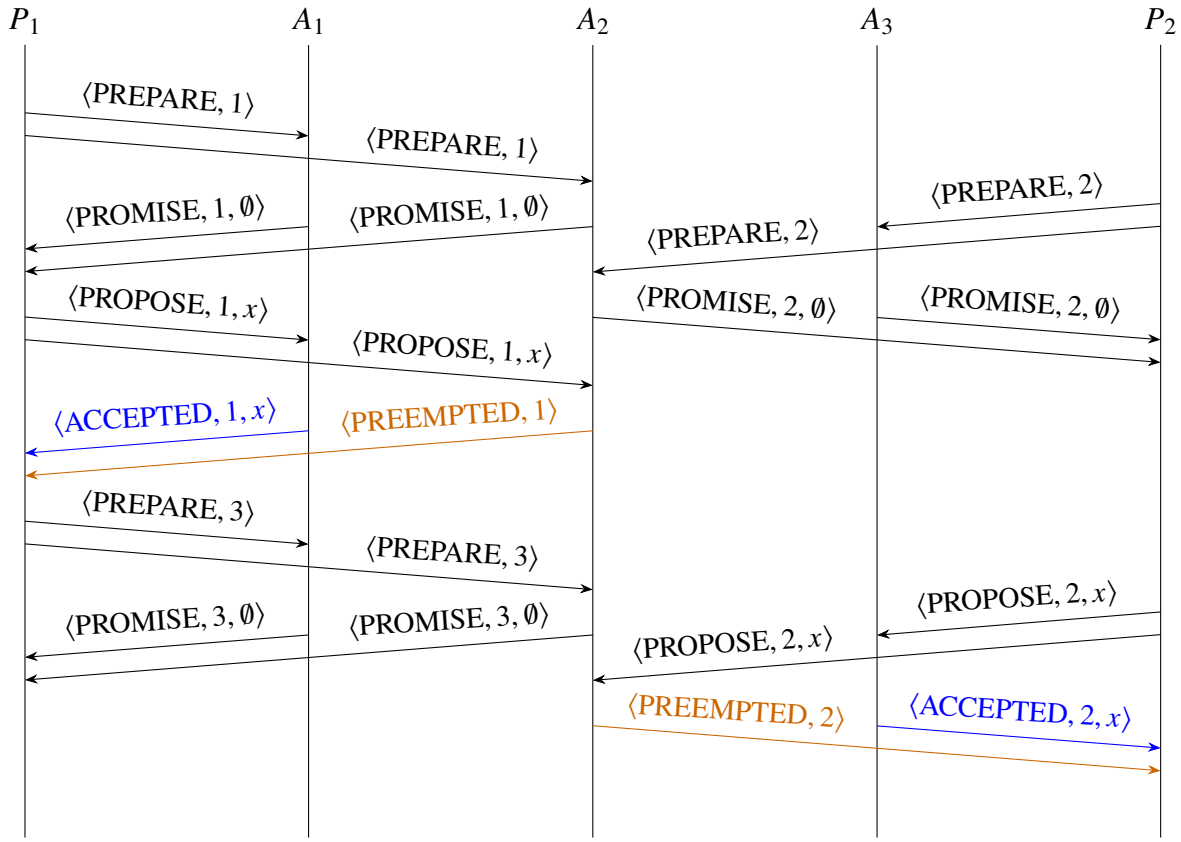
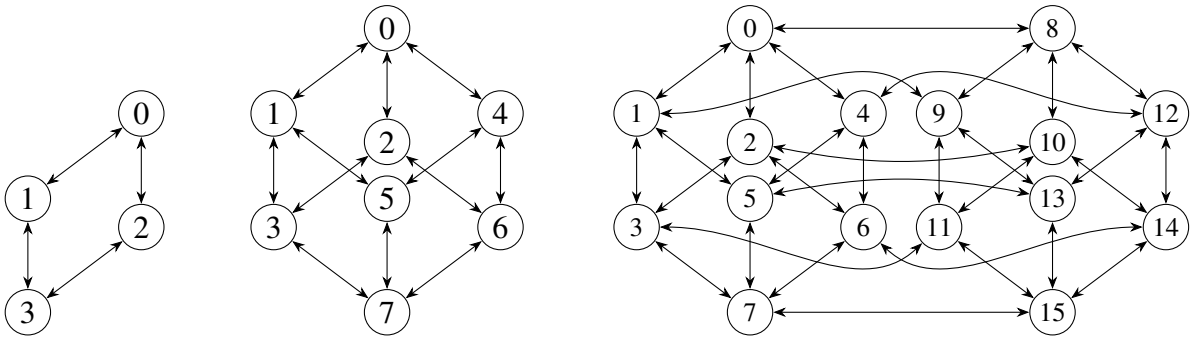


Figura 2.2: Cenário de “pingue-pongue”.

Essas duas soluções resolvem este problema do Paxos, mas só mudam o problema de lugar, pois mesmo com elas, o Paxos não tem terminação garantida. Isso ocorre em todo algoritmo de consenso para sistemas assíncronos sujeitos a falhas por parada, como provado por Fischer et al. (1985). Um dos fatores principais é que em um sistema assíncrono, não é possível diferenciar um processo falho de um extremamente lento. O resultado é que o Paxos garante todas as propriedades de um algoritmo de consenso menos a de que um valor é decidido depois de algum tempo finito.

3 TOPOLOGIA HIERÁRQUICA VCUBE

O vCube é um detector de falhas distribuído que mantém uma topologia virtual hierárquica (Duarte Jr. et al., 2022). O vCube foi originalmente proposto como um algoritmo de diagnóstico adaptativo e distribuído (Duarte Jr e Nanya, 1998; Duarte Jr et al., 2014). Quando todos os processos estão corretos e o número de processos é uma potência de dois, a topologia criada é de um hipercubo perfeito como mostra a Figura 3.1. O hipercubo possui simetria e diâmetro logarítmico, o que garante que o algoritmo é escalável. Conforme os processos vão falhando ou recuperando, o algoritmo reorganiza esta topologia, mas preservando suas propriedades logarítmicas.



(a) 4 processos (b) 8 processos (c) 16 processos

Figura 3.1: Hipercubo formado por 4, 8 e 16 processos.

Detectores de falha se baseiam em testes periódicos acontecendo em intervalos determinados pelo relógio local de cada processo. Os processos não são sincronizados entre si, portanto os intervalos de teste podem diferir entre processos. Uma rodada de teste ocorre quando todo conjunto de testes assinalados para todos os processos corretos são executados. Como o vCube é um algoritmo adaptativo, os conjuntos de testes são dinâmicos e mudam conforme os resultados de testes anteriores.

O vCube assume o modelo de falha *crash*, mas pode ser estendido para o modelo *crash-recovery* (Duarte Jr. 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, quando não há garantia de tempo, pode-se afirmar apenas que processos são corretos ou suspeitos de estarem falhos.

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. Para isso, é empregada uma estratégia hierárquica de teste, 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.

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 3.1: Valores de $c(i,s)$ para 8 processos

Por exemplo, para um sistema com 8 processos, a Tabela 3.1 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 3.2 que é uma nova representação da Figura 3.1(b). 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.

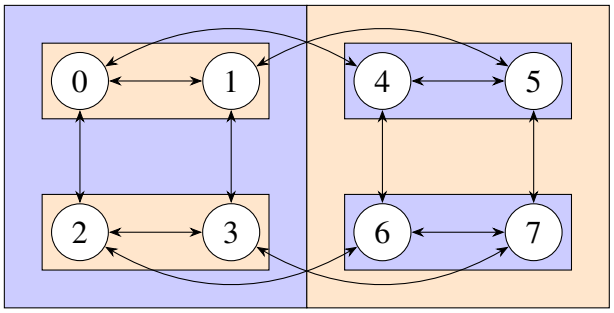


Figura 3.2: vCube formado por 8 processos.

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 3.3. 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.

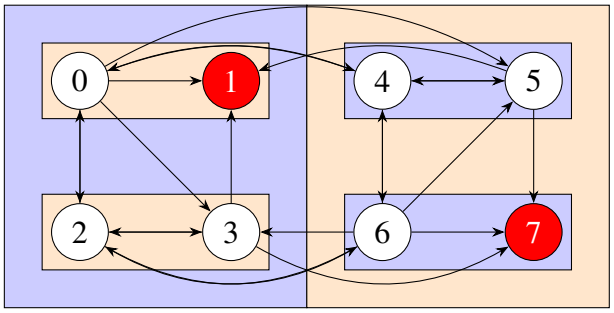


Figura 3.3: vCube formado por 8 processos sendo 1 e 7 falhos.

Cada processo mantém informações sobre o estado de todos os processos localmente. O estado é mantido como um *timestamp*: ao testar um processo correto que antes foi testado como falho ou vice-versa, o *timestamp* deste processo é incrementado. Inicialmente o *timestamp* é igual a -1, indicando que não se tem informações sobre o estado do processo. Se no primeiro teste, o processo é testado correto, o *timestamp* recebe o valor 0, caso contrário recebe o valor

1. Assim, a paridade do *timestamp* indica se o processo está correto ou não. Definimos como processos corretos aqueles com paridade par e os suspeitos de estarem falhos como ímpar.

Quando um processo correto testa outro processo correto, ele obtém informações sobre os outros processos com *timestamps* diferentes de -1, e só são obtidas informações novas. Para determinar quais informações são novas, os *timestamps* recebidos são comparados com os locais um a um. Se o *timestamp* recebido for maior, ele é copiado. Caso contrário, mantém-se o *timestamp* local. Assim, os processos aprendem sobre mudanças dos estados de todos os processos, sem precisar testar todos diretamente.

4 DIFUSÃO DE MELHOR ESFORÇO SOBRE O VCUBE

A difusão (*broadcast*) é um tipo de transmissão de mensagens em que um processo origem (neste trabalho também chamado de emissor ou difusor), 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., 2013).

Existem diversos tipos de difusão, cada uma atendendo propriedades específicas. Definimos informalmente como difusão de melhor esforço aquela que garante que se o processo difusor 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).

Um algoritmo simples que atende a especificação da difusão de melhor esforço segue. O difusor envia uma mensagem para todos os outros processos sobre um enlace perfeito. Se o difusor falha no meio da transmissão, a entrega não é garantida. Porém, se ele não falha, o enlace garante que a mensagem é recebida por todos os destinatários.

Em Rodrigues et al. (2014) é proposta uma solução para a difusão de melhor-esforço baseada na topologia do vCube. Diferente do algoritmo tradicional descrito acima, 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 difusor 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 $c(j, s-1)$.

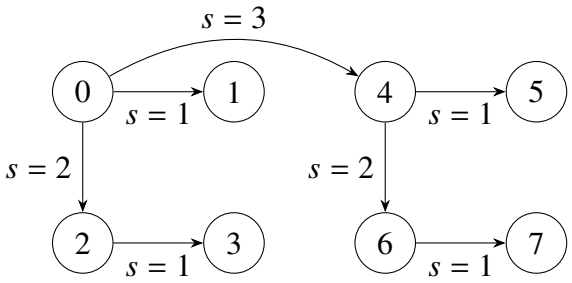


Figura 4.1: Difusão de melhor esforço partindo do processo 0 em um sistema de 8 processos.

Em um sistema com 8 processos, uma difusão partindo do processo 0 acontece como mostra a Figura 4.1. 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 $\log_2 8 = 3$.

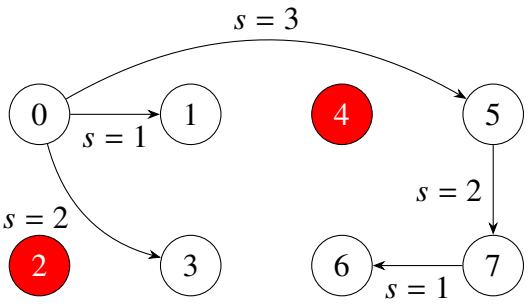


Figura 4.2: Difusão de melhor esforço partindo do processo 0 em um sistema de 8 processos sendo 2 e 4 falhos.

Em um cenário de falha, o sistema adquire a forma encontrada na Figura 4.2 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.

5 HYPERPAXOS: PAXOS EM MÚLTIPLAS INSTÂNCIAS SOBRE O VCUBE

O Paxos possui muitas variantes que procuram tornar o consenso mais eficiente (Regis e Mendizabal, 2022). Um exemplo é o Fast Paxos proposto em Lamport (2006) que permite reduzir a latência em uma mensagem, com a desvantagem de precisar de mais *acceptors* corretos para funcionar. Outro exemplo é o Cheap Paxos descrito em Lamport e Massa (2004) que trabalha com apenas $f + 1$ *acceptors* e f *acceptors* ociosos, reconfigurando o sistema em caso de falha.

Com a proposta de ter um protocolo de alta vazão, em Jalili Marandi et al. (2017), os autores propõem uma variante do Paxos sobre uma topologia de anel. Em Charapko et al. (2021), para evitar o gargalo nos *proposers*, processos retransmissores são escolhidos aleatoriamente para transmitir para grupos de processos, também permitindo maior vazão. Já em Terra (2020), a autora propõe uma variante do Paxos sobre o vCube, definindo o algoritmo para uma instância. O presente trabalho apresenta o HyperPaxos que expande e generaliza a implementação do Paxos sobre o vCube para múltiplas instâncias.

Neste algoritmo, o modelo do sistema é estático, não admitindo mudanças na configuração dos processos. Assumimos 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.

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 no Capítulo 4.

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.

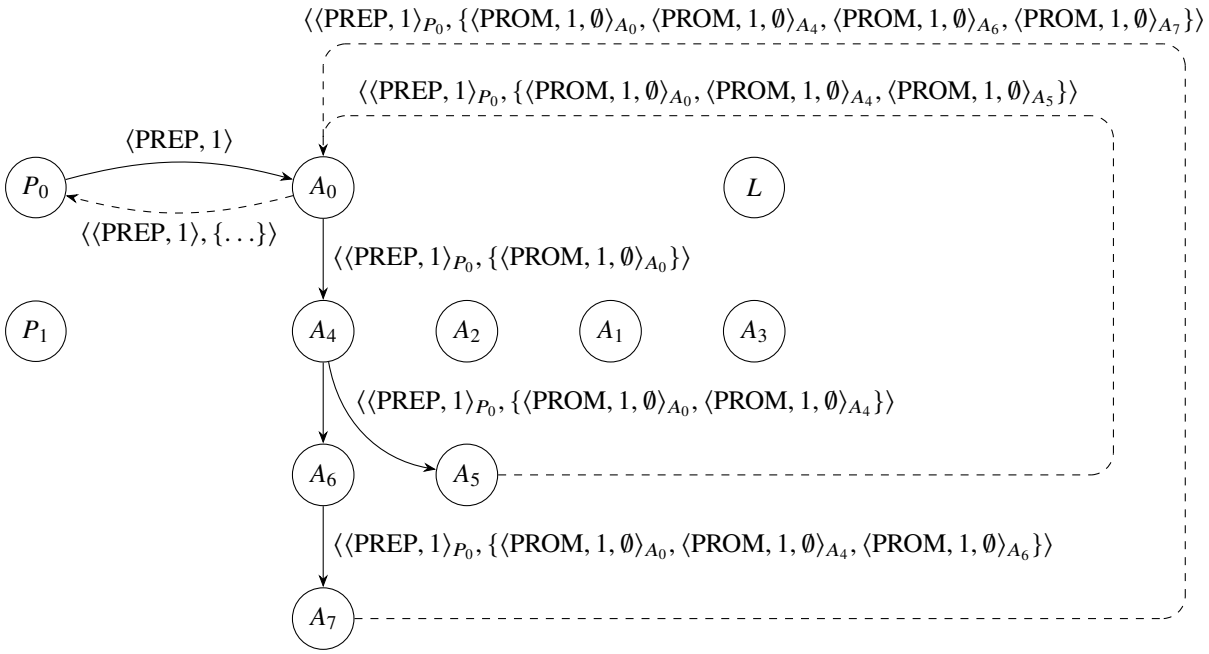


Figura 5.1: Difusão na fase 1 do maior *cluster* do *acceptor* 0 em um sistema com $n_a = 8$.

A Figura 5.1 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 5.2. 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, enviando a proposta para o *acceptor* 3 como mostra a Figura 5.3. 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*.

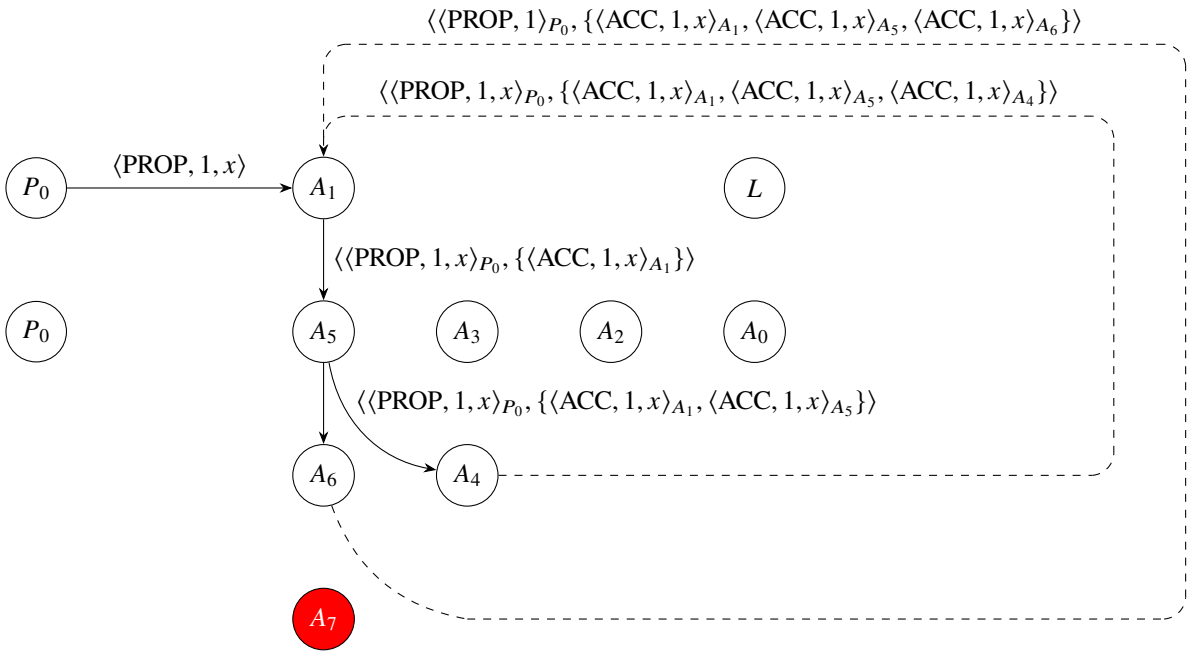


Figura 5.2: Difusão da fase 2 do maior *cluster* do *acceptor* 1 em um sistema com $n_a = 8$ e *acceptor* 7 falho.

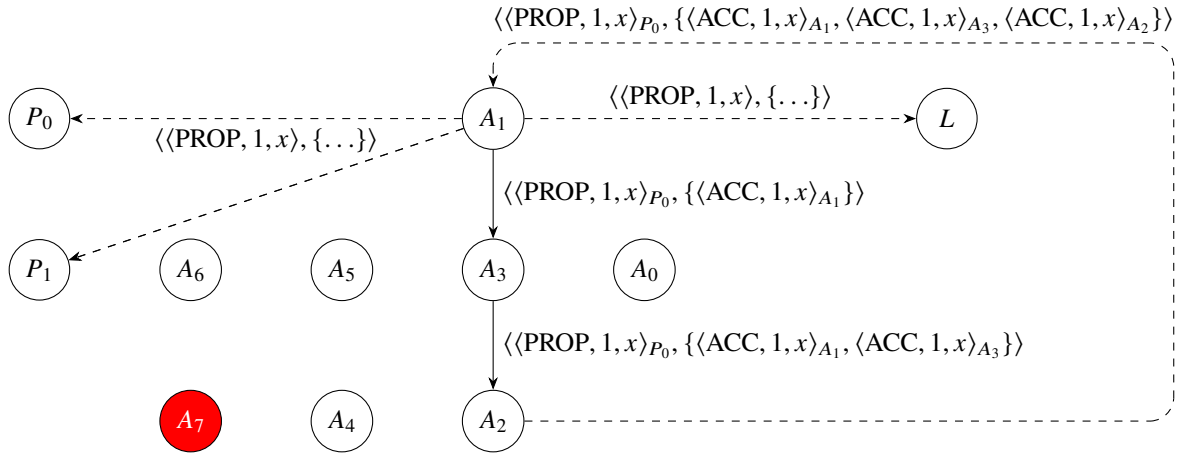


Figura 5.3: Difusão na fase 2 do segundo maior *cluster* do *acceptor* 1 em um sistema com $n_a = 8$ e *acceptor* 7 falho.

Note que esta difusão adotada pelo HyperPaxos não é uma difusão de melhor esforço, pois mesmo se o difusor da mensagem m não falhar, não é garantido que todo *acceptor*, após um intervalo de tempo finito, entregará m. Isso pode acontecer caso um *acceptor* falhe dentro de um *cluster* que recebe a difusão, uma vez que este *acceptor* não continuará a difusão. Portanto, caso haja suspeita de falha de um *acceptor* que recebeu a difusão, o *proposer* é instruído a fazer uma nova difusão. Antes do GST, processos corretos podem ser considerados falhos pelo detector de falhas. Neste caso, o Paxos continua cumprindo suas garantias, menos a de terminação como discutido na Seção 2.6.

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 5.4 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.

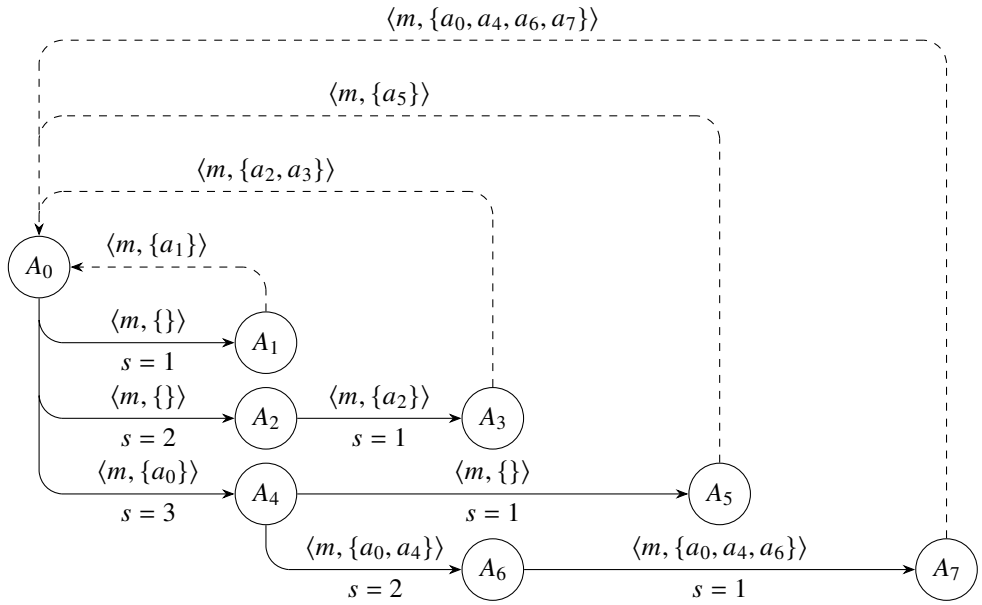


Figura 5.4: Difusão partindo do processo 0 em um sistema de 8 processos.

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.

6 IMPLEMENTAÇÃO SOBRE A LIBPAXOS

A LibPaxos (Primi e Sciascia, 2013) é um conjunto de projetos constituído de várias implementações do algoritmo Paxos. Dentre elas, se encontra a libPaxos, uma biblioteca escrita em C que permite a uma aplicação usar o Paxos para decidir valores. A libPaxos conta com três versões:

- A primeira versão é chamada de libPaxos simplesmente, ou libpaxos-T. Ela utiliza a biblioteca *pthread* para lidar com a concorrência e comunicação *multicast* IP com UDP. Porém, por utilizar *threads*, se mostrou muito sensível à pilha de protocolos de rede do sistema operacional (Primi, 2009).
- A segunda versão chamada de libPaxos^2 ou libpaxos-E usa a biblioteca *libevent* que implementa notificação de eventos. Esta versão também utiliza *multicast* IP com UDP, e inclui um serviço de eleição de líder. Sua implementação é detalhada em Primi (2009).
- A terceira versão é chamada de libPaxos^3. O *multicast* foi substituído por *unicast* com protocolo TCP. Não há eleição de líder e nesta versão a arquitetura é mais limpa, no sentido que a comunicação separada da lógica do Paxos. Sua implementação é encontrada em Sciascia (2016).

Escolhemos como base da nossa implementação a libPaxos^3, e no restante deste capítulo toda menção à libPaxos é para esta versão, caso não seja especificado. Este capítulo apresenta a libPaxos na Seção 6.1, e as nossas bibliotecas libvCube e libHyperPaxos, respectivamente, na Seção 6.2 e na Seção 6.3. Por fim, a Seção 6.4 apresenta alguns resultados experimentais obtidos.

6.1 A BIBLIOTECA LIBPAXOS

A terceira versão da libPaxos é um projeto de Sciascia (2016) iniciado em 2013 que reescreve a segunda versão da libPaxos para ter uma arquitetura mais limpa, uso do sistema de construção *CMake*, testes unitários e a substituição do *multicast* UDP por *unicast* TCP. Esta biblioteca é dividida em duas partes: a *libpaxos* e a *libevpaxos*. A *libpaxos* contém apenas o algoritmo Paxos, e não tem código de rede, podendo ser utilizada com qualquer biblioteca de rede. Já a *libevpaxos* contém uma implementação de rede usando *libevent* e a *libpaxos*.

Os três papéis do Paxos nesta implementação são bem separados, permitindo que seja possível executar um processo com apenas um papel. Em especial, a biblioteca define também o papel de *replica*, que executa os três papéis simultaneamente, todos na mesma porta TCP. Também é definido um papel de cliente, que apenas submete valores para os *proposers*. Como consequência, mais uma mensagem é necessária para cada valor decidido, mas essa separação permite que os clientes falhem sem afetar o funcionamento do Paxos. Ao submeter um valor, o cliente não recebe resposta. Para verificar se o valor foi aceito pelo sistema, é necessário também executar um *learner* para aprender sobre os valores decididos (Primi, 2009).

Como a libPaxos adota o protocolo *full-duplex* (bidirecional) TCP, alguns papéis ganham a responsabilidade de iniciar a conexão, assim como reconectar caso a conexão seja perdida. Nesse sentido, clientes que propõem valores ao sistema devem se conectar a um ou mais *proposers*. *Proposers* e *learners* devem se conectar a todos os *acceptors*. E *acceptors* não precisam iniciar a conexão com nenhum processo, afinal, eles obtêm canais bidirecionais das conexões feitas pelos *proposers* e pelos *learners*.

Para que valores sejam propostos ao sistema, os clientes enviam $\langle$CLIENT_VALUE, $v\rangle$ com seu valor v para um *proposer*. Os *proposers* mantêm os valores recebidos em uma fila para poder fazer as propostas na fase 2. As mensagens entre os processos são codificadas usando *MessagePack*, e os valores são de tamanho arbitrário.

Para executar a fase 1, um *proposer* envia $\langle$PREPARE, $i, b\rangle$ para todos os *acceptors*, sendo i o número da instância e b o número da proposta. O *proposer* espera receber respostas validando o número de proposta b de uma maioria de *acceptors*. Um *acceptor* sempre responde ao PREPARE com um $\langle$PROMISE, $i, b', p\rangle$. Se o *acceptor* julga o número de proposta b válido para a instância i, ele responde fazendo uma promessa com $b' = b$ e p a proposta já aceita na fase 2, se existir. Se o *acceptor* não julga o número de proposta b válido, pois já recebeu um número de proposta b'' maior, então envia a promessa com $b' = b''$.

Caso o *proposer* receba ao menos uma resposta com número de proposta maior que o enviado, ele imediatamente reinicia a sua fase 1, aumentando o número de proposta. Se ele não conseguir atingir quórum em um tempo determinado, porque uma maioria de *acceptors* não respondeu ao pedido de preparação, a instância também é reiniciada.

Se o número de proposta for validado na fase 1, o *proposer* prossegue para a fase 2 enviando $\langle$ACCEPT, $i, b, v\rangle$ para todos os *acceptors*. O valor v vem da proposta de maior número das promessas da fase 1, ou da fila do *proposer*, se todas as promessas recebidas não tiverem valores. Os *acceptors* podem enviar dois tipos de mensagem como resposta. Se a proposta for aceita, o *acceptor* envia um $\langle$ACCEPTED, $i, b, v\rangle$ para todos os *proposers* e também para todos os *learners*. Se for rejeitada, o *acceptor* responde com $\langle$PREEMPTED, $i, b\rangle$ apenas para o *proposer* que enviou o ACCEPT.

Ao receber ao menos um PREEMPTED, o *proposer* volta para a fase 1 e aumenta o número de proposta. A mesma coisa acontece se o *proposer* não atingir quórum em um tempo determinado. Porém, se o *proposer* atingir quórum para a fase 2, ele verifica se o valor que foi decidido é aquele proposto para aquela instância. Se o valor decidido for diferente, o valor original retorna para a fila.

Como os *learners* também recebem as mensagens ACCEPT dos *acceptors*, eles aprendem o valor decidido pela maioria de *acceptors*. Os *learners* entregam para a aplicação valores decididos conforme o identificador da instância, em ordem. Porém, devido a falhas, um *learner* pode não receber alguma decisão de uma instância. Para resolver isso, periodicamente, o *learner* verifica se existem instâncias sem decisão no meio de instâncias com valor já decidido. Caso isso ocorra, o *learner* envia uma mensagem $\langle$REPEAT, $i\rangle$ para todos os *acceptors* que respondem com suas propostas aceitas com mensagens ACCEPTED.

Os *acceptors* podem falhar e recuperar e como eles precisam recuperar as propostas aceitas, essas informações são salvas em memória secundária. A libPaxos permite utilizar o banco de dados em disco LMDB para armazenar estas informações. Para reduzir a quantidade de instâncias que precisam ser armazenadas, existem mensagens para truncar as instâncias até um determinado identificador de instância.

Por fim, a biblioteca faz uma otimização de pré-execução da fase 1. Uma quantidade de instâncias a frente das já decididas tem sua fase 1 pré-executada, de forma que o *proposer* apenas aguarda um valor para iniciar a fase 2. Assim, quando um cliente propõe um valor, o *proposer* pode ir direto para a fase 2, diminuindo a latência entre a submissão do valor e a decisão.

6.2 A IMPLEMENTAÇÃO DA LIBVCUBE

Com base na implementação de rede da libPaxos, implementamos a biblioteca libvCube que provê um serviço de detecção de falhas baseado no vCube para qualquer aplicação conforme

especificado no Capítulo 3. Cada processo tem um identificador sequencial, e a biblioteca mantém a lista de processos suspeitos de estarem falhos. Inicialmente, o estado de todos os processos é desconhecido. Quando o estado de um processo muda, a aplicação é chamada para notificar da mudança.

Atribuímos a responsabilidade de iniciar a conexão do processo i com o processo j para o processo i se $i < j$, ou seja, o processo i inicia a conexão com os processos de identificador maior que ele. Assim, apenas uma conexão de bidirecional é criada entre quaisquer processos na rede. Ao iniciar a conexão, o processo que a iniciou envia uma mensagem $\langle \text{INIT}, i \rangle$, informando que seu identificador é i.

Para identificar se um processo é falho ou não, são utilizados *timeouts*. O teste consiste em enviar uma mensagem ⟨STATE_REQUEST⟩ de pedido de informações de estado e se o processo testado não responder, seja porque ele não está conectado, ou seja porque está lento, ele é considerado falho. Ao obter uma mensagem $\langle \text{STATE\_RESPONSE}, t \rangle$ como resposta, o processo copia as novidades do vetor de *timestamps* t.

6.3 A IMPLEMENTAÇÃO DA LIBHYPERPAXOS

A libHyperPaxos implementa o algoritmo proposto no Capítulo 5 seguindo como base a implementação da libPaxos. 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. Para este trabalho, usamos a libvCube como detector de falhas, mas qualquer serviço de detecção de falhas pode ser usado, inclusive utilizando o estado das próprias conexões TCP.

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 $i < j$.

Ao iniciar uma conexão, o processo envia a mensagem $\langle \text{INIT}, i, t \rangle$. Essa mensagem serve para identificar que o processo tem papéis t e identificador i. O receptor não precisa se identificar, pois o processo que iniciou a conexão conhece os papéis e o identificador do receptor, informação que vem da configuração.

Os valores são propostos ao sistema da mesma maneira que na libPaxos, em que os clientes enviam uma mensagem $\langle \text{CLIENT\_VALUE}, v \rangle$ para um *proposer*. A execução do Paxos consiste de duas fases através da difusão adotada pelo HyperPaxos. Na primeira fase, as mensagens de pedido de preparação são PREPARE e a resposta dos *acceptors* são mensagens PROMISE. Já na segunda fase, as propostas dos *proposers* são mensagens ACCEPT e as respostas dos *acceptors* são ACCEPTED ou PREEMPTED. Esses 5 tipos de mensagens são da libPaxos sendo encapsuladas para poderem ser enviadas na libHyperPaxos.

Na fase 1, o *proposer* encapsula o $\langle \text{PREPARE}, i, b \rangle$ dentro de uma mensagem INIT_PHASE_1 para poder enviar para o *acceptor* que fará a difusão na rede. O *acceptor* ao receber a mensagem INIT_PHASE_1 sabe que é o responsável pela difusão e deve difundir o $\langle \text{PREPARE}, i, b \rangle$ que está dentro do INIT_PHASE_1. Para isso, o *acceptor* difusor primeiro obtém sua resposta $\langle \text{PROMISE}, i, b' \rangle$ para o PREPARE recebido. Em seguida, cria uma mensagem TREE_PHASE_1 que contém a mensagem PREPARE do *proposer* e a sua resposta PROMISE para o PREPARE como na libPaxos. Com a mensagem TREE_PHASE_1 pronta, o *acceptor* difusor manda esta mensagem para seu maior *cluster* e espera as respostas desse *cluster*.

Os *acceptors*, ao receberem o TREE_PHASE_1, extraem a mensagem PREPARE da mensagem e obtêm a sua resposta PROMISE para o PREPARE. Os *acceptors* concatenam a sua resposta na mensagem TREE_PHASE_1 e a repassam para frente no vCube para os seus *clusters* menores. Caso o *acceptor* seja uma folha na árvore de difusão, então ele deve responder ao *acceptor* difusor com a mensagem RESP_PHASE_1. A mensagem RESP_PHASE_1 é semelhante à mensagem TREE_PHASE_1 no sentido de encapsular o PREPARE e as respostas PROMISE, no entanto, o objetivo é retornar ao difusor, ao invés de prosseguir na árvore de difusão.

Quando o *acceptor* difusor recebe uma mensagem RESP_PHASE_1, ele salva as respostas PROMISE. Ao ter todas as respostas PROMISE do *cluster*, o *acceptor* difusor verifica se existe uma maioria de promessas que validam o número de proposta do PREPARE. Se tiver maioria, o *acceptor* manda as respostas para o *proposer* em uma mensagem RESP_PHASE_1. Caso contrário, o *acceptor* difusor deve continuar a enviar a mensagem TREE_PHASE_1 para mais *clusters*. Se esgotar o envio para todos os *clusters*, o *acceptor* envia todas as respostas de volta ao *proposer*. Assim, o *proposer* não conseguirá validar o seu PREPARE e deverá reiniciar a fase 1 com um número de proposta maior.

A fase 2 ocorre analogamente, com mensagens INIT_PHASE_2, TREE_PHASE_2 e RESP_PHASE_2. A diferença é que os tipos das mensagens são diferentes. No INIT_PHASE_2, TREE_PHASE_2 e RESP_PHASE_2, a mensagem PREPARE da fase 1 é trocada por um $\langle \text{ACCEPT}, i, b, v \rangle$ da fase 2, em que i é a instância, b é o número da proposta e v é o valor a ser proposto. As respostas concatenadas nas mensagens TREE_PHASE_2 e RESP_PHASE_2 podem ser $\langle \text{ACCEPTED}, i, b, v \rangle$ ou $\langle \text{PREEMPTED}, i, b \rangle$, no qual i é a instância, b é o número da proposta e v é o valor já aceito.

Como a biblioteca libPaxos imediatamente reinicia a fase 1 no caso de receber um PROMISE de valor maior na fase 1 ou um PREEMPTED na fase 2, o difusor retorna imediatamente para o *proposer* nestes casos também. O *timeout* do *proposer* também é mantido: se o *acceptor* difusor não responde em um determinado tempo, a fase 1 também é reiniciada com um próximo *acceptor*. Como o *proposer* não sabe se o *acceptor* escolhido está falho ou não, esta estratégia permite que um *acceptor* correto seja escolhido depois de um tempo finito, se houver pelo menos um.

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. Por exemplo, na fase 2, $\langle \langle \text{PROPOSE}, 1, x \rangle_{P_0}, \{ \langle \text{ACCEPTED}, x \rangle_{A_0}, \langle \text{ACCEPTED}, 1, x \rangle_{A_4}, \langle \text{PREEMPTED}, 1 \rangle_{A_6} \} \rangle$ é codificado como $\langle \langle \text{PROPOSE}, x \rangle_{P_0}, \{0, 4\}, \{ \langle \text{PREEMPTED}, 1 \rangle_{A_6} \} \rangle$, onde por brevidade o identificador de instância foi omitido. 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.

A implementação atualmente não conta com a verificação de buracos no *learner*, então a mensagem REPEAT não é utilizada. Outra funcionalidade não implementada é o truncamento de instâncias. Estas não são funcionalidades essenciais para o Paxos, e devem ser implementados em outro momento futuro.

6.4 RESULTADOS EXPERIMENTAIS

Foram realizados experimentos comparando a vazão da libHyperPaxos e da libPaxos medindo os valores decididos por segundo. Nos testes utilizamos um cliente e várias réplicas. O cliente é um processo que manda valores aleatórios para um *proposer* selecionado e aprende os valores decididos executando o papel de *learner*. Usando essas funcionalidades, o cliente é responsável por medir quantos valores são decididos por segundo. A réplica é um processo que executa o papel *replica*, que pode propor, decidir e aprender valores. No caso da libHyperPaxos, as réplicas também executam a libvCube para obter as informações de detecção de falhas necessárias para a libHyperPaxos.

As réplicas e o cliente foram executados em máquinas físicas diferentes. As máquinas possuem processador Intel Core i5-7500 3.4GHz e 8GB de memória RAM, todas executando o sistema operacional Linux Mint LMDE 5. A rede utilizada foi a dos computadores do laboratório do Departamento de Informática da Universidade Federal do Paraná, e a conexão entre os computadores é Gigabit Ethernet, mas, conforme as nossas medições atinge velocidade média de 500 megabits por segundo.

As bibliotecas foram testadas 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 do cliente. Isso 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.

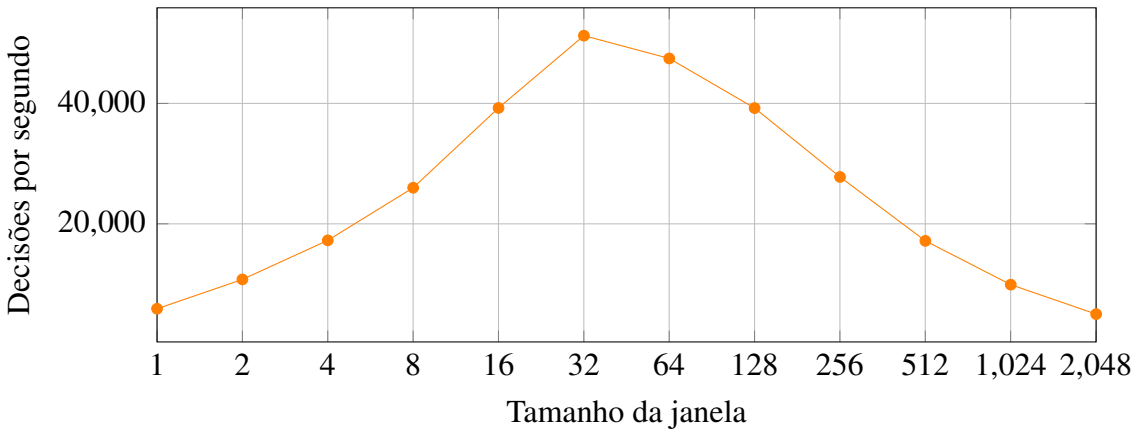


Figura 6.1: Tamanho da janela e decisões por segundo na libPaxos num sistema de 4 processos.

No caso da libPaxos, o tamanho de janela de pré-execução ótimo para nossa configuração de rede foi a janela de 32 instâncias. Na libHyperPaxos, esta janela foi de 128 instâncias. Diminuir ou aumentar o número da janela afeta drasticamente a quantidade de decisões por segundo obtidas, como mostram as Figura 6.1 e a Figura 6.2 em testes feitos em 4 réplicas. A mesma tendência se aplica a 8 e 16 réplicas. O parâmetro *outstanding* utilizado foi por volta de 1000

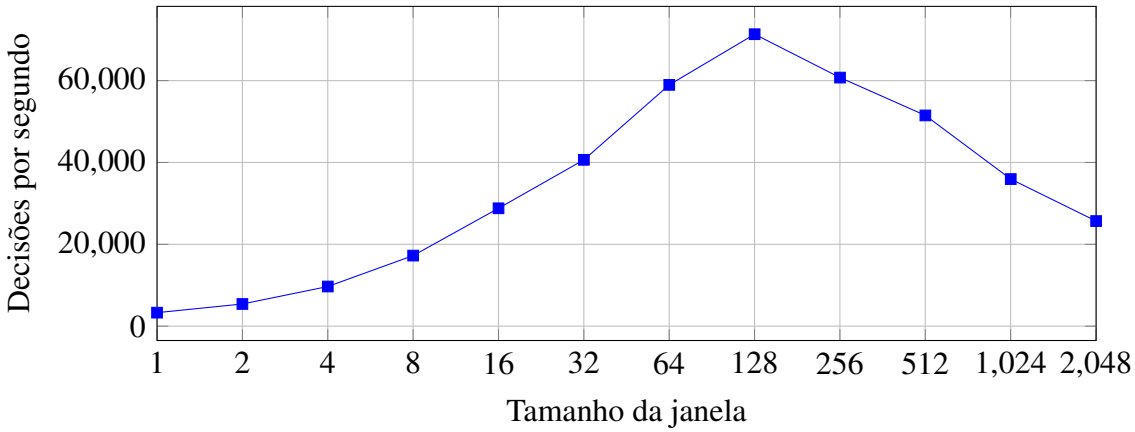


Figura 6.2: Tamanho da janela e decisões por segundo na libHyperPaxos num sistema de 4 processos.

valores simultâneos em todos os testes. Além disso, um teste rápido com 4 réplicas também mostrou que utilizar a representação com conjunto separado para algumas respostas também impacta o desempenho. Sem essa otimização, a libHyperPaxos consegue 25 mil valores por segundo. Já com a representação com conjunto, a quantidade de valores por segundo chega na casa dos 70 mil.

Os sistemas testados variam de 3 a 16 réplicas, mais o cliente. As réplicas são inicializadas com o cliente. Após 30 segundos, a amostra do último segundo de decisões por segundo é usada, depois de sua estabilização. Isso foi repetido 30 vezes, e o maior valor obtido foi utilizado. A Figura 6.3 apresenta os resultados dos testes, em um cenário sem falha. 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.

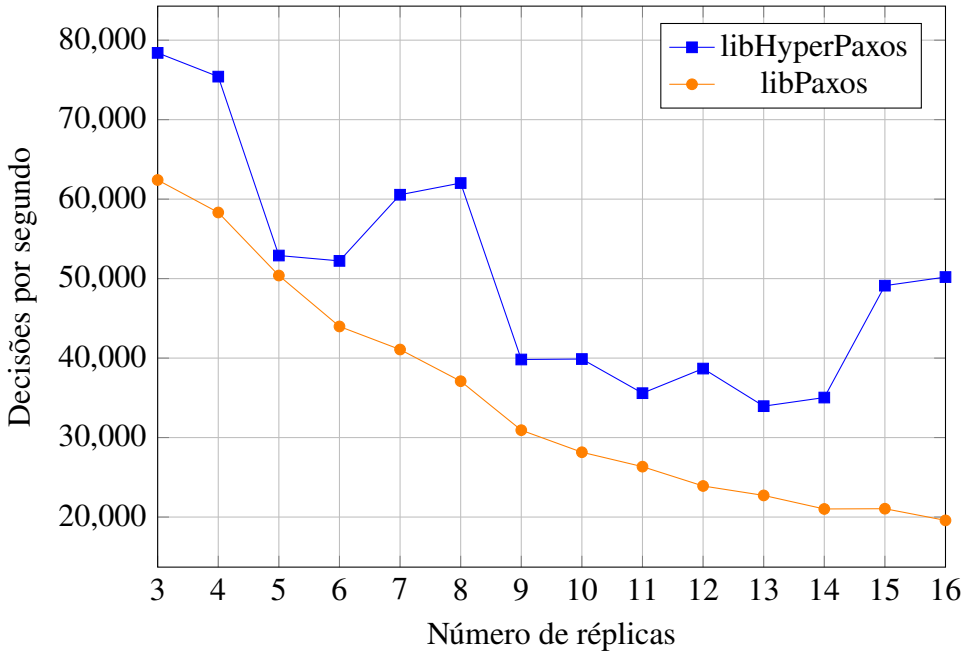


Figura 6.3: Valores decididos por segundo em relação ao número de réplicas.

Em ambas as bibliotecas, 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 da forma 2^k ou $2^k - 1$. 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. Por fim, a representação mais compacta possibilitada pela hierarquia da difusão permitiu uma redução no tamanho das mensagens e permitiu maior vazão.

7 CONCLUSÃO

Este trabalho apresentou o algoritmo HyperPaxos, uma versão hierárquica do algoritmo de consenso Paxos para múltiplas instâncias. O Paxos é amplamente utilizado em várias aplicações e serviços distribuídos atualmente em uso. Diversas variantes foram propostas procurando melhorar seu desempenho. Neste trabalho apresentamos uma versão do Paxos sobre o vCube, uma topologia hierárquica virtual escalável e com propriedades logarítmicas.

O HyperPaxos organiza os *acceptors* em *clusters*, usando o vCube. 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, uma biblioteca do Paxos. As bibliotecas foram comparadas em relação à quantidade de valores decididos por segundo, nos seus melhores parâmetros. Os resultados apresentaram aumento nos valores decididos por segundo na libHyperPaxos quando comparada à libPaxos.

Trabalhos futuros incluem melhorar a vazão das decisões por segundo com base na estrutura hierárquica nas fases 1 e 2, distribuindo as mensagens por todos os *clusters*. Além disso, é possível empregar uma estratégia de difusão melhor para *learners* e *proposers* ao final da fase 2. Resta ainda a comparação com outras variantes como o RingPaxos e o PigPaxos em um ambiente de teste mais controlado. Por fim, é possível também utilizar o vCube para definir versões hierárquicas de outros protocolos de consenso.

REFERÊNCIAS

Baker, M. e Gates, D. (2019). Lack of redundancies on Boeing 737 MAX system baffles some involved in developing the jet. *The Seattle Times*.

Brewer, E. (2017). Spanner, TrueTime and the CAP Theorem. Relatório técnico, Google.

Burrows, M. (2006). The Chubby lock service for loosely-coupled distributed systems. Em *Proceedings of the 7th symposium on Operating systems design and implementation*, páginas 335–350.

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

Charapko, A., Ailijiang, A. e Demirbas, M. (2021). PigPaxos: Devouring the Communication Bottlenecks in Distributed Consensus. Em *Proceedings of the 2021 International Conference on Management of Data*, páginas 235–247. ACM.

Charron-Bost, B., Pedone, F. e Schiper, A., editores (2010). *Replication: Theory and Practice*, volume 5959 de *Lecture Notes in Computer Science*. Springer, Berlin, Heidelberg.

de Araujo, J. P., Arantes, L., Duarte, E. P., Rodrigues, L. A. e Sens, P. (2017). A publish/subscribe system using causal broadcast over dynamically built spanning trees. Em *2017 29th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, páginas 161–168. IEEE.

Duarte Jr, E. P., Bona, L. C. E. e Ruoso, V. K. (2014). VCube: A Provably Scalable Distributed Diagnosis Algorithm. Em *2014 5th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems*, páginas 17–22, New Orleans, LA, USA. IEEE.

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

Duarte Jr., E. P., Rodrigues, L. A., Camargo, E. T. e Turchetti, R. (2022). The Missing Piece: A Distributed System-level Diagnosis Model for the Implementation of Unreliable Failure Detectors. Em *2022 11th Latin-American Symposium on Dependable Computing (LADC)*, páginas 1–10.

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

Fischer, M. J., Lynch, N. A. e Paterson, M. S. (1985). Impossibility of distributed consensus with one faulty process. *Journal of the ACM (JACM)*, 32(2):374–382.

Google (2022). Replication | Cloud Spanner | Google Cloud. https://cloud.google.com/spanner/docs/replication. Acessado em 27/08/2022.

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

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

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)*, páginas 51–58.

Lamport, L. (2006). Fast Paxos. *Distributed Computing*, 19:79–103.

Lamport, L. e Massa, M. (2004). Cheap Paxos. Em *International Conference on Dependable Systems and Networks (DSN 2004)*.

Lerner, A. (2014). The Cost of Downtime. https://blogs.gartner.com/andrew-lerner/2014/07/16/the-cost-of-downtime/. Acessado em 18/09/2022.

Primi, M. (2009). Paxos made code: implementing a high throughput atomic broadcast. Dissertação de Mestrado, University of Lugano, Lugano, Suíça.

Primi, M. e Sciascia, D. (2013). LibPaxos: Open-source Paxos. http://libpaxos.sourceforge.net/. Acessado em 19/06/2022.

Raynal, M. (2005). A short introduction to failure detectors for asynchronous distributed systems. *ACM SIGACT News*, 36(1):53–70.

Regis, S. e Mendizabal, O. M. (2022). Análise comparativa do algoritmo Paxos e suas variações. Em *Anais do Workshop de Testes e Tolerância a Falhas (WTF)*, páginas 71–84. SBC. ISSN: 2595-2684.

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

Rodrigues, L. A., Cohen, J., Arantes, L. e Duarte Jr, E. P. (2013). A Robust Permission-Based Hierarchical Distributed k-Mutual Exclusion Algorithm. Em *2013 IEEE 12th International Symposium on Parallel and Distributed Computing*, páginas 151–158, Bucharest, Romania. IEEE.

Rodrigues, L. A., Duarte Jr, E. P. e Arantes, L. (2014). Árvores geradoras mínimas distribuídas e autonômicas. *Anais do 32o Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuıdos–SBRC*, 2014:1–14.

Rodrigues, L. A., Jeanneau, D., Duarte Jr, E. P. e Arantes, L. (2017). Uma Solução de Difusão Confiável Hierárquica em Sistemas Distribuídos Assíncronos. Em *Anais do XXXV Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos*. SBC.

Sciascia, D. (2016). sciascid / libpaxos — Bitbucket. https://bitbucket.org/sciascid/libpaxos/src/master/. Acessado em 19/06/2022.

Terra, A. d. C. (2020). Investigando a implementação do consenso escalável sobre o VCube. Dissertação de Mestrado, Universidade Federal do Paraná.

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