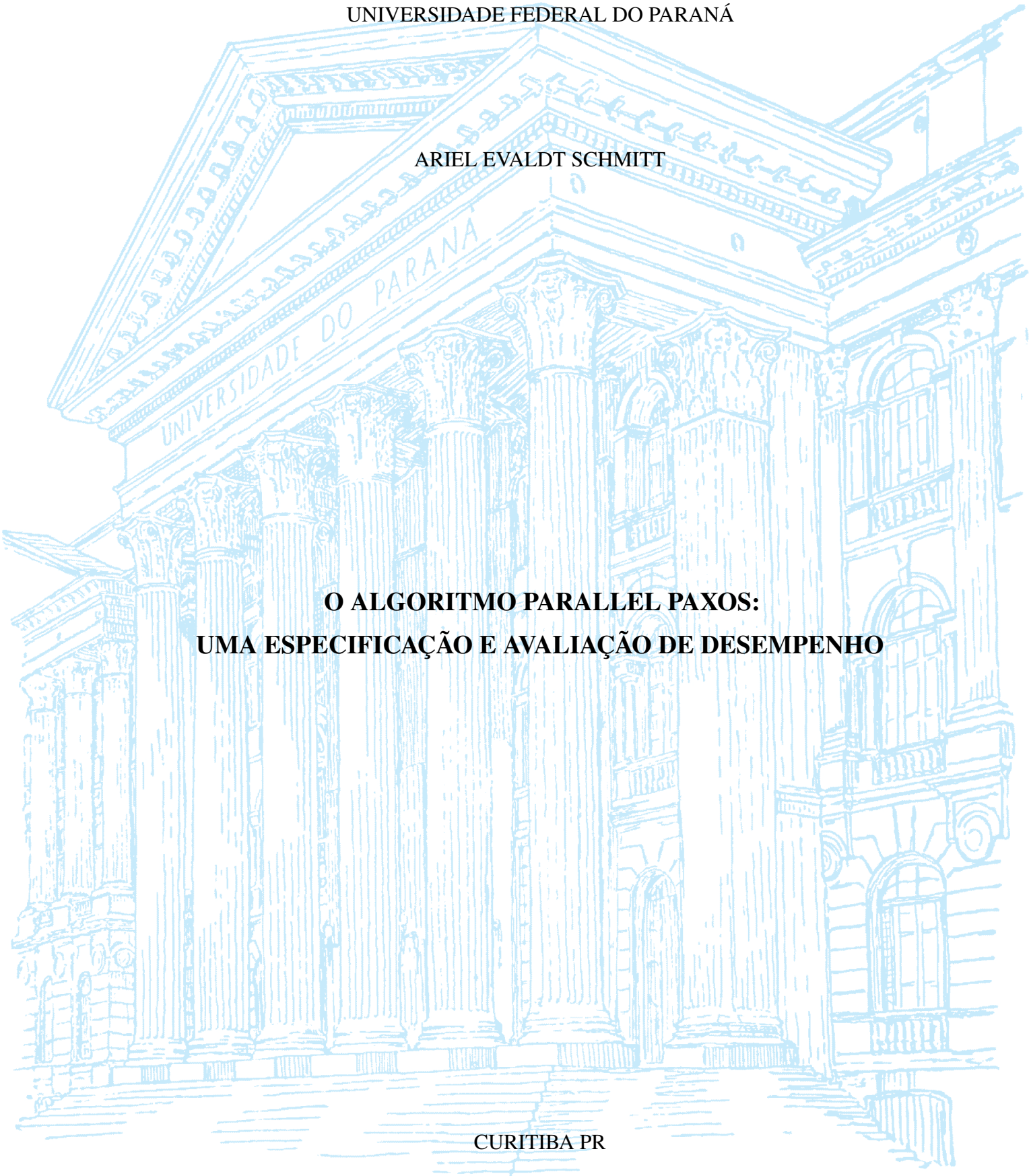
UNIVERSIDADE FEDERAL DO PARANÁ

ARIEL EVALDT SCHMITT



O ALGORITMO PARALLEL PAXOS:
UMA ESPECIFICAÇÃO E AVALIAÇÃO DE DESEMPENHO

CURITIBA PR

2025

ARIEL EVALDT SCHMITT

O ALGORITMO PARALLEL PAXOS: UMA ESPECIFICAÇÃO E AVALIAÇÃO DE DESEMPENHO

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

Área de concentração: *Computação*.

Orientador: Elias Procópio Duarte Jr.

CURITIBA PR

2025

Universidade Federal do Paraná
Setor de Ciências Exatas
Curso de Ciência da Computação

Ata de Apresentação de Trabalho de Conclusão de Curso 2

Título do Trabalho: O Algoritmo Parallel Paxos: Uma Especificação e Avaliação de Desempenho

Autor: GRR20203949 Nome: Ariel Evaldt Schmitt

Apresentação: Data: 10/07/2025 Hora: 19:30 Local: Videoconferência

Orientador: Elias Procópio Duarte Jr. (UFPR) Membro 1: Edson Tavares de Camargo (UTFPR-Toledo)

Membro 2: Giovanni Venâncio de Souza (UFPR) Membro 3: Fernando Monteiro Kiotheka (UFPR)

AVALIAÇÃO – Produto escrito	**ORIENTADOR**	**MEMBRO 1**	**MEMBRO 2**	**MEMBRO 3**	**MÉDIA**
Conteúdo (00-40)	40	40	40	40	40
Referência Bibliográfica (00-10)	10	10	10	10	10
Formato (00-05)	5	5	5	5	5
AVALIAÇÃO – Apresentação Oral					
Domínio do Assunto (00-15)	15	15	15	15	15
Desenvolvimento do Assunto (00-05)	5	5	5	5	5
Técnica de Apresentação (00-03)	3	3	3	3	3
Uso do Tempo (00-02)	2	2	2	2	2
AVALIAÇÃO – Desenvolvimento					
Nota do Orientador (00-20)	20	**********	**********	**********	20
NOTA FINAL	**********	**********	**********	**********	**100**

Os pesos indicados são sugestões.

Conforme decisão do colegiado do curso de Ciência da Computação, a entrega dos documentos comprobatório de trabalho de Conclusão de Curso 2 deve deve respeitar os seguintes procedimentos: o orientador deve abrir um processo no Sistema Eletrônico de Informações (SEI – UFPR); Selecionar o tipo: *Graduação: Trabalho Conclusão de Curso*; informar os interessados: nome do aluno e o nome do orientador; anexar esta ata escaneada e a versão final do PDF da monografia do aluno; Tramitar o processo para CCOMP (Coordenação de Ciência da Computação).

gov.br Documento assinado digitalmente
ELIAS PROCOPIO DUARTE JUNIOR
Data: 16/07/2025 12:22:24-0300
Verifique em https://validar.iti.gov.br

gov.br Documento assinado digitalmente
GIOVANNI VENANCIO DE SOUZA
Data: 16/07/2025 13:01:32-0300
Verifique em https://validar.iti.gov.br

gov.br Documento assinado digitalmente
FERNANDO MONTEIRO KIOTHEKA
Data: 16/07/2025 18:35:35-0300
Verifique em https://validar.iti.gov.br

gov.br Documento assinado digitalmente
EDSON TAVARES DE CAMARGO
Data: 16/07/2025 22:24:03-0300
Verifique em https://validar.iti.gov.br

AGRADECIMENTOS

Primeiramente, gostaria de agradecer ao meu pai e à minha mãe que sempre me apoiaram e me incentivaram — especialmente nos estudos —, prestando todo o apoio necessário para que eu pudesse me desenvolver tanto profissionalmente como pessoalmente. Agradeço também a todos os demais familiares que, mesmo distantes, nunca deixaram de entrar em contato para saber sobre meu bem-estar e possíveis novidades — ainda que eu nem sempre tome a iniciativa de fazer o mesmo.

Gostaria de agradecer também a todos os meus amigos — tanto da época do ensino fundamental e médio quanto aos que conheci durante a graduação. Tudo teria sido mais difícil sem as amizades com quem pude conversar, contar piadas, estudar, festejar, entre tantas outras atividades que ajudaram a aliviar o estresse e tornar essa jornada mais leve.

Agradeço a todos os professores com os quais tive contato, em especial aos docentes do Departamento de Informática da UFPR, pela excelência no ensino e por contribuírem significativamente para a minha formação.

Deixo meu agradecimento aos membros da banca, os professores Edson Tavares e Giovanni Venâncio, e ao mestrando Fernando Kiotheka por todas as sugestões, críticas construtivas e comentários positivos.

Por fim, agradeço ao professor Elias Duarte Jr. por todo o apoio durante a realização deste trabalho. Sua orientação foi excepcional e de extrema importância em meio a tantas ideias, dúvidas e incertezas.

RESUMO

O consenso é considerado por muitos o problema central de sistemas distribuídos. Neste problema, um conjunto de processos devem se comunicar e cooperar entre si a fim de que todos decidam por um mesmo valor ou ação. Por exemplo, em bancos de dados distribuídos, todas as réplicas devem decidir igualmente se irão efetivar ou abortar uma transação, garantindo a consistência dos dados replicados. O Paxos é um dos principais algoritmos de consenso. Três papéis de processos são definidos no Paxos: os *proposers* propõem um valor, os *acceptors* aceitam ou não o valor proposto e os *learners* aprendem o valor decidido. A propriedade de segurança (*safety*) do consenso é garantida pelo Paxos sob qualquer circunstância. Este trabalho descreve o algoritmo Parallel Paxos em que múltiplos processos de um sistema paralelo replicam dados entre si. Cada processo funciona como *proposer* e *acceptor* dos seus próprios dados e escolhe $2f$ outros processos do sistema como seus *acceptors*. O Parallel Paxos garante que, se o processo falhar e for substituído, os dados estarão disponíveis, garantindo a continuidade da execução paralela. Além disso, se o processo é apenas suspeito de ter falhado, mas continua correto e um substituto é criado e ambos salvam dados replicados, o Parallel Paxos garante a consistência do sistema. No trabalho é descrito um experimento em que processos comunicando sobre UDP/IP e também TCP/IP são falsamente suspeitos e substituídos, o que causa a inconsistência dos dados replicados. O Parallel Paxos utiliza uma estratégia multi-líder, o que elimina o gargalo causado por um único *proposer* existente no Paxos. Além disso, reduz o número de mensagens e passos de comunicação necessários para decidir um valor. Diversos experimentos foram realizados, utilizando a biblioteca LibPaxos, os quais consistem em três clientes emitindo valores aleatórios para os *proposers* decidirem. As métricas utilizadas são a vazão média por cliente e a vazão média total, em decisões por segundo. Os resultados experimentais demonstram um ganho de vazão de até 46% quando o Parallel Paxos é implementado com múltiplos processos ou *threads*. Para implementações em que o conjunto de instâncias é particionado e distribuído entre os *proposers*, os resultados evidenciam um aumento de até 61% na vazão. Destaca-se que, dos experimentos realizados com três clientes, as implementações paralelas foram as únicas capazes de superar a vazão total do Paxos com um único cliente.

Palavras-chave: Consenso. Paxos. Tolerância a falhas.

LISTA DE FIGURAS

LISTA DE TABELAS

SUMÁRIO

1 INTRODUÇÃO

Sistemas distribuídos estão cada vez mais presentes, tendo em vista o contexto em que a internet se tornou ubíqua. Os algoritmos clássicos de consenso distribuído são particularmente importantes para aplicações que demandam alta disponibilidade, tolerância a falhas e escalabilidade (Cachin et al., 2011; Corbett et al., 2013; Hunt et al., 2010; Ruchel et al., 2024; Venâncio et al., 2019). Um dos requisitos centrais dos sistemas distribuídos é a necessidade de que múltiplos processos cooperem entre si respeitando as propriedades desejadas. Em contextos que vão de bancos de dados replicados a serviços de nuvem, manter a consistência é algo crucial. De maneira resumida, manter a consistência significa garantir que todas as réplicas mantêm os mesmos valores armazenados e, consequentemente, se dois clientes realizam uma leitura dos dados replicados em um mesmo instante, então ambos observam os mesmos valores. Nesse contexto, algoritmos de consenso desempenham um papel fundamental, pois permitem que processos independentes alcancem um acordo em comum (Turek e Shasha, 2002; Venâncio et al., 2021; Ongaro e Ousterhout, 2014). Este acordo pode ser, por exemplo, uma ação a ser executada ou um valor a ser armazenado.

Entre os algoritmos de consenso existentes, o Paxos (Lamport, 1998, 2001; Van Renesse e Altinbunken, 2015) se destaca como um dos mais conhecidos. O Paxos garante a propriedade de segurança (*safety*) do consenso sob quaisquer circunstâncias. Isto é, dois processos corretos distintos jamais decidem por valores diferentes. Já a propriedade de progressão é garantida apenas sob certas condições. Isto significa que é possível que o algoritmo nunca termine sua execução, ou seja, os processos podem nunca alcançar uma decisão.

Esta limitação do Paxos com relação à progressão é esperada por conta da impossibilidade FLP (Fischer et al., 1985). A impossibilidade FLP prova que não existe algoritmo determinístico capaz de garantir a execução correta do consenso em um sistema assíncrono sujeito a falhas por parada. Isso ocorre pois, em sistemas assíncronos, os limites temporais para os eventos do sistema não são conhecidos e, por conta disso, é impossível distinguir um processo falho de um processo lento. Por falhas por parada, ou falhas *crash*, entende-se que o processo para de responder completamente (Cachin et al., 2011).

O Paxos define três papéis que são exercidos pelos processos do sistema: *proposer*, *acceptor* e *learner*. Os *proposers* são responsáveis por receber as requisições dos clientes e propor os valores. Os *acceptors* são responsáveis por aceitar, ou não, o valor proposto. Por fim, os *learners* devem ser capazes de aprender o valor que foi decidido pelo sistema. A execução do Paxos é dividida em duas fases e em cada uma delas é necessário que uma maioria de *acceptors* não estejam falhos. Por conta disso, para um sistema em que f processos podem falhar, são necessários um total de $N = 2f + 1$ processos.

Para que a progressão do algoritmo seja garantida, é necessário que apenas um *proposer*, chamado de coordenador, possa emitir propostas. O problema disso é que o coordenador se torna um gargalo no sistema, já que todas as requisições dos clientes precisam ser encaminhadas e respondidas por meio dele. Por conta disso, surgiram algumas variantes do Paxos que adotam uma estratégia multi-líder, como o Egalitarian Paxos (Moraru et al., 2013) e o Mencius (Mao et al., 2008).

Diante desse contexto, este trabalho tem como objetivo apresentar o algoritmo Parallel Paxos (de Camargo et al., 2017a,b; de Camargo, 2017), detalhando seu funcionamento e suas motivações. No Parallel Paxos, cada cliente tem seu próprio coordenador e este coordenador atende a um único cliente. Dessa maneira, o gargalo causado pelo coordenador único é eliminado.

Além disso, cada cliente possui seu próprio conjunto de réplicas e, consequentemente, cada cliente tem sua própria sequência de execuções do Paxos. Algumas otimizações ainda são possíveis, como por exemplo coalocar o cliente com o *proposer*-coordenador, a fim de diminuir o número de mensagens e passos de comunicação necessários para decidir um valor.

O Parallel Paxos é capaz de garantir a consistência do sistema mesmo na presença de falsas suspeitas de falhas. Isso é fundamental, pois, quando ocorre uma falsa suspeita, o sistema pode substituir o processo suspeito por um novo processo. Como o processo original continua em execução, é necessário que o sistema mantenha a consistência ao permitir que ambos salvem dados replicados simultaneamente.

Manter os dados replicados consistentes mesmo sob falsas suspeitas de falhas é uma tarefa crucial. Outras formas de replicação, como a replicação passiva, ou Primary-Backup (Alsberg e Day, 1976), não são capazes de manter a consistência quando isso ocorre. Neste tipo de replicação, existe um processo denominado *primary* e os demais são *backups*. O *primary* processa as requisições dos clientes, envia o resultado para os *backups*, aguarda as respostas de todos eles e, por fim, responde ao cliente. Ao detectar a falha do *primary*, um *backup* pode assumir este papel. No entanto, conforme experimento realizado, a existência de dois processos *primary* pode gerar uma inconsistência entre os *backups*. Por conta disso, algoritmos como o Paralell Paxos são de extrema importância.

O algoritmo Parallel Paxos foi implementado utilizando a biblioteca LibPaxos (Primi e Sciascia, 2013) que consiste em uma coleção de implementações de código-aberto do algoritmo Paxos. Para isso, utilizou-se a terceira versão da biblioteca pois ela isola completamente o núcleo do Paxos de qualquer código específico para a comunicação entre processos. No total foram realizadas três implementações utilizando diferentes estratégias. A primeira delas utiliza processos distintos para cada uma das réplicas, a segunda implementa as réplicas por meio de *threads* e a última utiliza o mesmo conjunto de réplicas para todos os clientes mas particiona o conjunto de instâncias do consenso entre eles — cada cliente possui um conjunto disjunto de instâncias.

Com o algoritmo implementado, diversos experimentos foram realizados a fim de comparar o Parallel Paxos com o Paxos tradicional. As métricas utilizadas foram a vazão média por cliente e a vazão média total, em decisões por segundo. Os experimentos principais consistem em três clientes enviando valores aleatórios para seus *proposers*. Todas as implementações paralelas com três clientes apresentaram um ganho de vazão quando comparadas com o Paxos tradicional com três clientes. Na implementação em que cada réplica consiste em um processo distinto, houve um ganho de vazão de aproximadamente 38% para um único cliente. Já na implementação com *threads*, um único cliente obteve um ganho de 46% na sua vazão. A implementação que utiliza um único conjunto de réplicas, com particionamento das instâncias entre os clientes, obteve um ganho de vazão de 61%. Por fim, destaca-se que, dos experimentos com três clientes, as implementações paralelas foram as únicas capazes de obter uma vazão total maior do que um único cliente requisitando decisões no Paxos tradicional.

O restante deste trabalho está organizado da seguinte maneira. O Capítulo 2 apresenta o problema do consenso e suas propridades, além de descrever o algoritmo Paxos. O Capítulo 3 apresenta e especifica o algoritmo Parallel Paxos. As implementações, os experimentos e seus resultados são apresentados no Capítulo 4. Por fim, o Capítulo 5 conclui o trabalho.

2 O ALGORITMO DE CONSENSO PAXOS

O consenso é considerado por muitos o problema central de sistemas distribuídos. Este capítulo inicia introduzindo o consenso e descreve brevemente as suas propriedades. Em seguida é apresentado o algoritmo Paxos. A Seção 2.3 descreve a primeira e segunda fase de execução de uma instância do algoritmo. A Seção 2.4 discorre sobre como os processos aprendem o valor decidido. Na Seção 2.5 é discutida a propriedade de terminação no Paxos e em quais condições ela é garantida. Por fim, a Seção 2.6 trata sobre replicação máquina de estados e como o Paxos pode ser utilizado para garantir a consistência entre um conjunto de réplicas.

2.1 O PROBLEMA DO CONSENSO

Um sistema distribuído consiste de um conjunto de processos que se comunicam e cooperam entre si (Cachin et al., 2011). Frequentemente, os diferentes processos precisam executar ações coordenadas a fim de manter a consistência do sistema. Solucionar o problema do consenso (Fischer et al., 1985) implica em garantir o acordo entre os diferentes processos. Um exemplo clássico em que precisamos de consenso são os bancos de dados distribuídos. Na replicação distribuída, um conjunto de processos gerenciadores de recursos devem decidir igualmente se irão abortar ou efetivar uma transação, garantindo a consistência dos dados replicados (Wiesmann et al., 2000).

Em sistemas assíncronos (Lynch, 1996), os limites temporais para os eventos do sistema não são conhecidos. Como por exemplo, não há garantia de tempo para a entrega de uma mensagem ou para a execução de uma instrução por um processo. Sendo assim, é impossível distinguir um processo falho de um processo lento.

Um resultado conhecido, denominado de impossibilidade FLP (Fischer et al., 1985), prova que não existe algoritmo determinístico capaz de garantir a execução correta do consenso em um sistema assíncrono sujeito a falhas por parada. A falha de um único processo em um momento inoportuno pode fazer com que os processos nunca alcancem um acordo.

Além disso, processos podem falhar de diferentes maneiras. O modelo de falhas mais comum é o modelo de falhas por parada, ou modelo de falhas *crash*, em que o processo afetado simplesmente não responde para qualquer que seja a entrada recebida. Outro modelo muito conhecido é o modelo de falhas bizantinas (Lamport, 1982), em que o processo apresenta comportamento arbitrário e possivelmente malicioso. Suportar falhas bizantinas é mais complexo e exige mais recursos, pois não se pode confiar nas mensagens recebidas de um processo. Por conta disso, assumimos que podem ocorrer apenas falhas por parada, ou falhas por parada com recuperação. Nesta última, os processos devem ser capazes de recuperar o seu estado anterior à falha, para então continuarem a execução a partir deste estado (Cachin et al., 2011).

O problema do consenso pode ser especificado por meio de três propriedades:

Acordo (*Agreement*): Processos corretos distintos nunca decidem por valores diferentes.

Terminação (*Termination*): Os processos corretos eventualmente chegam a uma única decisão.

Validade (*Validity*): Apenas um valor que foi proposto pode ser decidido.

A propriedade do acordo garante que não ocorra uma inconsistência nas decisões de dois processos. Essa propriedade está relacionada à segurança (*safety*) do consenso, pois impede que

o sistema entre em estados inválidos. Já a propriedade de terminação garante que, eventualmente, algum valor será decidido, evitando o caso em que o sistema para de decidir valores. Essa é uma propriedade de progressão (*liveness*), pois assegura que o sistema irá progredir, ou seja, irá fazer o que deve ser feito. A propriedade de validade garante que apenas valores propostos são decididos, ou seja, não são permitidos valores espúrios. Por fim, destaca-se que o sistema decide uma única vez. No entanto, é possível executar várias instâncias do problema do consenso a fim de obter novas decisões.

Um dos principais algoritmos de consenso é o Paxos, que recebeu este nome devido a um sistema legislativo ficcional, da ilha de Paxos na Grécia (Lamport, 1998). Devido a impossibilidade FLP, o algoritmo garante a terminação apenas sob certas condições. No entanto, a segurança do consenso é sempre garantida, mesmo diante de falhas de processos. O algoritmo Paxos é descrito na seção a seguir.

2.2 O ALGORITMO DE CONSENSO PAXOS

Antes de descrever o algoritmo propriamente dito, é importante definir três papéis que são exercidos pelos processos executando o Paxos: *proposers*, *acceptors* e *learners* (Lamport, 2001). Os *proposers* são responsáveis por propor um valor, os *acceptors* são responsáveis por aceitar ou não o valor proposto e, por fim, os *learners* devem ser capazes de aprender, ou seja, descobrir o valor que foi decidido pelos *acceptors*. Note que, um único processo pode assumir mais de um papel ao mesmo tempo dentro de um sistema.

A fim de garantir as propriedades do consenso no Paxos, é necessário que um quorum de *acceptors* esteja correto. Considere um sistema distribuído com N processos e capaz de tolerar até f falhas por parada (*crash*). Considerando que f processos estão falhos, então outros $f + 1$ devem estar funcionando corretamente. Sendo assim, a quantidade de processos do sistema deve ser pelo menos $N = 2f + 1$ processos. Dado um conjunto de $2f + 1$ elementos, qualquer subconjunto de $f + 1$ elementos tem intersecção de pelo menos 1 elemento com qualquer outro subconjunto de $f + 1$ elementos (Rodrigues et al., 2016). Por conta disso, jamais haverá uma situação em que dois quóruns decidem dois valores distintos.

É importante notar que, mesmo se mais do que f *acceptors* estejam falhos , a propriedade de segurança continuará sendo satisfeita, ou seja, dois valores diferentes jamais serão aceitos por duas maiorias de *acceptors*. De fato, se mais do que f *acceptors* estão falhos, então um *proposer* nem sequer obterá respostas de uma maioria de *acceptors* e, com isso, apenas a propriedade da terminação será afetada. Essa propriedade é discutida com mais detalhes na Seção 2.5.

2.3 DECIDINDO UM VALOR COM O PAXOS

Como mencionado na seção anterior, a corretude do algoritmo depende da propriedade da intersecção de quóruns. Portanto, para que seja possível suportar a falha de *acceptors*, utilizam-se vários *acceptors*, mais especificamente, $2f + 1$. A fim de garantir a consistência do sistema, uma instância do algoritmo Paxos é dividida em duas fases. A primeira fase consiste em uma preparação de um *proposer* com um quorum de *acceptors*. A segunda fase consiste na proposta do valor em si.

Para compreender o funcionamento de cada fase, é necessário entender primeiramente como um *proposer* opera. Os *proposers* são os processos responsáveis por receber uma requisição de um cliente externo e tentar fazer com que o sistema decida por esta requisição. Como podem existir vários clientes e vários *proposers*, é necessário que haja uma maneira de diferenciar uma proposta da outra. Para isso, utilizam-se os números de propostas, também chamados de *ballots*.

Cada *proposer* deve atribuir um número para suas propostas, e dois *proposers* distintos jamais podem utilizar um mesmo número. Além disso, é importante que exista uma ordem total entre estes números. Utilizar números naturais de maneira crescente garante essa ordem. A Tabela 2.1 apresenta uma possibilidade de conjuntos disjuntos de números de propostas para um sistema com quatro *proposers*. Sendo N o número total de *proposers* do sistema, um *proposer* i obtém seus números de proposta por meio do conjunto $B = \{i + kN \mid k \in \mathbb{N}\}$, em ordem crescente.

Tabela 2.1: Conjuntos disjuntos de números de propostas para um sistema com 4 *proposers*.

Proposer	**Números de Proposta**
0	0 4 8 12 ⋯
1	1 5 9 13 ⋯
2	2 6 10 14 ⋯
3	3 7 11 15 ⋯

A primeira fase, que chamamos de **p1**, funciona da seguinte maneira:

p1a Um *proposer* seleciona um número de proposta b e envia uma mensagem PREPARE<b> para um quorum de *acceptors*.

p1b Um *acceptor*, ao receber uma mensagem PREPARE<b>, deve responder apenas se ainda não respondeu nenhuma outra mensagem PREPARE<b'> em que $b' > b$. Sendo b maior do que qualquer outro número de proposta já recebido, o *acceptor* deve responder ao *proposer* com uma mensagem PROMISE<b, v, vb>, em que v é o último valor aceito por este *acceptor* e vb é o número da proposta associada a este valor. Os valores v e vb são vazios caso o *acceptor* não tenha aceito nenhuma proposta. Ao responder com a mensagem PROMISE<b, v, vb>, o *acceptor* está prometendo ao *proposer* que não aceitará nenhum outro valor cujo número de proposta seja menor do que b.

Um *proposer* pode avançar para a Fase 2, que chamamos de **p2**, após receber a mensagem **p1b** de uma maioria de *acceptors*. A Fase 2 funciona da seguinte maneira:

p2a O *proposer* agora deve escolher um valor para a sua proposta. Caso algum dos *acceptors* tenha informado em uma mensagem **p1b** que já havia aceito um valor v, o *proposer* deve selecionar o valor v associado ao maior número de proposta recebido vb. Caso contrário, o *proposer* pode selecionar qualquer valor de sua escolha. Ao selecionar um valor, o *proposer* envia uma mensagem ACCEPT<b, v> para um quorum de *acceptors*.

p2b Ao receber uma mensagem ACCEPT<b, v>, um *acceptor* aceita a proposta apenas se não respondeu a uma mensagem PREPARE<b'> em que $b' > b$. Ao aceitar a proposta, o *acceptor* responde com uma mensagem ACCEPTED<b>.

Ao receber a mensagem ACCEPTED<b> de uma maioria de *acceptors*, o *proposer* descobre que a sua proposta foi decidida e nada será capaz de alterar essa decisão. Note que uma decisão no Paxos ocorre assim que uma maioria de acceptors aceita um valor. A mensagem ACCEPTED<b> é opcional e serve apenas para informar que isso aconteceu. A Figura 2.1 ilustra a execução das duas fases do algoritmo com dois *proposers*.

A primeira fase do algoritmo é essencial para garantir a propriedade de segurança do consenso, ela consiste basicamente em uma coordenação entre os valores propostos pelos *proposers*. Note pela Figura 2.1 que, sem a primeira fase, o *proposer* P_2 seria capaz de alterar

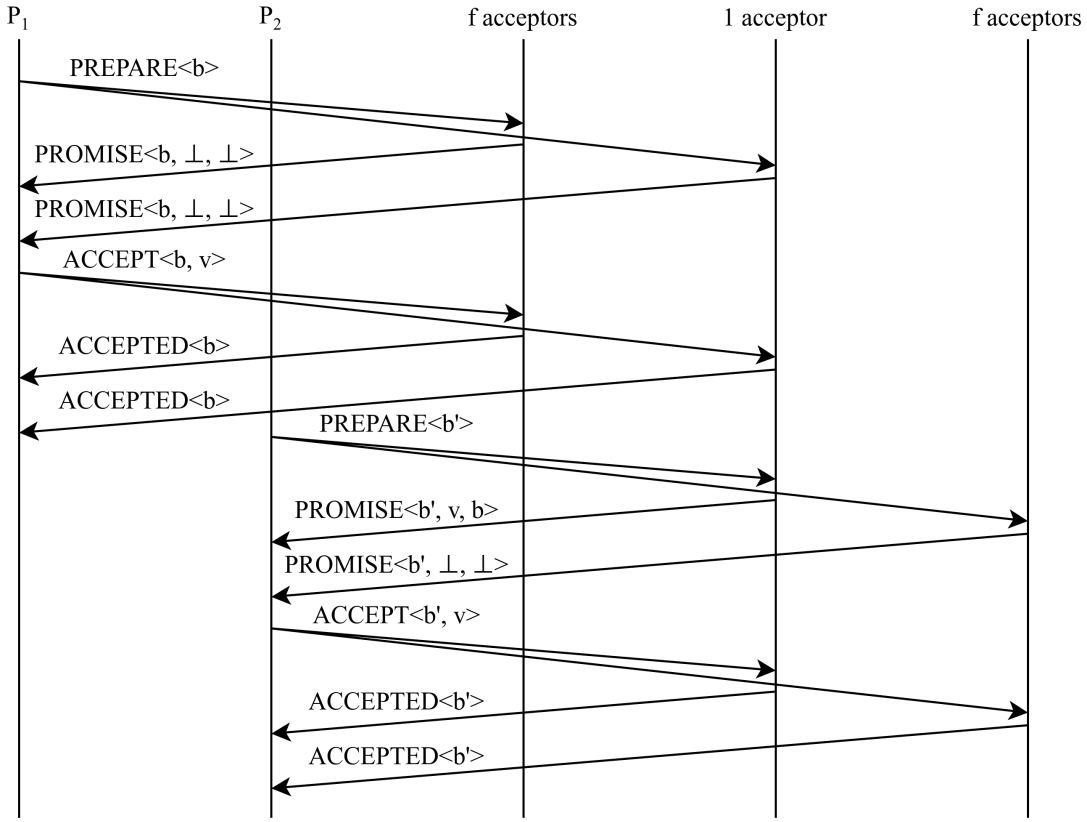


Figura 2.1: Este diagrama de tempo mostra a execução normal do algoritmo Paxos com dois *proposers*. O *proposer* P_1 executa a primeira e segunda fase com uma maioria de *acceptors* e consegue decidir pelo valor *v* com número de proposta *b*. O *proposer* P_2 executa a Fase 1 com outra maioria de *acceptors* e descobre, por meio do *acceptor* em comum com a maioria utilizada por P_1, que um valor já foi aceito . Por conta disso, P_2 não tem outra alternativa senão escolher o mesmo valor *v* para a sua proposta, a qual é aceita por sua maioria de *acceptors*. O símbolo ⊥ nas respostas dos *acceptors* indica que eles ainda não aceitaram nenhum valor.

o valor decidido pelo *proposer* P_1. Foi por meio das respostas dos *acceptors* na Fase 1 que o *proposer* P_2 descobriu que um valor já havia sido aceito, e possivelmente decidido, pelo sistema. Sendo assim, ele não pôde propor um valor diferente do que já havia sido decidido.

Outro ponto que pode passar despercebido, ou não ser compreendido facilmente, é o fato de que *acceptors* podem aceitar mais do que um valor. Este é um requisito necessário pois se houver dois ou mais *proposers* propondo valores, é possível que ocorra um cenário em que todos os *acceptors* aceitaram um valor mas nenhum valor foi aceito por uma maioria de *acceptors*. Por conta disso, um *acceptor* deve sempre aceitar um valor com número de proposta maior do que qualquer promessa feita na Fase 1. A Figura 2.2 mostra uma execução em que isso ocorre.

Não é necessário que um *proposer* se comunique sempre com uma mesma maioria de *acceptors*. É possível executar a primeira fase com uma maioria e a segunda fase com outra. É garantido que, caso o quorum seja obtido, pelo menos um dos *acceptors* tenha participado das duas fases. Esse comportamento pode ser observado na Figura 2.2. Por fim, destaca-se que o *proposer* pode executar previamente a primeira fase do algoritmo mesmo antes de receber um valor a ser decidido. Assim, quando um cliente solicitar a decisão de um valor, o *proposer* poderá iniciar diretamente a segunda fase, reduzindo o tempo necessário para concluir a decisão.

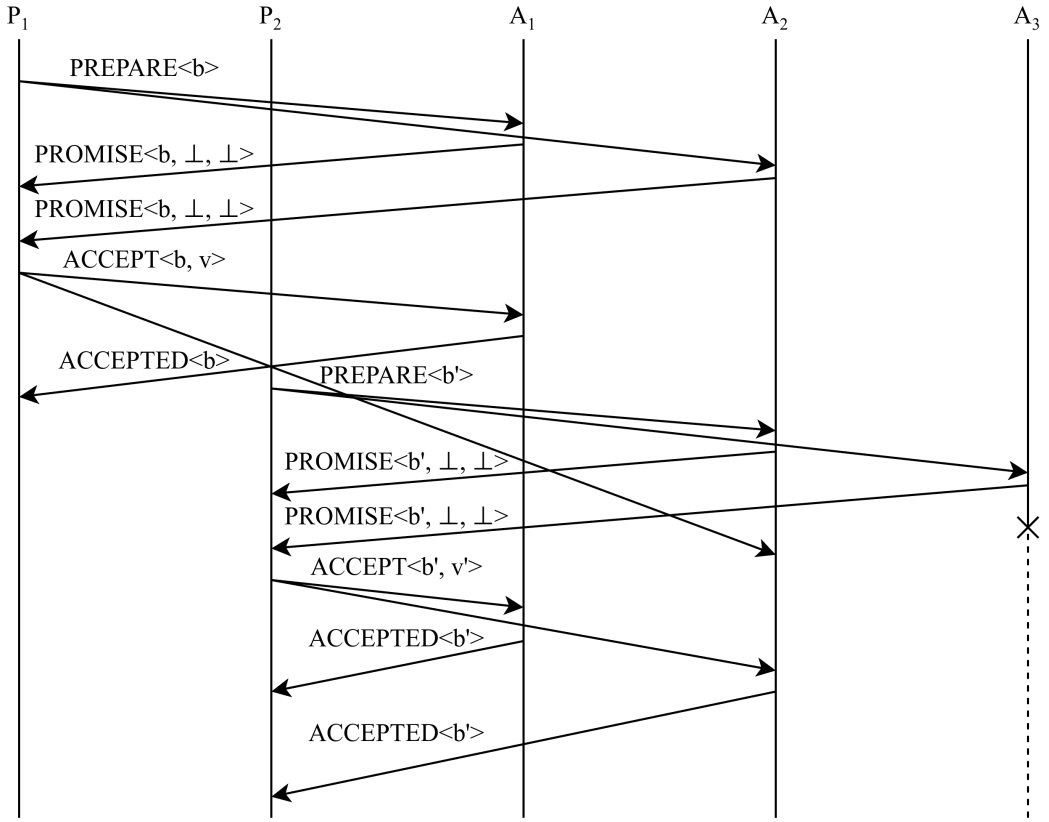


Figura 2.2: Este diagrama de tempo exemplifica uma execução do algoritmo Paxos em que um *acceptor* aceita dois valores diferentes mas apenas um valor é decidido. O *proposer* P_1 obtém quorum ao executar a Fase 1 com número de proposta b e então executa a Fase 2 com valor v. O canal de comunicação atrasa a mensagem **p2a** enviada por P_1 para o *acceptor* A_2 e, por conta disso, apenas o *acceptor* A_1 aceita o valor v. Um outro *proposer* P_2 seleciona o número de proposta $b' > b$ e obtém quorum ao executar a Fase 1 com os *acceptors* A_2 e A_3 que ainda não aceitaram nenhum valor. A_3 falha logo em seguida. Por fim, P_2 escolhe o valor v' e executa a segunda fase com os *acceptors* A_1 e A_2 que aceitam a proposta fazendo com que o valor v' seja decidido pelo sistema. Note que A_1 havia aceito o valor v proposto por P_1 e, posteriormente, aceitou o valor v' proposto por P_2, já que $b' > b$.

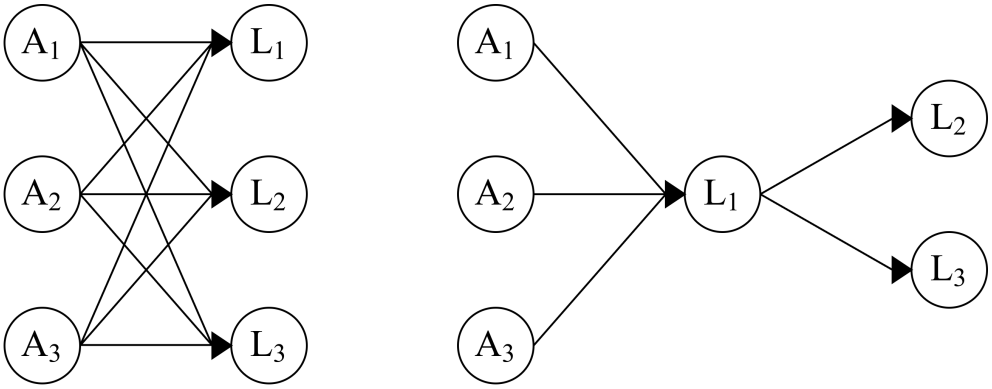
2.4 APRENDENDO UM VALOR

Na Seção 2.2, foram apresentados os papéis dos processos no Paxos, e na Seção 2.3 foi descrito o comportamento dos *proposers* e *acceptors* durante a execução do consenso. A seguir, será apresentado o comportamento dos *learners* que até então não foi abordado. Os *learners* são processos interessados na decisão do consenso, ou seja, eles devem ser capazes de descobrir qual foi o valor decidido pelos *acceptors*. A seguir, são discutidas duas abordagens, apresentadas em Lamport (2001), para que um *learner* descubra o valor decidido no sistema.

Ao aceitar um valor, além de notificar o *proposer*, os *acceptors* podem também notificar todos os *learners*. Um *learner* aprende o valor decidido ao receber a mensagem **p2b** de uma maioria de *acceptors*, todas contendo o mesmo número de proposta. Esta é a maneira mais simples e direta. No entanto, exige um grande número de mensagens, correspondente ao produto do número de *acceptors* pelo número de *learners* do sistema.

Como o sistema assume que falhas bizantinas não ocorrem, um *learner* pode aprender o valor decidido por meio de outro *learner*. Neste caso, uma maioria não é necessária pois as mensagens do sistema são confiáveis. Sendo assim, é possível eleger um líder para os *learners*. Os *acceptors* então notificam os valores aceitos para este líder, o qual aguarda por uma maioria. Ao obter quorum, o líder aprende o valor decidido e pode notificar os demais *learners*. Desta

forma, o número de mensagens é proporcional à soma do número de *acceptors* com o número de *learners* do sistema. Embora esta abordagem diminua o número de mensagens, os *learners* precisam de dois passos de comunicação para aprender o valor decidido ao invés de apenas um. Note que, mesmo se um líder falhar, outro processo é eleito em seguida — como por exemplo, com o algoritmo de eleição de líder de Aguilera et al. (2001). Além disso, se houver mais de um líder, a consistência ainda assim estará garantida. A Figura 2.3 ilustra as duas abordagens.



(a) Cada *acceptor* notifica cada *learner.*

(b) Os *acceptors* notificam o líder dos *learners* e este notifica os demais *learners.*

Figura 2.3: Duas abordagens para notificar os *learners* sobre a decisão de um valor.

2.5 TERMINAÇÃO

O algoritmo Paxos não está imune à impossibilidade FLP, apresentada na Seção 2.1. Sendo assim, existem dois cenários em que os processos executando o Paxos podem não alcançar uma decisão. O primeiro cenário ocorre quando mais do que f *acceptors* estão falhos ou suspeitos. O segundo cenário ocorre quando dois ou mais *proposers* estão executando simultaneamente.

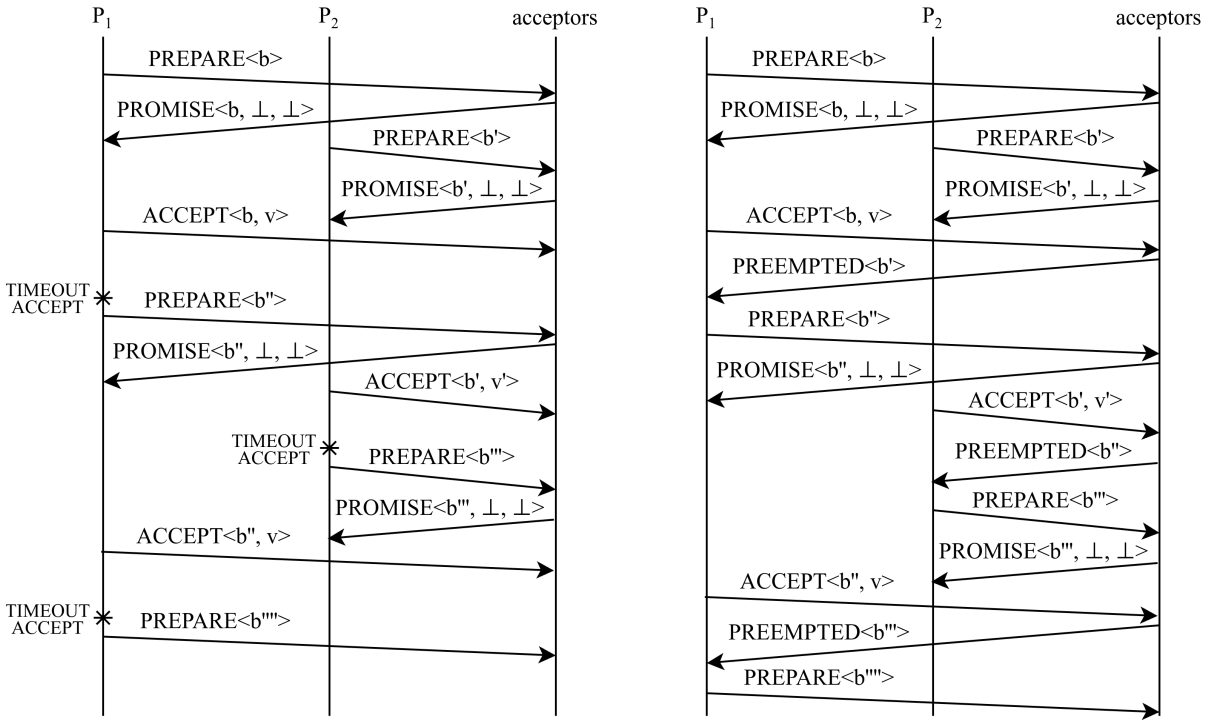
Como apresentado na Seção 2.3, um *proposer* só avança a execução do algoritmo quando obtém respostas de um quorum de *acceptors*. Como o total de processos no sistema é igual a $2f + 1$, se mais do que f processos estão falhos então o quorum nunca será obtido. Por conta disso, o valor de f deve ser escolhido cuidadosamente de modo que este cenário ocorra com baixa probabilidade.

Se dois ou mais *proposers* estão propondo valores, ao mesmo tempo, é possível que um interfira na execução do outro. Por exemplo, um *proposer* P_1 pode executar a Fase 1 com número de proposta b e obter quorum. Logo em seguida, o *proposer* P_2 também executa a Fase 1 com número de proposta $b' > b$ e obtém quorum. Quando P_1 executa a Fase 2 ele não consegue atingir quorum pois os *acceptors* já prometeram não aceitar nenhuma proposta com número menor do que b'. Então P_1 executa novamente a Fase 1 com número $b'' > b'$ e obtém quorum, invalidando a Fase 1 do processo P_2, que precisará executá-la novamente. Este ciclo pode continuar indefinidamente.

A fim de evitar este segundo cenário, é necessário que apenas um *proposer* possa emitir propostas. Para isso, é possível utilizar um algoritmo de eleição de líder para selecionar este *proposer*, que muitas vezes é chamado de coordenador. Diversos algoritmos podem ser utilizados para selecionar este coordenador, um exemplo é o algoritmo de Aguilera et al. (2001). Ainda assim, devido às características assíncronas do sistema, é possível que dois processos acreditem que são líderes e concorram entre si. No entanto, note que nenhum dos dois cenários apresentados nesta seção afeta a segurança do consenso. Portanto, embora a presença de dois líderes possa

interromper o progresso do algoritmo, ela não viola a consistência e, por isso, não representa um problema — sendo tolerável quando ocorre com baixa probabilidade.

Um *proposer* pode utilizar duas estratégias para detectar que deve aumentar seu número de proposta porque, possivelmente, está concorrendo com outro *proposer*. Uma delas é estipular um *timeout* para obter quorum. Se uma maioria de *acceptors* não respondeu dentro do tempo estipulado então ele deve aumentar seu número de proposta e executar novamente a Fase 1. A outra necessita que os *acceptors* informem o *proposer* que um número de proposta maior já foi adotado (Lamport, 2001). Por exemplo, os *acceptors* podem responder com uma mensagem PREEMPTED<b'> (Van Renesse e Altinbunken, 2015), a qual indica para o *proposer* que sua proposta de número b foi rejeitada pois o número $b' > b$ já foi adotado. A Figura 2.4 demonstra duas execuções do algoritmo em que dois *proposers* estão concorrendo pela decisão, cada execução utiliza uma das estratégias mencionadas acima.



(a) O *proposer* aguarda por um timeout antes de aumentar o seu número de proposta.

(b) Os *acceptors* avisam o *proposer* quando seu número de proposta foi rejeitado.

Figura 2.4: Dois *proposers* concorrendo pela decisão do consenso.

Apesar da existência de um único *proposer* garantir a progressão do algoritmo, essa abordagem introduz algumas limitações. A principal delas é que esse *proposer* pode se tornar um gargalo, já que todas as requisições dos clientes precisam ser encaminhadas e respondidas por meio dele. Em sistemas geograficamente distribuídos, essa centralização pode aumentar significativamente a latência, pois clientes em diferentes regiões devem se comunicar com um único líder, que pode estar fisicamente distante. Para mitigar esses problemas, variantes do Paxos foram propostas, como o Egalitarian Paxos (Moraru et al., 2013) e o Mencius (Mao et al., 2008), que exploram a ideia de múltiplos líderes para distribuir a carga e reduzir a latência.

2.6 REPLICAÇÃO MÁQUINA DE ESTADOS

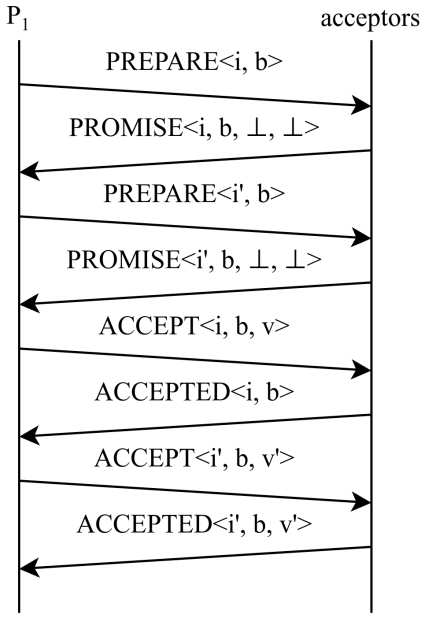
A abordagem de replicação máquina de estados foi proposta inicialmente para sistemas sem falhas em Lamport (1978). Ela consiste em uma forma de replicação ativa (Wiesmann et al., 2000) em que diferentes réplicas são representadas por uma máquina de estados determinística. Portanto, se todas as réplicas executam a mesma sequência de requisições então elas mantêm o mesmo estado (Schneider, 1990).

Considere um sistema com vários clientes fazendo requisições para várias réplicas. É garantido que todas as réplicas irão receber e executar os comandos na mesma ordem se os clientes emitirem suas requisições por meio de difusão atômica. A difusão atômica garante que se dois processos corretos p e q entregam duas mensagens m e m' então p entrega m antes de m' se e somente se q entrega m antes de m'. É possível demonstrar que o problema da difusão atômica é equivalente ao problema do consenso por meio de uma redução de um problema em outro (Chandra e Toueg, 1991).

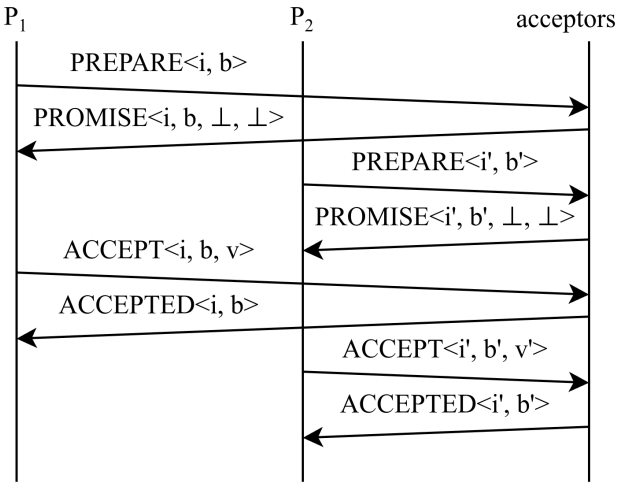
Sendo assim, o algoritmo Paxos pode ser utilizado para replicação máquina de estados. A ideia consiste em executar várias instâncias do algoritmo, onde cada uma dessas instâncias decide uma requisição a ser executada pelas réplicas. Por exemplo, a primeira instância decide a primeira requisição, a segunda instância decide a segunda requisição, e assim por diante. É importante destacar que não é preciso aguardar a decisão de uma instância para então partir para outra, elas podem ocorrer simultaneamente. É possível até mesmo que a requisição $r + 2$ seja decidida antes da requisição r. Essa abordagem é chamada de multi-Paxos (Van Renesse e Altinbunken, 2015).

A principal diferença do multi-Paxos para o Paxos é o controle dos números das instâncias. Como apresentado nas seções anteriores, todas as mensagens do algoritmo possuem um número de proposta b. No multi-Paxos, toda mensagem também deve possuir um número de instância i. O par $<i, b>$ deve ser único por proposta e dois *proposers* distintos não podem utilizar o mesmo par. Cada processo do sistema pode participar de várias instâncias simultaneamente. Antes de responder a uma mensagem com base no algoritmo apresentado na Seção 2.3, um processo deve ler o valor de i, resgatar o estado desta instância na memória e responder a mensagem de acordo com o estado daquela instância. A Figura 2.5 mostra duas execuções do multi-Paxos, uma com um único *proposer* e outra com dois *proposers* propondo em instâncias diferentes. Perceba que, dois *proposers* que estão propondo em instâncias diferentes não interferem um no outro.

Assim como em uma única instância do Paxos, no multi-Paxos os *proposers* também podem executar antecipadamente a primeira fase do algoritmo. No entanto, no multi-Paxos, um *proposer* pode executar essa fase para um número ilimitado de instâncias. Dessa forma, à medida que recebe requisições dos clientes, ele pode iniciar diretamente a segunda fase para cada uma delas, o que reduz a latência do sistema.



(a) Um *proposer* emitindo propostas em duas instâncias.



(b) Dois *proposers* emitindo propostas em instâncias diferentes.

Figura 2.5: Duas execuções do multi-Paxos.

3 O ALGORITMO PARALLEL PAXOS

No Capítulo 2, foi apresentado o algoritmo de consenso Paxos, e na Seção 2.6, discutiu-se sobre como utilizá-lo na implementação da replicação máquina de estados. Este capítulo apresenta o algoritmo Parallel Paxos, que surgiu no contexto de sistemas de computação de alto desempenho baseados em MPI (*Message Passing Interface*), com o objetivo de tolerar falhas e garantir consistência entre múltiplas réplicas (de Camargo et al., 2017a,b; de Camargo, 2017). Ele herda as propriedades do consenso garantidas pelo Paxos, mas adapta sua estrutura para explorar paralelismo e melhorar o desempenho em ambientes distribuídos de alto desempenho. A seção a seguir discorre sobre algumas características da replicação passiva e ativa. A Seção 3.2 especifica o algoritmo Parallel Paxos na sua forma mais básica. Algumas possíveis otimizações para o algoritmo são apresentadas na Seção 3.3, que visam reduzir a latência das requisições e o número de processos necessários. Por último, a Seção 3.4 discute uma estratégia que permite que múltiplos *proposers* proponham valores simultaneamente sem que interfiram um no outro.

3.1 REPLICAÇÃO PASSIVA E ATIVA

Replicação consiste em criar múltiplas cópias de objetos, como arquivos, dados, banco de dados entre outros, os quais possivelmente são alterados e acessados por múltiplos clientes. Técnicas de replicação são empregadas com o objetivo de garantir alta disponibilidade, integridade, desempenho, ou qualquer combinação destas (Charron-Bost et al., 2010). O principal desafio em replicar objetos é garantir a consistência de suas cópias, de modo que sua replicação seja transparente para o cliente. A replicação distribuída é utilizada tanto para garantir a disponibilidade como aumentar o desempenho nos mais diversos contextos (Huff et al., 2021; Duarte e dos Santos, 2001; Venâncio e Duarte Jr, 2022; Duarte Jr e Godoi, 2014)

Apesar de existirem diversas técnicas de replicação, duas delas tornaram-se bastante conhecidas: replicação ativa e replicação passiva. Na replicação ativa, todas as réplicas processam todas as requisições independentemente. Na replicação passiva, apenas uma réplica processa as requisições e envia os resultados para as demais. Replicação máquina de estados é o termo mais comum utilizado para replicação ativa, a Seção 2.6 apresentou sua implementação por meio da execução de uma sequência de instâncias do Paxos. A seguir, será apresentada uma visão geral da replicação passiva.

Na replicação passiva, também conhecida como Primary-Backup, uma réplica é denominada como *primary* e as demais são *backups* (Alsberg e Day, 1976). Os clientes interagem apenas com a *primary*. Ao receber uma requisição de um cliente, a *primary* processa a requisição, envia o resultado para todos os *backups*, aguarda a confirmação de todos eles e, por fim, responde ao cliente. Nesta abordagem, o resultado das requisições pode ser não-determinístico, pois apenas uma réplica processa e envia o resultado para as demais. Requisições de leitura podem ser respondidas diretamente pela *primary*, sem passar pelos *backups*.

O papel de *primary* pode migrar entre as réplicas, o que permite ao sistema tolerar falhas. Assim, considerando N réplicas, ele pode tolerar até $N - 1$ falhas. Ao detectar a falha da réplica primária, um *backup* pode passar a exercer este papel e processar as requisições dos clientes. No entanto, para que isso seja possível é necessário monitorar a réplica primária. Na Seção 2.1, mencionamos que, em sistemas assíncronos sujeitos a falha, não é possível distinguir um processo falho de um processo lento. Portanto, há a possibilidade do sistema operar com duas réplicas primárias ao mesmo tempo caso haja uma falsa suspeita de falhas, o que pode

levar à inconsistência entre as réplicas. Sendo assim, a abordagem de replicação passiva deve assumir o modelo de falhas *fail-stop*, em que (1) se um processo falha, todos os demais processos eventualmente irão detectar a falha, e (2) nenhum processo detecta a falha de um outro processo a não ser que este esteja de fato falho (Charron-Bost et al., 2010).

Uma maneira de evitar o desacordo entre os diferentes processos com relação à detecção de falhas, sem depender do modelo *fail-stop*, é utilizar um serviço centralizado de visões. Este serviço centralizado de visões é responsável por detectar a falha de processos e informar os demais processos. Dessa maneira, todos os processos tem a mesma visão do sistema, mesmo que haja falsas suspeitas, isto é, um processo correto é suspeito de estar falho. Desta forma, todos os processos terão a mesma visão de quem está falho e quem está correto. No entanto, essa abordagem introduz um ponto único de falhas e o sistema pode não progredir se este serviço falhar. Em van Renesse e Schneider (2004) é proposta a implementação de um processo *master* replicado, utilizando o algoritmo Paxos, que exerce o papel do serviço centralizado de visões. Assim, o serviço em si também é replicado e pode suportar um número configurável de falhas.

De qualquer forma, a replicação passiva necessita que a detecção de falhas seja perfeita ou, alternativamente, depende de um serviço centralizado de visões — o que pode se tornar um ponto único de falha. Por outro lado, as técnicas de replicação ativa conseguem evitar inconsistências sem depender da detecção perfeita de falhas. Como visto na Seção 2.5, a existência de dois processos líderes não afeta a propriedade de segurança do consenso no algoritmo Paxos e, consequentemente, também não afeta a consistência na replicação máquina de estados quando implementada com o Paxos. A seguir apresentamos o algoritmo Parallel Paxos, que surge como uma alternativa para o Paxos tradicional, permitindo paralelismo das requisições de múltiplos clientes.

3.2 ESPECIFICAÇÃO DO ALGORITMO PARALLEL PAXOS

O modelo de sistema sob o qual o Parallel Paxos opera é o mesmo do Paxos: um sistema assíncrono sujeito a falhas por parada com recuperação. No Parallel Paxos, cada cliente tem sua própria sequência de execuções do Paxos e, consequentemente, seu próprio *proposer*. Além do mais, cada *proposer* possui seu próprio conjunto de $2f + 1$ *acceptors*. Dessa maneira, é possível que vários *proposers* executem concorrentemente sem que interfiram um no outro. A fim de especificar o seu funcionamento, vamos considerar dois clientes c_1 e c_2 que fazem requisições simultâneas. Cada um dos seus *proposers*, p_1 e p_2, propõem os valores v_1 e v_2, respectivamente. A execução segue da seguinte maneira:

1. Ambos os clientes c_1 e c_2 enviam suas requisições para os seus próprios *proposers*, p_1 e p_2, que propõem, respectivamente, os valores v_1 e v_2.

2. Ao receber as requisições, os *proposers* iniciam a execução da primeira fase do Paxos de maneira independente:

   a) Primeiramente, ambos selecionam seus números de proposta, por exemplo b_1 e b_2.

   b) Cada *proposer* envia uma mensagem **p1a** para sua própria maioria de *acceptors*. O *proposer* p_1 envia PREPARE<b_1> e p_2 envia PREPARE<b_2>.

3. Ao receber uma mensagem **p1a**, um *acceptor* responde de acordo com a especificação do Paxos. Uma resposta do tipo **p1b** é enviada para o *proposer* apenas caso o seu número de proposta b seja maior do que qualquer outro já recebido.

4. Ao receber as respostas **p1b** de uma maioria de seus *acceptors*, os *proposers* prosseguem para a segunda fase:

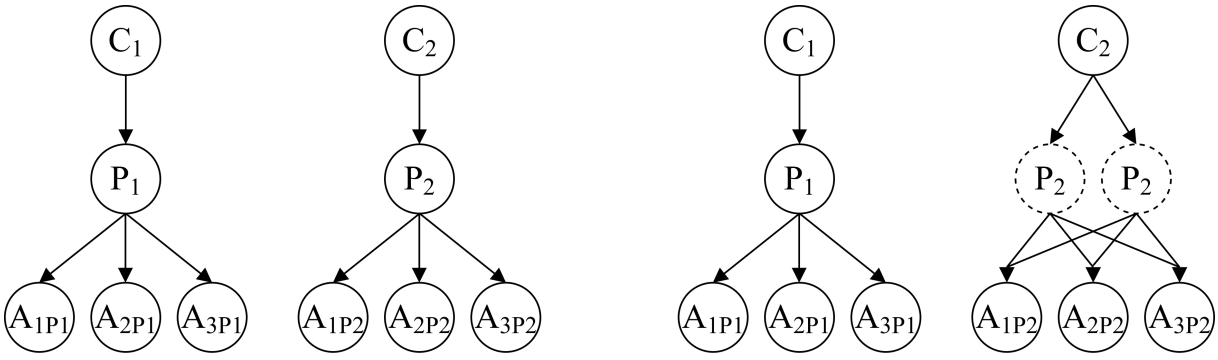
   a) Caso nenhum dos *acceptors* reportou já ter aceito um valor, os *proposers* podem escolher os valores solicitados por seus clientes. Senão, devem escolher o valor reportado com maior número de proposta, como dita a especificação do Paxos.

   b) Supondo que nenhum *acceptor* reportou um valor aceito, o *proposer* p_1 escolhe o valor v_1 e p_2 escolhe v_2. Com os valores escolhidos, ambos os *proposers* enviam suas mensagens **p2a** para as suas respectivas maiorias de *acceptors*.

5. Ao receber uma mensagem **p2a**, um *acceptor* responde de acordo com a especificação do Paxos. O valor é aceito apenas se o número de proposta é maior do que o de qualquer outra mensagem **p1a** adotada. Ao aceitar um valor, um *acceptor* responde ao *proposer* com uma mensagem **p2b**.

6. Após receber a mensagem **p2b** de sua maioria de *acceptors*, o *proposer* aprende que o seu valor foi escolhido e, agindo como um *learner*, notifica o cliente.

O gargalo causado por um único *proposer* no Paxos tradicional é eliminado no Parallel Paxos, pois cada cliente possui seu próprio *proposer*. O problema de terminação, causado por múltiplos *proposers* ativos, também é eliminado pois cada *proposer* possui seu próprio conjunto de *acceptors*. Em execuções típicas, há apenas um *proposer* ativo por cliente. No entanto, devido à natureza assíncrona do sistema, é possível que outro *proposer* se torne ativo para um mesmo cliente. Essa situação ocorre no seguinte caso: o *proposer* é falsamente suspeito de ter falhado e outro *proposer* é iniciado. Mesmo nesse cenário, a consistência do sistema é preservada, pois o Parallel Paxos herda as garantias de consistência do Paxos. Neste caso, o sistema pode apresentar uma falha temporária de progresso apenas para o cliente que tiver múltiplos *proposers*, enquanto os demais continuam operando normalmente. A Figura 3.1(a) apresenta o cenário típico de execução do Parallel Paxos com dois clientes, já a Figura 3.1(b) apresenta o cenário em que um dos clientes possui dois *proposers* e pode não progredir na decisão de seus valores.



(a) Cenário típico.

(b) Uma falsa suspeita de falha causou o cenário em que o cliente C_2 possui dois *proposers* e pode não progredir.

Figura 3.1: Parallel Paxos com dois clientes em um cenário típico e em um cenário com uma falsa suspeita de falha.

3.3 OTIMIZAÇÕES NO PARALLEL PAXOS

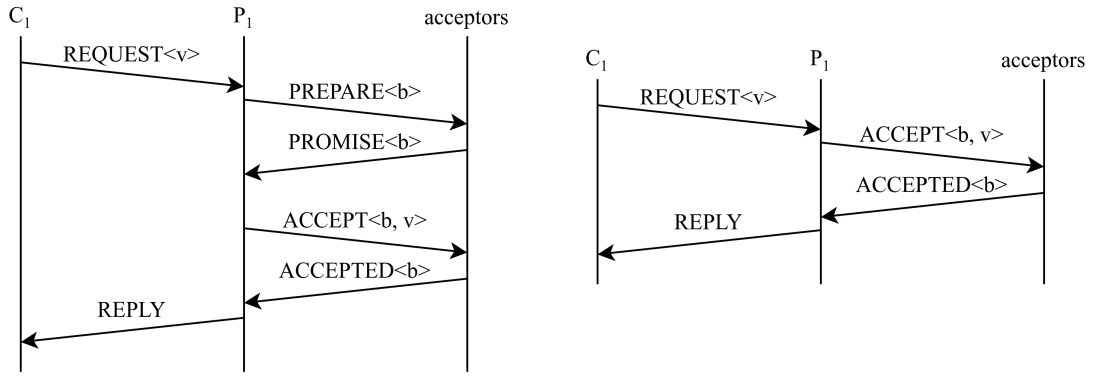
Na seção anterior, o algoritmo Parallel Paxos foi apresentado e especificado. Nesta seção, discutimos otimizações que visam reduzir o número de processos necessários e permitir que decisões sejam alcançadas com um menor número de mensagens e passos de comunicação. As otimizações envolvem a unificação de processos e a pré-execução de fases do algoritmo. Com isso, é possível eliminar parte das trocas de mensagens presentes na especificação original.

Da forma como o algoritmo foi especificado, em uma execução típica a seguinte sequência de mensagens é enviada:

- O cliente envia a requisição para o seu *proposer*. (1 mensagem)
- O *proposer* inicia a fase 1 com os *acceptors*. (f + 1 mensagens)
- Os *acceptors* adotam o número de proposta e respondem para o *proposer*. (f + 1 mensagens)
- O *proposer* inicia a fase 2 com os *acceptors*. (f + 1 mensagens)
- Os *acceptors* aceitam o valor proposto e respondem para o *proposer*. (f + 1 mensagens)
- O *proposer*, agindo como um *learner*, notifica a resposta para o cliente. (1 mensagem)

Então, em uma execução normal, o cliente aprende o resultado da sua requisição em 6 passos de comunicação. Um total de $4f + 6$ mensagens são enviadas. Muitas dessas mensagens podem ser eliminadas com algumas estratégias que serão apresentadas a seguir.

A primeira otimização consiste em fazer com que os *proposers* executem previamente a primeira fase do algoritmo para um número pré-determinado de instâncias. Sendo assim, ao receber requisições do cliente, o *proposer* pode executar diretamente a segunda fase, eliminando $2f + 2$ trocas de mensagens com os *acceptors* e dois passos de comunicação. Com isso, as etapas 2 e 3 da especificação não precisam ser executadas a cada vez que chegar uma nova requisição. A Figura 3.2 mostra duas execuções, uma com e outra sem essa otimização.

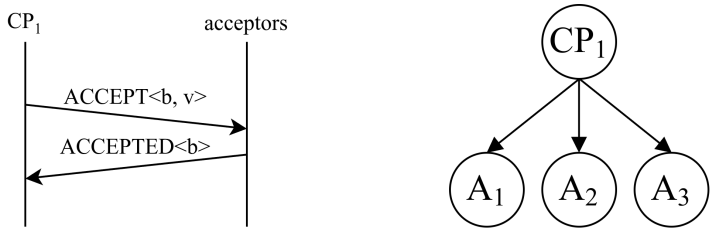


(a) Sem a execução prévia da primeira fase. (b) Com a execução prévia da primeira fase.

Figura 3.2: Duas execuções do Parallel Paxos, uma com e outra sem a execução prévia da primeira fase do Paxos.

Outra possível melhoria consiste em alocar o cliente e o *proposer* em um mesmo processo. Este esquema elimina 2 mensagens — a requisição do cliente para o *proposer* e a resposta do *proposer* para o cliente. Além do mais, aplicando essa coalocação de processos em

conjunto com a pré-execução da primeira fase do algoritmo, o cliente é capaz de decidir um valor após dois passos de comunicação e $2f + 2$ mensagens. A Figura 3.3(a) exibe uma execução do Parallel Paxos com as otimizações vistas até aqui e a Figura 3.3(b) apresenta o esquema de topologia dos processos.



(a) Execução com o cliente e *proposer* coalocados.

(b) Esquema com o cliente e *proposer* coalocados.

Figura 3.3: Parallel Paxos com o cliente e *proposer* coalocados em um mesmo processo.

Ainda é possível coalocar um dos *acceptors* junto com o cliente-*proposer*, reduzindo a necessidade de trocar mensagens com $f + 1$ *acceptors* a cada fase do algoritmo, pois o *proposer* também é um *acceptor*. Dessa forma, com todas as otimizações apresentadas até então, o cliente pode decidir por um valor em 2 passos de comunicação e $2f$ mensagens. A Figura 3.4 apresenta uma execução do algoritmo com 3 *acceptors*, um cliente e um *proposer*. O cliente e o *proposer* estão alocados no mesmo processo junto de um dos *acceptors*. Note que $f = 1$ e no total $2f = 2$ mensagens são enviadas.

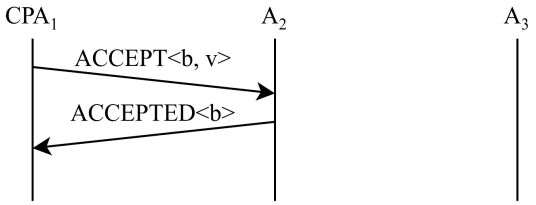


Figura 3.4: Execução do Parallel Paxos com o cliente, *proposer* e um *acceptor* coalocados em um mesmo processo.

Até agora as otimizações foram voltadas para o sistema de um único cliente. Contudo, o Parallel Paxos tem como foco sistemas com múltiplos clientes. Na seção anterior vimos que em sistemas com múltiplos clientes, cada cliente tem o seu próprio *proposer* e cada *proposer* tem o seu próprio conjunto de *acceptors*. Entretanto, é possível que todos os *proposers* utilizem o mesmo conjunto de $2f + 1$ *acceptors* e mesmo assim não concorram pela decisão das instâncias. Para isso, é preciso utilizar alguma estratégia que distribui as instâncias entre os diferentes *proposers* ou então fazer com que cada processo *acceptor* aja como vários *acceptors* em um único processo, um para cada *proposer*. Essas estratégias serão discutidas na próxima seção. A Figura 3.5 mostra o esquema da Figura 3.1(a) com a otimização em que os *acceptors* são compartilhados entre todos os *proposers*.

A seguir, apresenta-se o número de processos e mensagens necessários para o funcionamento de um sistema com três clientes, com e sem as otimizações discutidas anteriormente. Primeiramente, sem a unificação de processos, o sistema conta com três clientes, três *proposers* e nove *acceptors*, totalizando 15 processos. Um cliente recebe a resposta de uma requisição em seis passos de comunicação e $4f + 6$ mensagens são trocadas. Aplicando todas as otimizações discutidas anteriormente, podemos coalocar o cliente, seu *proposer* e um dos *acceptors* em um mesmo processo. Dessa forma, no total são utilizados três processos e um cliente recebe o

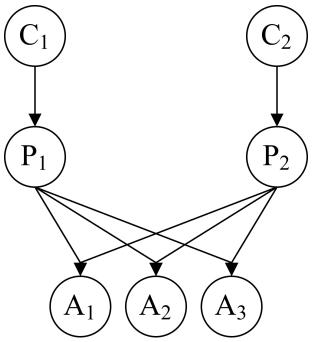


Figura 3.5: Parallel Paxos com *acceptors* compartilhados.

resultado de uma requisição em dois passos de comunicação e $2f$ mensagens, considerando a pré-execução da primeira fase do Paxos. A Figura 3.6 apresenta um sistema com três clientes onde todas as otimizações foram aplicadas.

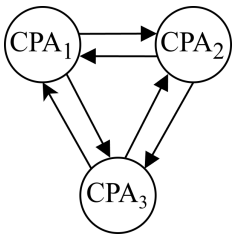


Figura 3.6: Parallel Paxos com três clientes e todas as otimizações aplicadas.

Nesta forma final, um grupo de processos, que estão executando uma aplicação paralela podem todos replicar dados entre si. Dessa maneira, cada processo exerce os papéis de *proposer*, *acceptor* e *learner*. O processo é *proposer* para suas próprias requisições e utiliza os demais processos da aplicação como seus *acceptors*. Sendo assim, não são necessários processos especializados para exercer apenas um papel do Paxos, desde que o número total de processos executando a aplicação paralela seja de pelo menos $2f + 1$ processos. Caso ocorra uma falsa suspeita de falhas e um novo processo seja criado para substituir o processo suspeito, ocorrerá o cenário discutido na seção anterior, em que existem dois *proposers* para um mesmo cliente. Neste caso, como o Parallel Paxos herda as propriedades do Paxos, a consistência do sistema continua garantida — o único impacto é que o cliente com dois *proposers* terá sua sua progressão afetada. Por fim, destaca-se que o Parallel Paxos garante a progressão mesmo na presença dos múltiplos *proposers*, um para cada cliente, ao contrário do Paxos tradicional que necessita de um único *proposer* coordenador.

3.4 COMO MÚLTIPLOS PROPOSERS PODEM UTILIZAR OS MESMOS ACCEPTORS SEM QUE CONCORRAM ENTRE SI

Uma das otimizações apresentadas na seção anterior foi o compartilhamento dos mesmos *acceptors* para diferentes *proposers*, os quais podem propor valores simultaneamente. No entanto, conforme visto na Seção 2.5, a progressão do algoritmo Paxos não é garantida se múltiplos *proposers* estão ativos e propondo valores ao mesmo tempo. Para solucionar esse problema, podemos particionar o conjunto de instâncias e distribuir as partições entre os diferentes *proposers*. Sendo assim, cada *proposer* recebe um conjunto disjunto de instâncias e pode propor apenas nas instâncias que pertencem ao seu conjunto. A divisão das instâncias pode seguir a mesma regra que os *proposers* utilizam para selecionar seus números de propostas (Tabela 2.1). A

Figura 3.7 apresenta uma divisão de instâncias para um sistema com três *proposers*, instâncias representadas com a mesma cor pertencem a um mesmo conjunto. Sendo N o número total de *proposers* do sistema, um *proposer* i obtém seus números de instâncias por meio do conjunto $B = \{i + kN \mid k \in \mathbb{N}\}$, em ordem crescente. Dessa maneira, os diferentes *proposers* do sistema nunca concorrem entre si porque eles nunca propõem nas mesmas instâncias.

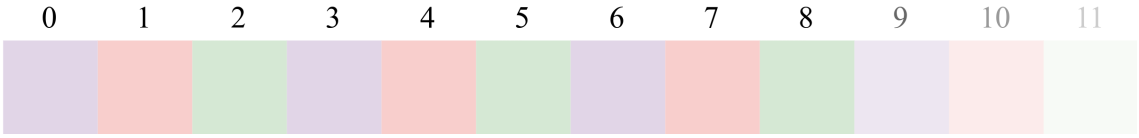


Figura 3.7: Divisão das instâncias do multi-Paxos para três *proposers*.

A variante do Paxos, denominada Mencius (Mao et al., 2008), utiliza a mesma estratégia de divisão de instâncias entre diferentes *proposers*, visando uma melhor distribuição de carga entre eles. No entanto, como ela é utilizada para replicação máquina de estados tradicional, em que uma ordem total entre as requisições é necessária, ainda são utilizados alguns mecanismos para preencher possíveis lacunas. Essas lacunas se formam entre as instâncias devido o desbalanceamento de carga entre os *proposers*. Sendo assim, a divisão de instâncias entre diferentes *proposers*, em que cada *proposer* atende a um único cliente, deve ser utilizada apenas quando uma ordem parcial — por cliente — é o suficiente para a aplicação em questão. Como no Parallel Paxos cada cliente deseja replicar os seus dados e, posteriormente, recuperar apenas os seus próprios dados em ordem, então essa estratégia pode ser utilizada sem problemas. Assim, requisições simultâneas de diferentes clientes não interrompem a decisão uma das outras. O *proposer* coordenador deixa de ser um ponto de gargalo do sistema. No entanto, os *acceptors* terão que lidar com um número maior de propostas simultâneas e podem se tornar o novo gargalo.

Outras alternativas podem envolver o uso de *threads*. Cada processo *acceptor* pode criar uma *thread* para atender a um único *proposer*. Dessa maneira, cada *thread* possui sua própria sequência de execuções do Paxos e os diferentes *proposers* não concorrem entre si. Na prática é como se fossem vários *acceptors* em um só, um em cada *thread*. Note que dessa maneira também não é possível uma ordem total entre as requisições dos diferentes clientes, pois cada *proposer* possui sua própria sequência de instâncias, a fim de evitar a concorrência com outros *proposers*.

4 IMPLEMENTAÇÃO E RESULTADOS EXPERIMENTAIS

Este capítulo descreve a implementação do algoritmo Parallel Paxos e apresenta os resultados dos experimentos realizados para avaliar seu desempenho. Inicialmente, discute-se o cenário em que o Parallel Paxos se faz necessário — aquele em que uma falsa suspeita de falha faz com que existam dois processos coordenadores. Em seguida, a biblioteca LibPaxos é brevemente apresentada. Por fim, são discutidas diferentes abordagens para a implementação do Parallel Paxos, seguidas de uma análise comparativa de desempenho entre elas, considerando o número de decisões por segundo como métrica principal.

4.1 CLIENTE-SERVIDOR: QUANDO DOIS SERVIDORES PARECEM UM SÓ

A Seção 3.1 tratou sobre a replicação passiva e ativa, e discutiu-se a necessidade de adotar o modelo de falhas *fail-stop*, isto é, com detectores de falhas perfeitos (Duarte Jr et al., 2023; Turchetti e Duarte, 2015; Turchetti et al., 2016; Moraes e Duarte Jr, 2011; Duarte Jr et al., 2022; Stein et al., 2023), quando a replicação passiva é empregada em sistemas assíncronos. Isso ocorre devido à possibilidade de que uma falsa suspeita de falha leve o sistema a operar com dois servidores primários simultaneamente, o que pode causar inconsistências entre os *backups*. Sendo assim, antes de partir para os experimentos relacionados à execução do Parallel Paxos, foram realizadas algumas implementações em linguagem C de um simples sistema cliente-servidor, com o objetivo de verificar que o cenário em que dois processos servidores são confundidos como sendo um só pode, de fato, ocorrer.

A primeira implementação realizada consiste em dois processos, um cliente e um servidor UDP. O servidor abre um *socket* e permanece ouvindo em uma determinada porta. Ao receber uma sequência de caracteres por meio de sua entrada padrão, o cliente envia a sequência para o servidor através de uma mensagem UDP. Para cada mensagem recebida, o servidor concatena seu id de processo (pid) e responde de volta para o cliente como uma espécie de confirmação de recebimento (*ack*). Caso o conteúdo da mensagem recebida pelo servidor seja igual a "*fork*", ele invoca a criação de um processo filho por meio da chamada de sistema *fork* disponível no Linux.

A partir do momento que o servidor cria um processo filho, nota-se que o cliente passa a se comunicar com ambos os processos: pai e filho. No entanto, a cada mensagem enviada pelo cliente, apenas um dos servidores recebe a mensagem e, consequentemente, responde com o *ack*. Neste caso, do ponto de vista do cliente, ele está comunicando com um único processo servidor. Como no experimento o pid do servidor foi concatenado em cada resposta, foi possível notar que na realidade são dois processos diferentes respondendo ao cliente. Alterando o protocolo para TCP, o mesmo comportamento ocorre, o cliente continua se comunicando com os dois servidores como se fossem apenas um. A única diferença observada entre o UDP e TCP é que no UDP os servidores ficam alternando na recepção de mensagens do cliente. Por exemplo, o servidor pai recebe a primeira mensagem, o servidor filho recebe a segunda, o pai recebe a terceira e assim por diante. Já no TCP, um processo pode receber várias mensagens antes que o outro passe a receber as próximas, não foi observado um padrão como no UDP.

Após ter observado este comportamento, um novo experimento foi realizado a fim de verificar se essa duplicação de processos poderia causar uma inconsistência em um sistema Primary-Backup. Para isso, dois novos processos, que atuam como *backups*, foram introduzidos no experimento. O código do servidor foi alterado para que, ao receber uma mensagem do

cliente, ele a repasse para ambos os *backups*. Os *backups* simplesmente salvam o último valor recebido do servidor. Após receber a mensagem "*fork*" do cliente, o servidor cria um processo filho. A fim de simular uma alta latência na rede, o processo filho, ao receber uma mensagem do cliente, a repassa imediatamente para o primeiro *backup* e aguarda dois segundos antes de enviar para o segundo *backup*. Enquanto isso, o processo pai envia a mensagem imediatamente para ambos os *backups*.

Neste cenário, foi possível observar uma inconsistência entre os dois *backups*, causada pela coexistência de dois servidores primários em conjunto com um atraso entre as mensagens de um dos servidores. Este atraso pode ocorrer naturalmente em uma rede de computadores. A Figura 4.1 ilustra o experimento realizado. Primeiramente, o cliente envia, quase que ao mesmo tempo, dois valores para o servidor armazenar. O cliente acredita estar se comunicando com um único servidor, mas na prática, devido à duplicação causada pelo *fork*, dois processos diferentes receberam cada uma das mensagens. Em um segundo instante, o servidor S envia o valor a, recebido do cliente, para ambos os *backups*. O servidor S' envia o valor b para o *backup* B_1 imediatamente, mas não envia nada para o *backup* B_2. O *backup* B_1 recebe o valor a, enviado por S, depois de já ter recebido o valor b, enviado por S'. Portanto, o valor a é o que fica salvo em seu armazenamento. Já o *backup* B_2 inicialmente recebe o valor a enviado por S e aproximadamente dois segundos depois recebe o valor b, enviado por S', que chegou atrasado. Logo, o *backup* B_1 salvou a e o *backup* B_2 salvou b, levando o sistema a um estado inconsistente.

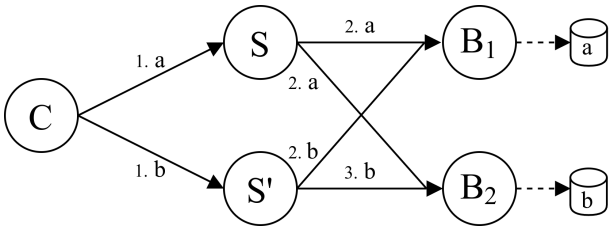


Figura 4.1: Inconsistência em um sistema Primary-Backup com dois servidores primários.

4.2 A BIBLIOTECA LIBPAXOS

A biblioteca LibPaxos (Primi e Sciascia, 2013) é uma coleção de implementações de código-aberto do algoritmo Paxos. A biblioteca teve suas primeiras versões desenvolvidas em Primi (2009) e, posteriormente, novas versões foram desenvolvidas visando melhorar alguns aspectos da implementação. A terceira versão da LibPaxos (Sciascia, 2016) consiste em uma reescrita completa da versão anterior, desenvolvida com foco em uma estrutura de código mais clara e modular. Um exemplo disso é que a terceira versão da biblioteca separa completamente o núcleo do Paxos de qualquer código específico para comunicação entre os processos. Isto significa que essa versão não depende de nenhuma biblioteca de rede em particular.

A LibPaxos 3 implementa os três papéis do Paxos e opera conforme a especificação do algoritmo apresentada no Capítulo 2. Além disso, algumas amostras de uso da biblioteca são disponibilizadas. Entre elas estão os programas cliente, *proposer*, *acceptor*, *learner* e um programa para réplica. A réplica é um processo que emprega os três papéis do Paxos, então ela é capaz de propor e aceitar valores e ainda aprender os valores decididos pelo sistema.

É possível configurar uma série de parâmetros por meio de um arquivo de configuração. Entre estes parâmetros estão os IPs e portas que cada processo irá utilizar, o tipo de armazenamento utilizado pelos *acceptors*, valor de *timeout* das fases do *proposer*, tamanho da janela de pré-execução da primeira fase, entre outros.

4.3 IMPLEMENTAÇÃO E RESULTADOS EXPERIMENTAIS

Uma série de experimentos foram realizados a fim de comparar a vazão, em decisões por segundo, entre o Paxos tradicional e diferentes implementações do Parallel Paxos. A terceira versão da biblioteca LibPaxos foi utilizada para executar os experimentos. Com exceção do primeiro experimento, que serve como base para as comparações, todos os demais foram executados com três clientes e três réplicas da LibPaxos. Lembrando que réplicas são processos que empregam os três papéis do Paxos: *proposer*, *acceptor* e *learner*. Os clientes são processos que enviam valores aleatórios para um *proposer*. Cada cliente executa o papel de *learner* no sistema, desse modo, ele consegue aprender quando seus valores foram decididos. Sendo assim, o próprio cliente aprende e registra o número de decisões por segundo que ele mesmo está conseguindo obter.

Todos os experimentos foram executados em uma mesma máquina, equipada com o sistema operacional Ubuntu 22.04.4 LTS, operando em modo de usuário único, a fim de minimizar a quantidade de processos em execução simultânea e mitigar possíveis interferências nos resultados. A máquina possui processador AMD® Ryzen 3 pro 2100ge e 16GB de memória RAM DDR4. Em todos os experimentos, diferentes réplicas foram alocadas em núcleos distintos do processador, a fim de distribuir melhor a carga de trabalho.

O primeiro experimento realizado teve como objetivo servir de linha de base para comparação com os demais. Ele consiste em um único cliente enviando valores para um dos *proposers*. É possível configurar o número máximo de requisições paralelas para um cliente, por exemplo, se o máximo é 1 então o cliente envia um valor e aguarda a sua decisão antes de enviar o próximo. Se o máximo é 2, então o cliente envia 2 valores e aguarda a decisão de um deles para então enviar o próximo, e assim por diante. Em todos os experimentos, o número máximo de requisições paralela inicia em 1, é elevado para 5 e, em seguida, é incrementado de 5 em 5 até atingir o limite de 50 valores em paralelo. Outro ponto importante é que a janela de pré-execução da primeira fase do algoritmo está configurada para 128 em todos os experimentos realizados, isto é, um *proposer* executa previamente a primeira fase do Paxos para 128 instâncias.

A Figura 4.2 apresenta a alocação dos processos nos núcleos do processador durante o primeiro experimento. Cada réplica é alocada em um núcleo separado, assim como o cliente. O cliente envia valores para uma das réplicas que atua como *proposer* e esta utiliza as outras como *acceptors*. A Figura 4.3 apresenta os resultados deste experimento, o eixo vertical indica o número de decisões por segundo e o eixo horizontal indica o máximo de requisições paralelas que o cliente pode realizar. Cada ponto no gráfico consiste em uma média das decisões por segundo do cliente obtida após 20 segundos de execução. Observando os resultados obtidos, notou-se que o sistema apresenta a vazão máxima em torno de 39 mil requisições por segundo quando o cliente está configurado para emitir no máximo 35 requisições em paralelo.

Dois novos clientes foram adicionados no segundo experimento. O objetivo desse experimento é observar em que medida a vazão de um único cliente é afetada quando múltiplos clientes enviam requisições para o mesmo *proposer*. A Figura 4.4 apresenta a alocação dos processos neste experimento. Podemos perceber, pela Figura 4.5, que a vazão máxima, assim como no experimento anterior, também foi atingida quando o cliente envia até 35 requisições em paralelo. Acreditamos que o resultado obtido evidencia o gargalo causado pela presença de um *proposer* líder. Note que, anteriormente, um único cliente era capaz de decidir até 39 mil valores por segundo e agora, com três clientes, cada um deles consegue decidir em média 11,5 mil valores por segundo. Houve uma queda de cerca de 70% no desempenho de um único cliente. Este experimento serve como base de comparação para as implementações paralelas que serão apresentadas adiante, as quais têm como objetivo obter uma maior vazão por cliente.

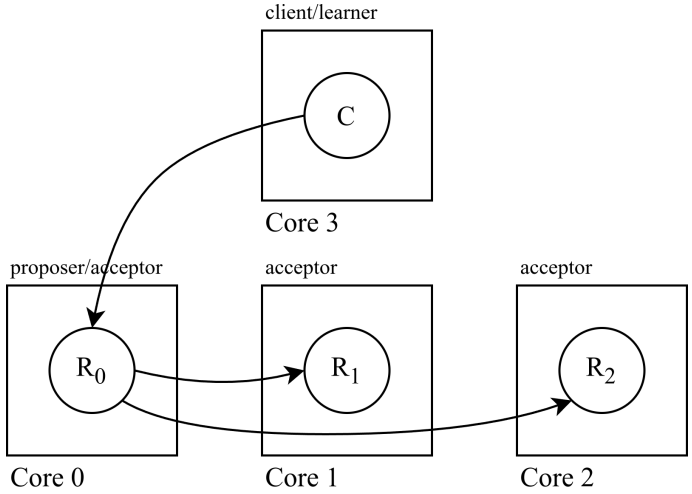


Figura 4.2: Alocação dos processos no experimento com um cliente.

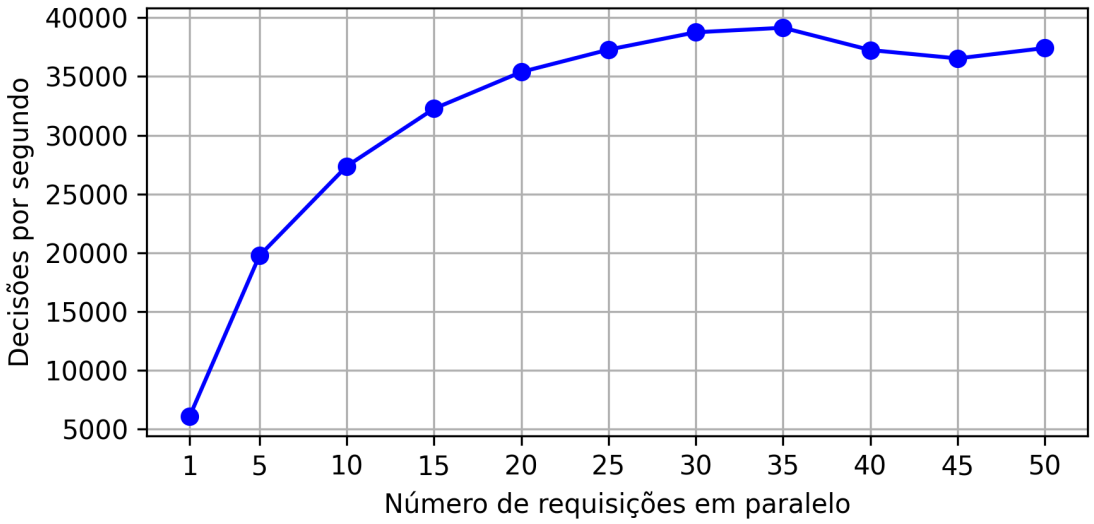


Figura 4.3: Vazão do Paxos com um único cliente.

Para o terceiro experimento, os clientes foram alocados nos mesmos núcleos que as réplicas. Um cliente para cada réplica. No entanto, ainda é necessário que todos os clientes enviem requisições para o mesmo *proposer* líder a fim de garantir a progressão do algoritmo. A Figura 4.6 apresenta a alocação dos processos neste experimento. A Figura 4.7 apresenta o resultado obtido. Neste caso, como os clientes estão compartilhando o processamento com as réplicas, a vazão do sistema reduziu mais uma vez. Os clientes conseguiram decidir em média 8,2 mil requisições por segundo, uma queda de aproximadamente 79% quando comparado com um único cliente. Ainda é possível notar que o cliente alocado no mesmo núcleo que o *proposer* líder teve sua vazão mais afetada do que os demais clientes.

Antes de partir para as soluções paralelas, foi conduzido um experimento em que cada cliente envia valores a um *proposer* diferente — ou seja, não há um *proposer* líder. A Figura 4.8 apresenta a alocação dos processos neste experimento. A Figura 4.9 apresenta os resultados, os quais estiveram de acordo com o esperado. A progressão não é garantida na ausência de um *proposer* único e, portanto, os clientes concorrem entre si. Dessa forma, na maior parte do tempo apenas um dos clientes obtém sucesso nas suas requisições.

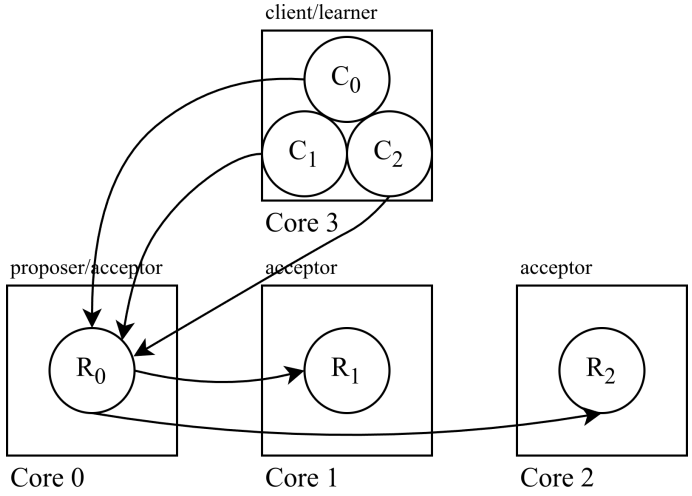


Figura 4.4: Alocação dos processos no experimento com três clientes isolados em um único núcleo.

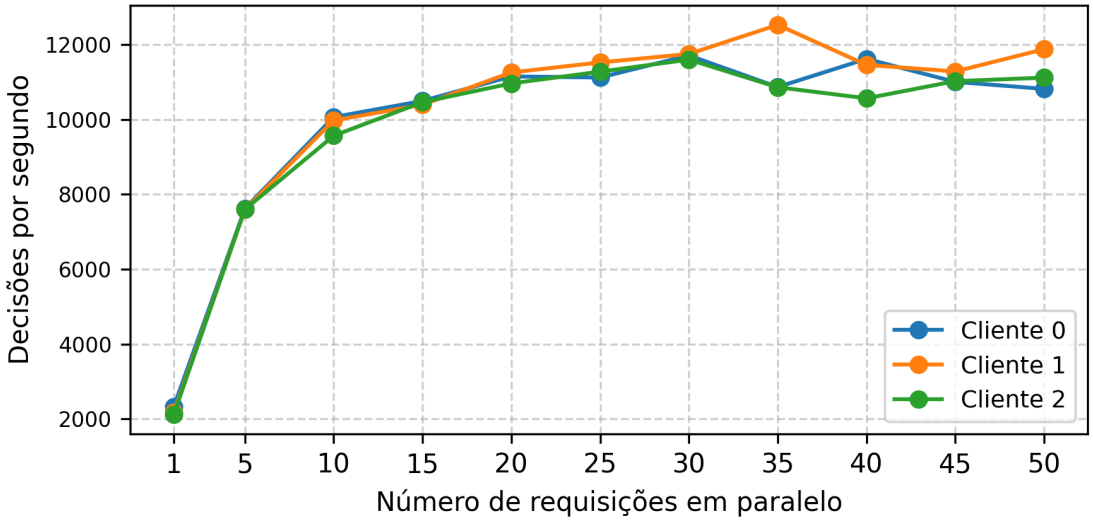


Figura 4.5: Vazão do Paxos com três clientes isolados em um único núcleo.

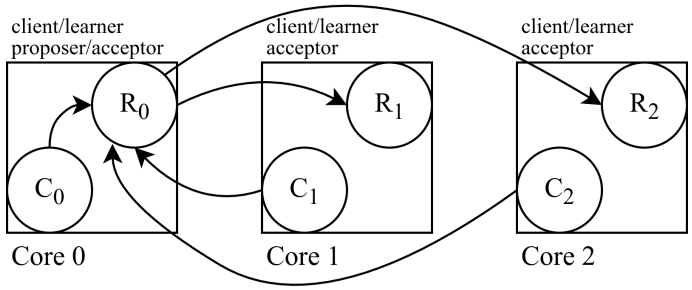


Figura 4.6: Alocação dos processos no experimento com três clientes alocados no mesmo núcleo que as réplicas.

Partindo para as soluções paralelas, três implementações do Parallel Paxos foram realizadas. Na primeira delas, cada cliente tem seu conjunto distinto de três réplicas. Na prática, são três sistemas distintos executando o Paxos e cada cliente utiliza um deles. A Figura 4.10 apresenta a alocação dos processos durante este experimento. Cada cliente tem seu próprio conjunto de réplicas e comunica apenas com o líder de suas réplicas. Os resultados obtidos,

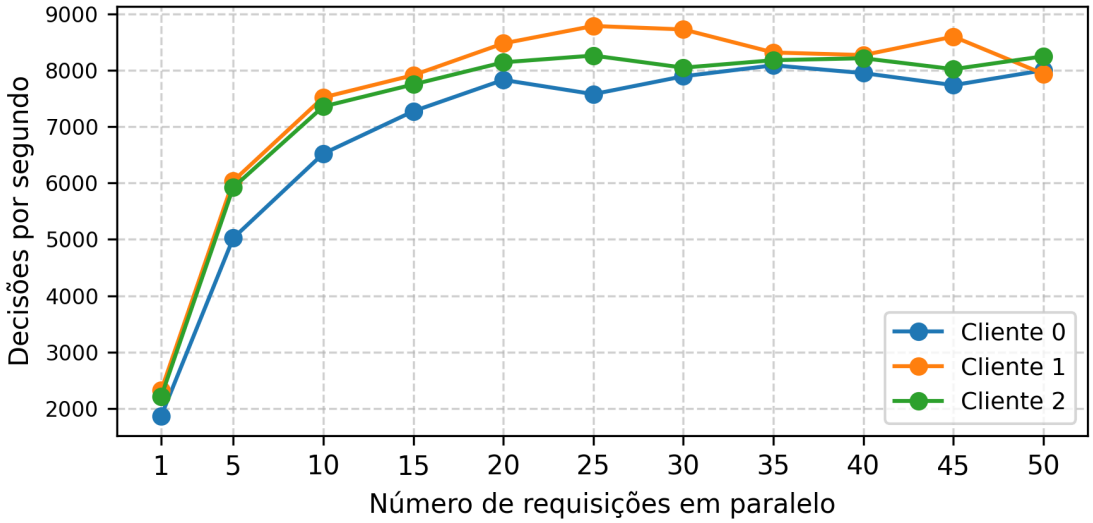


Figura 4.7: Vazão do Paxos com três clientes alocados junto com as réplicas.

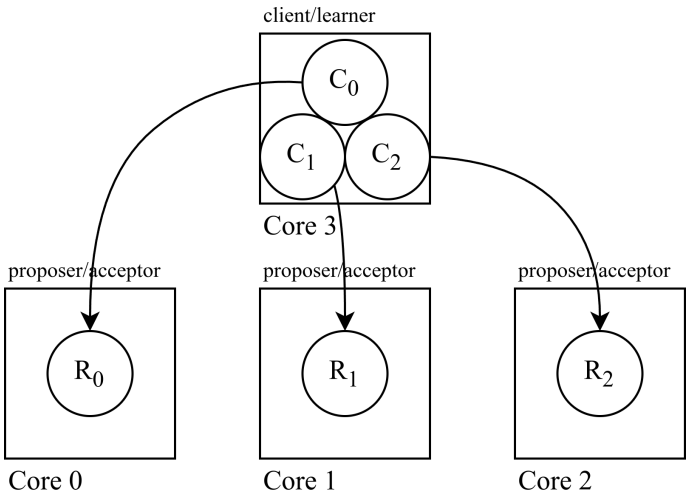


Figura 4.8: Alocação dos processos no experimento com três clientes enviando requisições para diferentes réplicas. Cada réplica utiliza as demais como seus *acceptors*. As flechas que indicam a comunicação entre as réplicas foram omitidas.

presentes na Figura 4.11, mostram que, nessa implementação do Parallel Paxos, os clientes atingiram uma vazão média de 15,9 mil requisições por segundo. Verificou-se um ganho de cerca de 38% na vazão por cliente em relação ao uso do Paxos tradicional com três clientes (Figura 4.5).

Ao contrário da abordagem paralela anterior, o próximo experimento utiliza *threads* para representar as réplicas. Em vez de nove processos distintos, há apenas três, cada um contendo três *threads*, uma para cada réplica. A Figura 4.12 apresenta a alocação dos processos durante este experimento. Assim como no experimento anterior, cada cliente tem seu conjunto de réplicas, mas essas réplicas são *threads* dentro de um processo. A vazão obtida na abordagem com *threads* foi superior à abordagem com processos. Acredita-se que a razão disso é o menor número de trocas de contexto entre processos no sistema operacional. Conforme a Figura 4.13, os clientes conseguiram decidir, em média, 16,8 mil valores por segundo. Houve um ganho de 46% quando

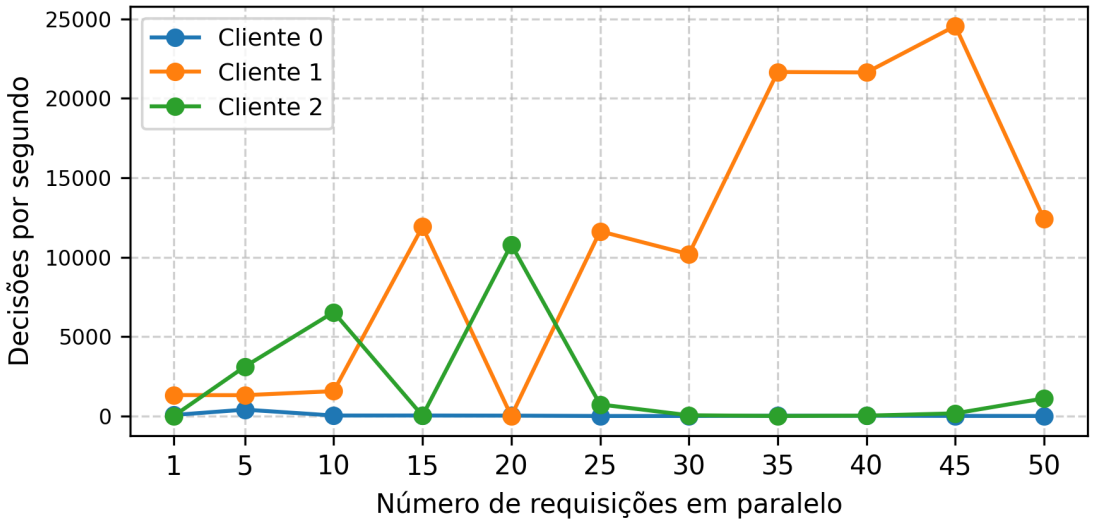


Figura 4.9: Vazão do Paxos com três clientes concorrentes.

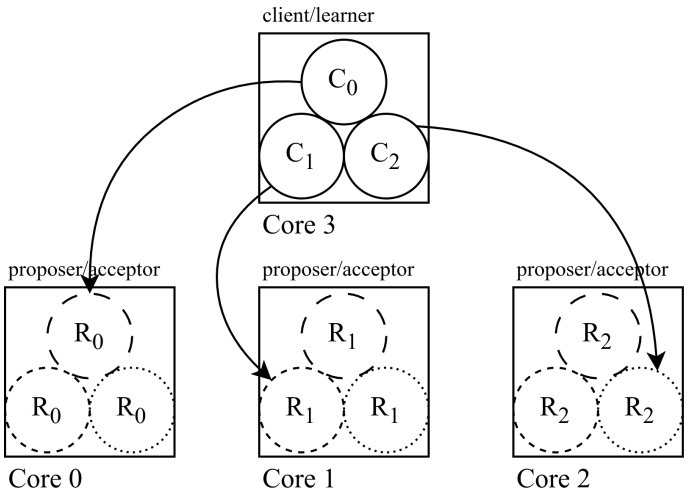


Figura 4.10: Alocação dos processos no experimento com três sistemas Paxos em paralelo. Cada cliente envia requisições para o seu próprio conjunto de réplicas. As réplicas representadas com o mesmo estilo de borda pertencem ao mesmo sistema. Cada réplica utiliza as demais, do mesmo sistema, como seus *acceptors*. As flechas que indicam a comunicação entre as réplicas foram omitidas.

comparado com o paxos tradicional com três clientes, e um ganho de 5.6% quando comparado com a abordagem paralela do experimento anterior.

As implementações paralelas vistas até agora foram, na prática, vários sistemas executando o Paxos separadamente. Uma delas utilizou diferentes processos para as réplicas e a outra utilizou *threads*. A última implementação realizada explora paralelismo dentro de um mesmo sistema Paxos, isto é, existem apenas três réplicas e três clientes. Cada cliente propõe valores para uma das réplicas e elas não concorrem entre si. No entanto, para que isso seja possível, a aplicação em questão não deve depender de uma ordem total entre as requisições de todos os clientes — na verdade, isso ocorre em todos os experimentos dessa seção. Em de Camargo (2017), o Parallel Paxos é utilizado para salvar as informações necessárias para que, após uma falha, um processo (cliente) seja capaz de recuperar o seu estado anterior à falha. Desse modo, uma ordem total entre as requisições de todos os clientes não é necessária, uma ordem parcial

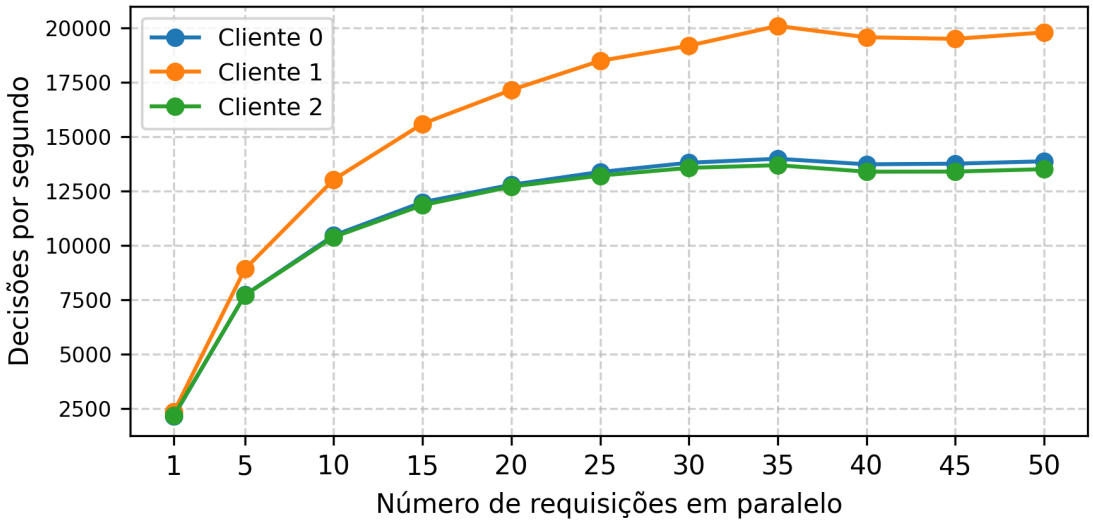


Figura 4.11: Vazão do Parallel Paxos implementado com três sistemas Paxos distintos.

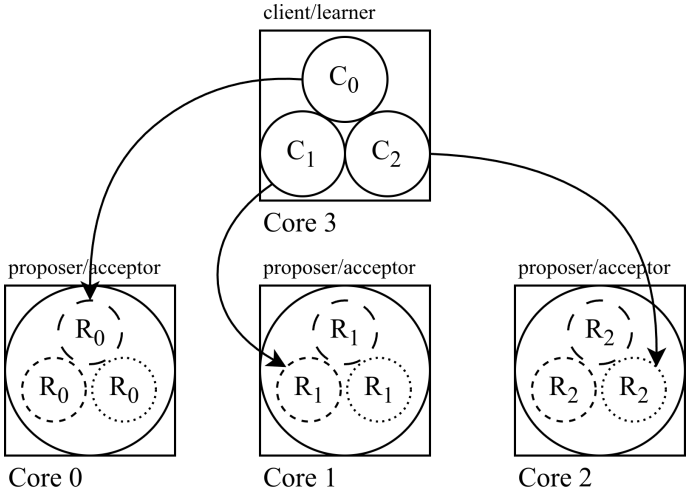


Figura 4.12: Alocação dos processos no experimento com três sistemas Paxos em paralelo utilizando *threads*. Cada cliente envia requisições para o seu próprio conjunto de réplicas. Uma réplica é uma *thread* dentro de um processo maior. As réplicas representadas com o mesmo estilo de borda pertencem ao mesmo sistema. Cada réplica utiliza as demais, do mesmo sistema, como seus *acceptors*. As flechas que indicam a comunicação entre as réplicas foram omitidas.

por cliente supre a necessidade da aplicação. Cada cliente recupera apenas as suas informações em ordem.

O próximo experimento utilizou a estratégia de divisão de instâncias apresentada na Seção 3.4. A Figura 4.14 apresenta a alocação dos processos durante o experimento. Note que cada cliente envia requisições para um *proposer* diferente. Os resultados obtidos com essa abordagem, Figura 4.15, demonstram que, em média, um cliente consegue decidir até 18,5 mil requisições por segundo. Houve um ganho de aproximadamente 61% quando comparado com o Paxos tradicional com três clientes. Quando comparada com as outras duas implementações paralelas apresentadas anteriormente, houve um ganho de cerca de 16% e 10%, respectivamente.

O gráfico da Figura 4.16 consolida os resultados de todos os experimentos realizados com três clientes. Podemos perceber que as três implementações paralelas foram as que apresentaram a maior vazão média por cliente. O experimento com três clientes enviando requisições para

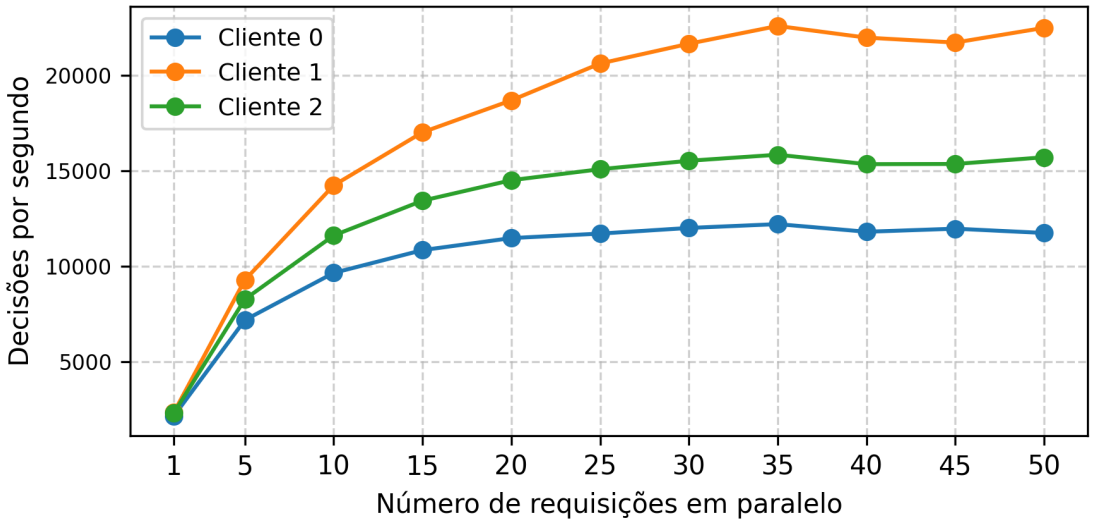


Figura 4.13: Vazão do Parallel Paxos implementado com três sistemas Paxos distintos.

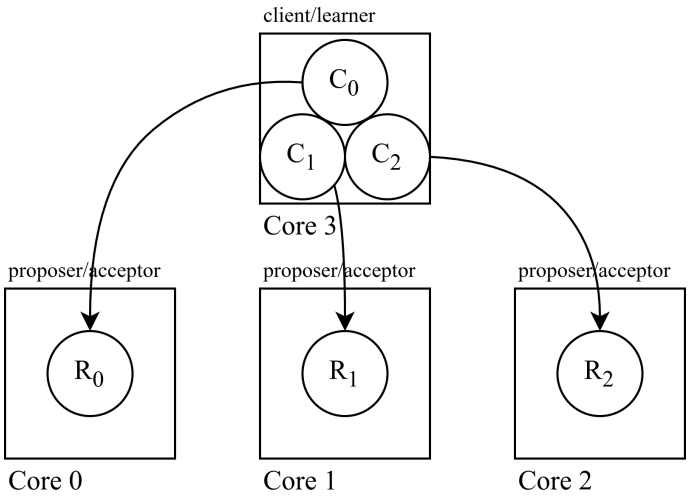


Figura 4.14: Alocação dos processos no experimento com três clientes em que cada *proposer* possui seu próprio conjunto disjunto de instâncias. As flechas que indicam a comunicação entre as réplicas foram omitidas.

três *proposers* distintos apresentou o pior resultado, como já era esperado, pois os *proposers* concorrem entre si.

Em muitas aplicações, os clientes emitem requisições de maneira pessimista, isto é, um cliente emite uma requisição e aguarda o seu resultado antes de prosseguir sua execução. Por conta disso, a Figura 4.17 apresenta o resultado consolidado da vazão média por cliente quando cada cliente emite no máximo uma requisição por vez. Por meio desta figura, podemos perceber que o ganho de desempenho das implementações paralelas é pequeno quando os clientes operam de maneira pessimista. No entanto, a Figura 4.16 mostra que, conforme os clientes aceitam um maior número de requisições em andamento, a diferença entre as soluções paralelas e as soluções com Paxos tradicional cresce consideravelmente.

Até o momento, as análises realizadas compararam a vazão média por cliente. No entanto, outra métrica interessante é a vazão média total do sistema, isto é, a soma da vazão de todos os clientes. O primeiro experimento, realizado com um único cliente, teve como objetivo servir de comparação para a vazão total. Neste experimento (Figura 4.3), um único cliente foi

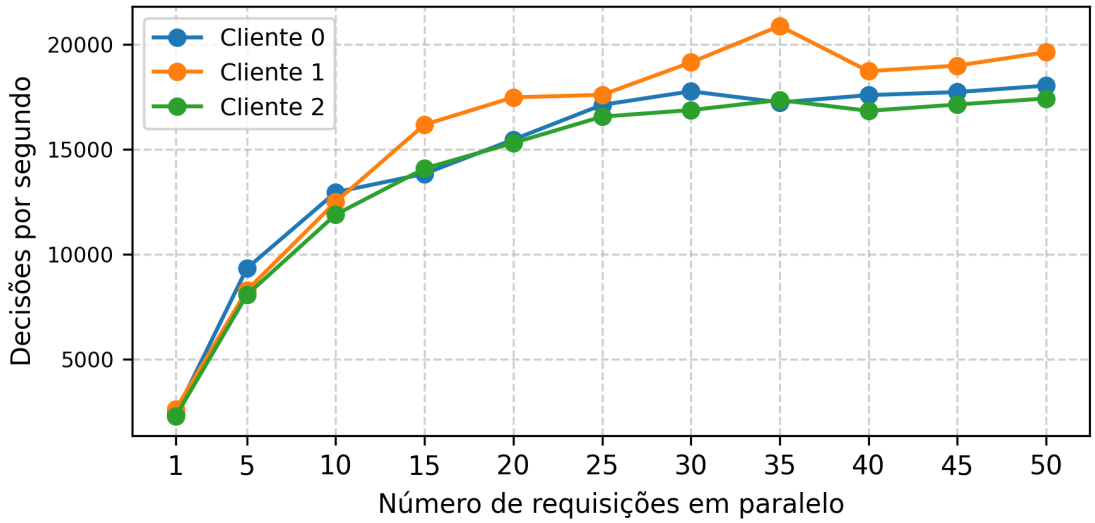


Figura 4.15: Vazão do Parallel Paxos implementado com a divisão de instâncias entre os *proposers*.

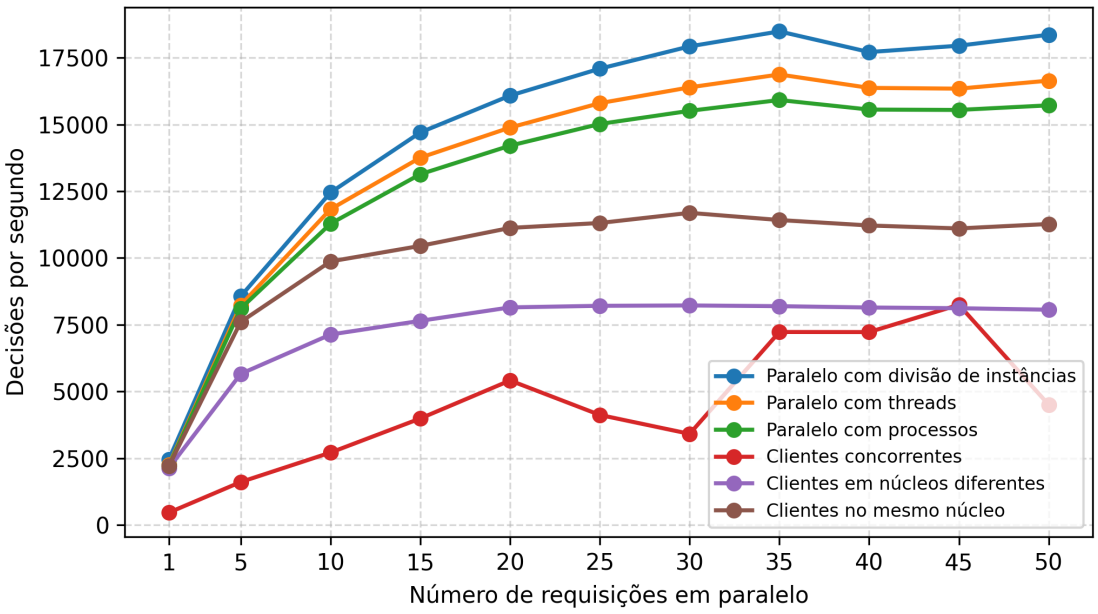


Figura 4.16: Vazão média por cliente nos experimentos realizados.

capaz de decidir até 39 mil requisições por segundo. A Figura 4.18 apresenta os resultados da vazão total do sistema nos diferentes experimentos. Note que, as implementações paralelas foram as únicas que conseguiram obter uma vazão total maior do que a vazão no sistema com um único cliente.

Por fim, a Figura 4.19 apresenta a vazão total do sistema com os clientes emitindo requisições de maneira pessimista — uma requisição por vez. Em conjunto com a Figura 4.18, é possível perceber que quando os clientes emitem poucas requisições paralelas, a vazão total no Paxos tracional com múltiplos clientes até consegue superar a vazão de um único cliente. No entanto, a partir de 15 requisições em paralelo, o sistema com três clientes começa a apresentar uma vazão total menor. Esta observação evidencia o gargalo causado pelo *proposer* líder, essencial para a progressão do Paxos tradicional.

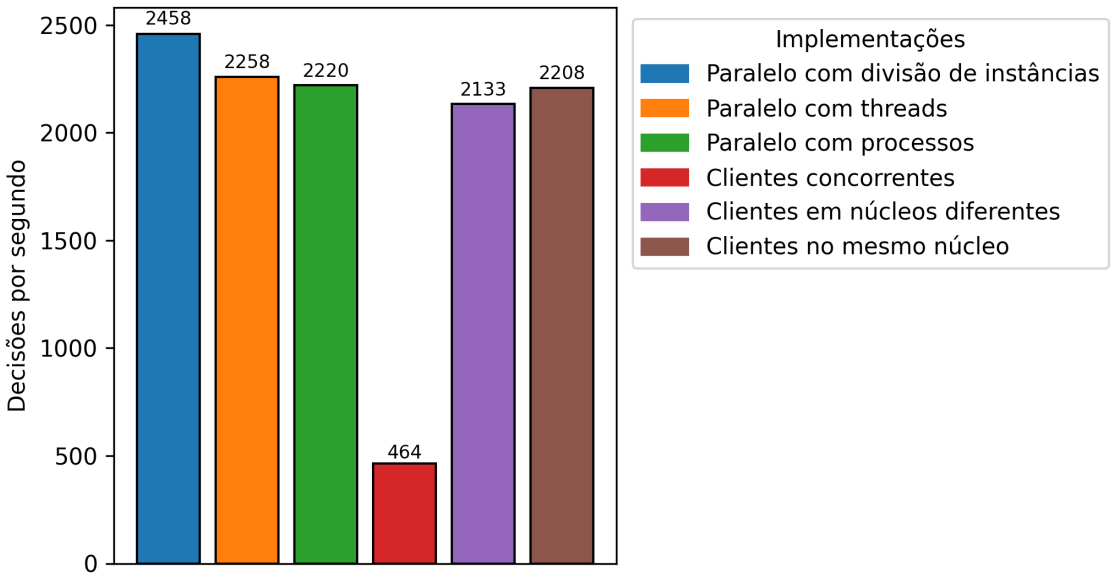


Figura 4.17: Vazão média por cliente nos experimentos realizados com os clientes emitindo requisições de maneira pessimista (1 requisição por vez).

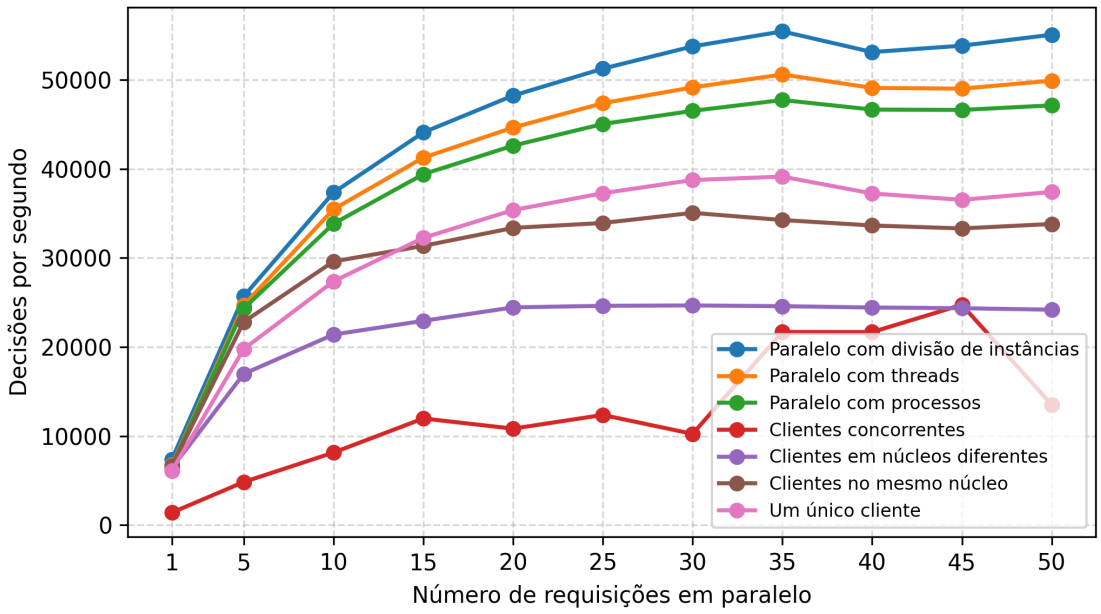


Figura 4.18: Vazão total média do sistema nos experimentos realizados.

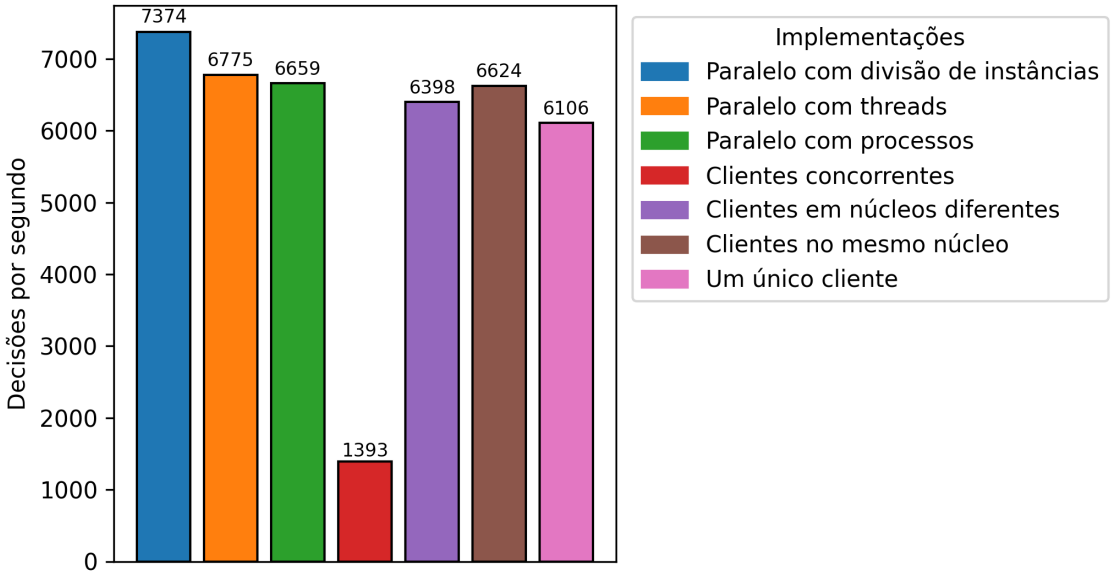


Figura 4.19: Vazão total média do sistema nos experimentos realizados com os clientes emitindo requisições de maneira pessimista (1 requisição por vez).

5 CONCLUSÃO

Este trabalho explorou o algoritmo Parallel Paxos para replicação distribuída tolerante a falhas. O algoritmo herda as propriedades do consenso garantidas pelo Paxos tradicional, mas adapta sua estrutura para explorar paralelismo e melhorar o desempenho em ambientes distribuídos de alto desempenho. O algoritmo emprega uma estratégia multi-líder a fim de eliminar o gargalo causado pelo coordenador único presente no Paxos tradicional. Além disso, foi comprovado por meio de experimentos que a replicação passiva, ou Primary-Backup, não é capaz de garantir a consistência dos dados replicados quando uma falsa suspeita de falhas faz com que dois coordenadores coexistam no sistema. Já o Parallel Paxos, garante a consistência do sistema sob quaisquer circunstâncias, pois herda as propriedades do Paxos.

Diversos experimentos foram realizados, utilizando a biblioteca LibPaxos, a fim de comparar diferentes implementações do Parallel Paxos com o Paxos tradicional. As métricas observadas foram a vazão média por cliente a vazão total do sistema, em decisões por segundo. O Parallel Paxos foi implementado de diferentes maneiras: com as réplicas executando em processos separados, em *threads* separadas dentro de um mesmo processo ou um único conjunto de réplicas cujas instâncias foram particionadas entre os *proposers*. Os resultados obtidos demonstraram um aumento na vazão por cliente de até 38%, 46% e 61% em relação ao paxos tradicional, respectivamente, para cada uma das implementações realizadas. Em particular, destaca-se que, dos experimentos com três clientes, as implementações paralelas foram as únicas capazes de superar a vazão total do Paxos tradicional com um único cliente. Ou seja, os resultados indicam que há um gargalo no *proposer* coordenador, e eliminar este processo centralizado melhora a vazão do sistema como um todo.

Trabalhos futuros incluem o estudo de topologias para permitir que os processos de um sistema paralelo possam replicar seus dados entre si executando o Parallel Paxos. Um exemplo de topologia que os processos poderiam formar inclui a $D_{1,2}$ (Duarte Jr et al., 1997; Duarte et al., 2012), em que um processo seleciona outros dois processos para serem seus *backups* ou, de maneira geral, topologias $D_{n,k}$. Além disso, são desejáveis experimentos em que os processos estão alocados em máquinas distintas (não apenas núcleos distintos), a fim de investigar o impacto que a comunicação da rede causa na vazão do sistema.

REFERÊNCIAS

Aguilera, M. K., Delporte-Gallet, C., Fauconnier, H. e Toueg, S. (2001). Stable leader election. Em *Proceedings of the 15th International Conference on Distributed Computing*, DISC '01, página 108–122. Springer-Verlag.

Alsberg, P. A. e Day, J. D. (1976). A principle for resilient sharing of distributed resources. Em *Proceedings of the 2nd International Conference on Software Engineering*, ICSE '76, página 562–570. IEEE Computer Society Press.

Cachin, C., Guerraoui, R. e Rodrigues, L. (2011). *Introduction to Reliable and Secure Distributed Programming*. Springer Publishing Company, Incorporated, 2nd edition.

Chandra, T. D. e Toueg, S. (1991). Unreliable failure detectors for asynchronous systems. Em *Proceedings of the Tenth Annual ACM Symposium on Principles of Distributed Computing*, página 325–340. Association for Computing Machinery.

Charron-Bost, B., Pedone, F. e Schiper, A. (2010). *Replication: theory and Practice*. Springer-Verlag, Berlin, Heidelberg.

Corbett, J. C., Dean, J., Epstein, M. et al. (2013). Spanner: Google's globally distributed database. *ACM Transactions on Computer Systems (TOCS)*, 31(3):1–22.

de Camargo, E. T. (2017). *Tolerância a Falhas em Sistemas MPI com Grupos Dinâmicos de Processos Recomendados e Registro de Mensagens Distribuído Baseado em Paxos*. Tese de doutorado, Universidade Federal do Paraná, Curitiba - PR.

de Camargo, E. T., Duarte Jr, E. P. e Pedone, F. (2017a). A consensus-based fault-tolerant event logger for high performance applications. Em *European Conference on Parallel Processing*, páginas 415–427. Springer.

de Camargo, E. T., Pedone, F. e Duarte Jr, E. P. (2017b). Um protocolo pessimista para registro de mensagens baseado em um event logger distribuído e tolerante a falhas. Em *Anais do XXXV Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos*. SBC.

Duarte, E. e dos Santos, A. L. (2001). Network fault management based on snmp agent groups. Em *Proceedings 21st International Conference on Distributed Computing Systems Workshops*, páginas 51–56. IEEE.

Duarte, E. P., Weber, A. e Fonseca, K. V. O. (2012). Distributed diagnosis of dynamic events in partitionable arbitrary topology networks. *IEEE Transactions on Parallel and Distributed Systems*, 23(8):1415–1426.

Duarte Jr, E. P. e Godoi, A. F. B. (2014). Reliable content distribution in p2p networks based on peer groups. *International Journal of Internet and Distributed Systems*, 2(2):5–14.

Duarte Jr, E. P., Nanya, T., Noguchi, S. e Mansfield, G. (1997). Non-broadcast network fault-monitoring based on system-level diagnosis. Em *International Symposium on Integrated Network Management*, páginas 597–609. Springer.

Duarte Jr, E. P., Rodrigues, L. A., Camargo, E. T. e Turchetti, R. (2022). A distributed system-level diagnosis model for the implementation of unreliable failure detectors. *arXiv preprint arXiv:2210.02847*.

Duarte Jr, E. P., Rodrigues, L. A., Camargo, E. T. e Turchetti, R. C. (2023). The missing piece: a distributed system-level diagnosis model for the implementation of unreliable failure detectors. *Computing*, 105(12):2821–2845.

Fischer, M. J., Lynch, N. A. e Paterson, M. S. (1985). Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 35(2):374–382.

Huff, A., Hiltunen, M. e Duarte, E. P. (2021). Rft: Scalable and fault-tolerant microservices for the o-ran control plane. Em *2021 IFIP/IEEE International Symposium on Integrated Network Management (IM)*, páginas 402–409. IEEE.

Hunt, P., Konar, M., Junqueira, F. P. e Reed, B. (2010). Zookeeper: Wait-free coordination for internet-scale systems. Em *2010 USENIX Annual Technical Conference (USENIX ATC 10)*, página 11. USENIX Association.

Lamport, L. (1978). Time, clocks and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565.

Lamport, L. (1982). The byzantine generals problem. *ACM transactions on programming languages and systems*, 4(3):382–401.

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*, 32(4):51–58.

Lynch, N. A. (1996). *Distributed Algorithms*. Morgan Kaufmann Publishers Inc.

Mao, Y., Junqueira, F. P. e Marzullo, K. (2008). Mencius: building efficient replicated state machines for wans. Em *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, página 369–384. USENIX Association.

Moraes, D. M. e Duarte Jr, E. P. (2011). A failure detection service for internet-based multi-as distributed systems. Em *2011 IEEE 17th International Conference on Parallel and Distributed Systems*, páginas 260–267. IEEE.

Moraru, I., Andersen, D. G. e Kaminsky, M. (2013). There is more consensus in egalitarian parliaments. Em *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, página 358–372. Association for Computing Machinery.

Ongaro, D. e Ousterhout, J. (2014). In search of an understandable consensus algorithm. Em *2014 USENIX annual technical conference (USENIX ATC 14)*, páginas 305–319.

Primi, M. (2009). Paxos made code: Implementing a high throughput atomic broadcast. Dissertação de Mestrado, Faculty of Informatics of the University of Lugano, Lugano - Suiça.

Primi, M. e Sciascia, D. (2013). Libpaxos: Open-source paxos. `https://libpaxos.sourceforge.net/`. Acessado em 02/06/2025.

Rodrigues, L. A., Arantes, L. e Duarte, E. P. (2016). An autonomic majority quorum system. Em *2016 IEEE 30th International Conference on Advanced Information Networking and Applications (AINA)*, páginas 524–531. IEEE.

Ruchel, L. V., de Camargo, E. T., Rodrigues, L. A., Turchetti, R. C., Arantes, L. e Duarte, E. P. (2024). Scalable atomic broadcast: A leaderless hierarchical algorithm. *Journal of Parallel and Distributed Computing*, 184:104789.

Schneider, F. B. (1990). Implementing fault-tolerant services using the state machine approach: a tutorial. *ACM Computing Surveys*, 22(4):299–319.

Sciascia, D. (2016). sciascid/libpaxos — bitbucket. `https://bitbucket.org/sciascid/libpaxos`. Acessado em 02/06/2025.

Stein, G., Rodrigues, L. A., Duarte Jr, E. P. e Arantes, L. (2023). Diamond-p-vcube: An eventually perfect hierarchical failure detector for asynchronous distributed systems. Em *Proceedings of the 12th Latin-American Symposium on Dependable and Secure Computing*, páginas 40–49.

Turchetti, R. C. e Duarte, E. P. (2015). Implementation of failure detector based on network function virtualization. Em *2015 IEEE International Conference on Dependable Systems and Networks Workshops*, páginas 19–25. IEEE.

Turchetti, R. C., Duarte Jr, E. P., Arantes, L. e Sens, P. (2016). A qos-configurable failure detection service for internet applications. *Journal of Internet Services and Applications*, 7(1):9.

Turek, J. e Shasha, D. (2002). The many faces of consensus in distributed systems. *Computer*, 25(6):8–17.

van Renesse, R. e Schneider, F. B. (2004). Chain replication for supporting high throughput and availability. Em *Proceedings of the 6th Conference on Symposium on Operating Systems Design & Implementation*, páginas 91–104. USENIX Association.

Van Renesse, R. V. e Altinbunken, D. (2015). Paxos made moderately complex. *ACM Computing Surveys*, 47(3):1–36.

Venâncio, G. e Duarte Jr, E. P. (2022). Nham: An nfv high availability architecture for building fault-tolerant stateful virtual functions and services. Em *Proceedings of the 11th Latin-American Symposium on Dependable Computing*, páginas 35–44.

Venâncio, G., Turchetti, R. C., de Camargo, E. T. e Duarte, E. P. (2021). Vnf-consensus: A virtual network function for maintaining a consistent distributed software-defined network control plane. *International Journal of Network Management*, 31(3):e2124.

Venâncio, G., Turchetti, R. C. e Duarte, E. P. (2019). Nfv-rbcast: Enabling the network to offer reliable and ordered broadcast services. Em *2019 9th Latin-American Symposium on Dependable Computing (LADC)*, páginas 1–10. IEEE.

Wiesmann, M., Pedone, F., Schiper, A., Kemme, B. e Alonso, G. (2000). Understanding replication in databases and distributed systems. Em *Proceedings 20th IEEE International Conference on Distributed Computing Systems*, páginas 464–474. IEEE.