Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Protocolo NOPaxos Tolerante a Falhas Bizantinas

Gabriel Faustino Lima da Rocha

Monografia apresentada como requisito parcial
para conclusão do Bacharelado em Ciência da Computação

Orientador
Prof. Dr. Eduardo Adilo Pelinson Alchieri

Brasília
2024

Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Protocolo NOPaxos Tolerante a Falhas Bizantinas

Gabriel Faustino Lima da Rocha

Monografia apresentada como requisito parcial
para conclusão do Bacharelado em Ciência da Computação

Prof. Dr. Eduardo Adilo Pelinson Alchieri (Orientador)
CIC/UnB

Prof. Dr. Edison Ishikawa
Universidade de Brasília

Prof. Dr. Marcos Fagundes Caetano
Universidade de Brasília

Prof. Dr. Marcelo Grandi Mandelli
Coordenador do Bacharelado em Ciência da Computação

Brasília, 30 de Agosto de 2024

Dedicatória

A minha mãe, que sempre me incentivou a buscar conhecimento, e ao meu pai, que esteve ao meu lado em todos os momentos, oferecendo seu apoio incondicional.

Agradecimentos

Primeiramente agradeço á Deus por ser me guiar durante todos esses anos. A meus pais, Jefferson e Cláudia, por terem sempre feito o possível para me ajudar e incentivaram a sempre buscar conhecimento e me aprimorar pessoalmente e profissionalmente. Quero agradecer também a minha noiva Letícia que sempre me apoiou e me ajudou durante esse processo final. A minha irmã pela parceria e amizade. Aos meus avós pela torcida e apoio durante esse processo.

Também quero agradecer os meus professores e professoras da universidade de brasília que sempre desempenharam um excelente papel como docentes e tenho muito a agradecer pelo conhecimento que me foi passado durante os anos que estive na universidade. Ao professor Dr. Eduardo Adilo Pelinson Alchieri pela orientação, profissionalismo e dedicação.

A toda equipe do Emulab por pelo apoio prestado e por disponibilizarem as máquinas utilizadas para o experimento, sem eles não seria possível concluir este trabalho.

Por fim gostaria de agradecer a todos os amigos que fiz durante o curso, em especial, Enzo, Gustavo, Pedro e Gabriel que me proporcionaram grandes momentos dentro e fora da universidade.

O presente trabalho foi realizado com apoio da Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES), por meio do Acesso ao Portal de Periódicos.

Resumo

Este trabalho apresenta uma análise comparativa e a integração de dois algoritmos de replicação de estado de máquinas voltados para tolerância a falhas, com foco na aplicação de técnicas avançadas para melhorar o desempenho em cenários distribuídos. O primeiro algoritmo, NoPaxos, propõe uma abordagem que elimina grande parte da sobrecarga tradicional do Paxos, utilizando um mecanismo de ordenação de mensagens na camada de rede, denominado *Ordered Unreliable Multicast* (OUM). Esse protocolo busca atingir alta consistência de réplica sem a necessidade de coordenação em todas as requisições, o que resulta em uma latência e *throughput* significativamente próximos aos de um sistema não replicado.

O segundo algoritmo avaliado é uma versão modificada do NoPaxos, ajustada para tolerar falhas bizantinas, utilizando o mecanismo de *Unique Sequential Identifier Generator* (USIG). A modificação apesar de simples gera alguns custos computacionais a mais devido a assinatura e verificação de assinatura, e do ponto de vista lógico o armazenamento da chave privada.

Os experimentos realizados demonstraram que, apesar da adição de complexidade para suportar falhas bizantinas, a arquitetura combinada mantém um desempenho aceitável, com um aumento moderado de latência devido ao tempo adicional necessário para a assinatura e verificação de pacotes. A integração desses algoritmos propõe um novo paradigma para a replicação de máquinas de estado em sistemas distribuídos, oferecendo um equilíbrio entre desempenho e tolerância a falhas, adequado para aplicações críticas em *data centers* modernos.

Palavras-chave: falhas bizantinas, replicação, consenso

Abstract

This work presents a comparative analysis and integration of two state machine replication algorithms focused on fault tolerance, with an emphasis on applying advanced techniques to enhance performance in distributed scenarios. The first algorithm, NoPaxos, introduces an approach that eliminates much of the traditional Paxos overhead by utilizing a network-layer message ordering mechanism called Ordered Unreliable Multicast (OUM). This protocol aims to achieve high replica consistency without requiring coordination on every request, resulting in latency and throughput that are significantly close to those of an unreplicated system.

The second algorithm evaluated is a modified version of NoPaxos, adapted to tolerate Byzantine faults using the Unique Sequential Identifier Generator (USIG) mechanism. Although this modification is relatively simple, it incurs additional computational costs during the signing and verification processes, as well as the logical storage of the private key.

The experiments conducted demonstrated that, despite the added complexity to support Byzantine faults, the combined architecture maintains acceptable performance, with a moderate increase in latency due to the additional time required for packet signing and verification. The integration of these algorithms proposes a new paradigm for state machine replication in distributed systems, offering a balance between performance and fault tolerance, suitable for critical applications in modern data centers.

Keywords: Byzantine failures , replication, consensus

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

Um sistema distribuído é frequentemente estruturado em termos de clientes e serviços. Cada serviço é composto por um ou mais servidores que disponibilizam operações para os clientes utilizarem por meio de requisições. São consideradas falhas nesses sistemas quando o comportamento não condiz com as especificações a ele atribuídas [2].

Essas falhas podem ser causadas por problemas comuns como defeito no hardware, falta de energia, perda de comunicação entre outros fatores. Essas falhas são as chamadas falhas por parada ou falhas por crash e são caracterizadas pela interrupção súbita e inesperada do funcionamento de um ou mais componentes do sistema. Existem também as falhas bizantinas, que são atribuídas a um dos componentes do sistema agir de forma maliciosa, como enviar informações incorretas ou conflitantes para diferentes partes do sistema, comprometendo a integridade e a consistência dos dados processados.

Para lidar com as falhas foram criados diversos algoritmos de consenso. Porém, o custo computacional para computadores entrarem em consenso é bem elevado na maioria dos sistemas, especialmente quando se trata de replicação. Isso ocorre, por exemplo, devido a ordem na qual os pacotes deve ser processados. Neste caso se uma das réplicas executa A e depois B e outra executa B depois A, os dados das duas podem apresentar inconsistências.

Nesse trabalho iremos apresentar uma solução encontrada em estudos recentes sobre a ordenação de pacotes para reduzir o impacto da replicação de sistemas e manter as vantagens envolvidas ao se utilizar sistemas distribuídos. A ordenação dos pacotes antes de serem enviados para o processamento nas réplicas possibilita que o consenso sobre a ordem de processamento não seja necessária na maior parte dos casos, reduzindo consideravelmente o custo computacional.

A ordenação de pacotes também pode ser utilizada, em conjunto com outras tecnologias como a assinatura digital, para tolerar falhas bizantinas. O número de ordenação do pacote pode ser integrado à mensagem e em conjunto com uma chave privada gerarem

uma assinatura digital especifica para aquele pacote naquela ordem, tornando os ataques nesse tipo de arquitetura muito mais difíceis.

Dessa forma, iremos apresentar neste texto, uma forma de utilizar a ordenação para lidar com as falhas por parada e bizantinas sem que haja um grande impacto no sistema, e mantendo as vantagens da replicação de dados.

1.1 Objetivos

Neste trabalho, apresentamos dois novos algoritmos destinados a aprimorar os protocolos de consenso para sistemas distribuídos. O primeiro algoritmo já foi proposto na literatura e é projetado para alcançar consenso entre réplicas sobre a ordem de processamento de mensagens com tolerância a falhas por parada. O segundo algoritmo, proposto neste trabalho, aborda o consenso com tolerância a falhas bizantinas utilizando assinatura digital e ordenação dos pacotes, de uma forma mais eficiente e menos custosa do que os algoritmos de consenso convencionais.

Protocolos de consenso como o Paxos são fundamentais para garantir a consistência dos dados em sistemas distribuídos, mas muitas vezes vêm com o custo de aumento de latência e sobrecarga. Esses protocolos necessitam de coordenação extensiva em cada solicitação, o que impõe uma penalidade significativa de latência e limita a escalabilidade do sistema. Nosso primeiro algoritmo busca mitigar esses custos de desempenho, reduzindo a necessidade de coordenação.

O desafio das falhas bizantinas, onde componentes podem falhar ou fornecer informações incoerentes para o restante do sistema, exige uma solução robusta que garanta a confiabilidade e integridade do sistema sob condições maliciosas. O segundo algoritmo apresentado visa solucionar isso através do emprego de assinatura das mensagens enviadas em conjunto com um contador que só é capaz de ser incrementado, tornando impossível gerar duas assinaturas iguais para mensagens diferentes.

Neste sentido, este trabalho demonstrou que é viável alcançar consenso em sistemas distribuídos assíncronos sem degradação significativa de desempenho. Nossa abordagem aproveita um novo protocolo de replicação que explora a ordenação de rede para minimizar a coordenação, aumentando o *throughput* e diminuindo a latência do sistema em comparação com os algoritmos de consenso existentes. Por meio de validação experimental, planejamos coletar dados relacionados a latência e ao *throughput* tanto do sistema de falhas por parada como o de falhas bizantinas, para no final analisarmos os resultados obtidos.

1.2 Organização do Texto

O texto está organizado em 5 capítulos principais, sendo dois delas a conclusão (capítulo 5) e a introdução (Capítulo 1). As outras três seções estão separadas em fundamentação teórica (Capítulo 2), onde iremos abordar todos os conceitos necessário para o entendimento das ideias propostas e algumas propostas bem conhecidas dentro do âmbito dos sistemas distribuídos.

No Capítulo 3 serão abordados os artigos utilizados como base para a implementação final, bem como o protocolo para tolerar falhas bizantinas. Nela será feito um resumo do conteúdo de cada artigo, de forma a apresentar o conteúdo necessário para o entendimento dos algoritmos utilizados e a construção da aplicação final, bem como alguns pseudo códigos que exemplificam a lógica utilizada no sistema implementado.

No Capítulo 4 apresentamos as características do ambiente em que foram realizados os experimentos e a metodologia utilizada para realizá-los. Além disso, são discutidos e apresentados por meio de gráficos e tabelas os dados coletados em nossos experimentos.

Capítulo 2

Fundamentação Teórica e Trabalhos Relacionados

Neste capítulo, abordaremos conceitos teóricos fundamentais e estudos relevantes em sistemas distribuídos tendo como objetivo nortear toda teoria das tecnologias adjacentes a proposta que serão abordados nesse trabalho. Teremos um grande foco em replicação e consenso, exemplificaremos os principais algoritmos de consenso, destacando os diferentes tipos existentes e suas aplicações práticas.

2.1 Sistemas Distribuídos

Um sistema distribuído consiste em uma coleção de computadores ou processos autônomos e interconectados, sejam elas hardware ou software, que colaboram visando atingir um objetivo comum [3]. A essência dessa colaboração se dá através do intercâmbio de mensagens, permitindo que processos em diferentes locais se comuniquem e coordenem suas ações. Uma característica distintiva desses sistemas é a inexistência de um relógio global unificado, implicando que a sincronização de eventos é alcançada por meio de protocolos de comunicação específicos, em vez de uma referência temporal comum.

Independência de falhas também é um aspecto intrínseco aos sistemas distribuídos. Isso significa que componentes individuais podem experimentar falhas sem necessariamente comprometer a integridade ou disponibilidade do sistema como um todo. Esta propriedade realça a necessidade de mecanismos robustos de tolerância a falhas, que asseguram a continuidade do serviço mesmo na presença de componentes defeituosos.

Ademais, sistemas distribuídos não estão restritos por limitações geográficas. Eles podem operar independente da localização física de seus componentes, desde que exista uma infraestrutura de rede adequada para realizar a comunicação. Esta característica

viabiliza uma ampla gama de aplicações, desde sistemas de processamento de transações financeiras globais até redes de sensores para monitoramento ambiental.

Contudo, a natureza descentralizada e a ausência de um relógio global acarretam desafios significativos, particularmente relacionados à concorrência e à coordenação entre processos. Problemas como ordem de execução, consistência de dados e detecção de falhas exigem soluções sofisticadas que geralmente envolvem algoritmos distribuídos complexos e mecanismos de consenso [4].

2.1.1 Concorrência

A concorrência, no contexto de sistemas distribuídos, refere-se à capacidade de executar múltiplas operações ou tarefas independentes, de maneira que a ordem de execução não prejudique o resultado final. Esta característica é fundamental para a eficiência e escalabilidade dos sistemas distribuídos, pois permite o processamento paralelo de dados, otimizando o uso dos recursos disponíveis e reduzindo o tempo de resposta.

Em um sistema distribuído, a concorrência manifesta-se através da execução independente de tarefas distribuídas por diversos componentes do sistema, cada um potencialmente localizado em diferentes nós da rede. Esta abordagem divide o problema principal em subproblemas menores, que podem ser resolvidos em paralelo, acelerando o processamento geral.

Entretanto, gerenciar a concorrência em tais sistemas implica enfrentar desafios relacionados à sincronização, coordenação de tarefas e alocação de recursos. Um dos principais problemas é a contenção por recursos compartilhados, que pode levar a estados de *deadlock*, onde múltiplas tarefas aguardam indefinidamente por recursos alocados por outras. Para mitigar esses problemas, estratégias eficazes de gerenciamento de recursos e algoritmos de sincronização são cruciais.

A alocação de recursos em sistemas distribuídos com alta concorrência pode ser abordada de diversas maneiras. Incrementar os recursos do sistema é uma solução direta, mas nem sempre viável ou custo-efetiva. Alternativamente, técnicas de escalonamento e balanceamento de carga podem ser empregadas para otimizar a distribuição de tarefas entre os recursos disponíveis. Estas técnicas buscam maximizar a utilização dos recursos, evitando sobrecargas em determinados nós e subutilização em outros.

Além disso, a coordenação eficaz entre os componentes do sistema é fundamental para o gerenciamento da concorrência. Protocolos de comunicação e algoritmos de consenso são empregados para garantir que todas as tarefas sejam executadas de forma coerente e que os resultados sejam corretamente integrados, preservando a consistência dos dados.

2.1.2 Ausência de Relógio Global

Em sistemas distribuídos, a coordenação efetiva entre componentes dispersos geograficamente impõe desafios significativos, dentre eles, a ausência de um relógio global sincronizado. Devido às limitações físicas e à variação inerente na latência de comunicação, sincronizar relógios em diferentes nós de uma rede de forma precisa é uma tarefa praticamente inviável. Além disso, a relatividade do tempo e as discrepâncias entre os relógios internos dos dispositivos tornam a ideia de um "tempo global correto" um conceito impraticável para operações cotidianas em tais sistemas.

Esta ausência de sincronia temporal absoluta leva à necessidade de mecanismos alternativos para a coordenação de tarefas e a garantia de consistência de dados, sem depender diretamente da noção de tempo. Nesse contexto, a comunicação baseada em mensagens emerge como a estratégia fundamental para o gerenciamento da ordem e do estado das operações distribuídas.

Além disso, sistemas de rótulos de tempo permitem que cada componente mantenha um registro do conhecimento que tem sobre o "tempo" nos outros componentes, facilitando a identificação de dependências causais entre eventos distribuídos. Essas técnicas permitem que sistemas distribuídos alcancem um grau de coordenação eficaz, assegurando a consistência de estados e a ordenação adequada de operações, apesar da falta de um relógio global.

2.1.3 Falhas Independentes

Os sistemas distribuídos, pela sua própria natureza, estão sujeitos a um leque diversificado de falhas que não se limitam aos problemas comuns observados em sistemas centralizados. Devido à distribuição geográfica dos componentes e à complexidade da rede de comunicação que os interliga, esses sistemas enfrentam desafios únicos no que diz respeito à detecção e manejo de falhas.

Falhas de rede são exemplos proeminentes, capazes de afetar severamente a capacidade de comunicação entre os nós do sistema. Interrupções ou latência elevada na rede podem resultar no atraso ou perda de mensagens, gerando estados inconsistentes entre os componentes. Apesar de os dispositivos envolvidos permanecerem operacionais, a incapacidade de se comunicarem eficazmente compromete a funcionalidade global do sistema.

Além disso, falhas de software e hardware em nós individuais introduzem desafios adicionais. Essas falhas podem variar desde erros de programação, que causam comportamentos inesperados da aplicação, até problemas físicos nos dispositivos, que levam à perda de serviço. A heterogeneidade dos sistemas distribuídos amplifica esses problemas,

uma vez que diferentes plataformas e tecnologias podem reagir de maneira distinta sob condições adversas.

A detecção oportuna dessas falhas é complicada pela distribuição dos componentes e pela falta de um estado global coerente. Mecanismos tradicionais de monitoramento e detecção de falhas podem não ser suficientemente rápidos ou precisos para identificar e isolar problemas antes que eles afetem a operacionalidade do sistema. Como consequência, falhas em partes específicas do sistema podem permanecer ocultas por períodos prolongados, exacerbando o impacto no desempenho e na disponibilidade.

Para enfrentar esses desafios, sistemas distribuídos adotam estratégias de tolerância a falhas, como replicação de dados, balanceamento de carga e reconexão automática. A replicação de dados entre diferentes nós garante que, mesmo na ocorrência de falhas de hardware ou de rede, cópias consistentes dos dados permaneçam acessíveis. O balanceamento de carga distribui as solicitações de maneira uniforme entre os nós, evitando sobrecargas que possam levar a falhas de desempenho. Mecanismos de reconexão automática e retransmissão de mensagens também ajudam a mitigar os efeitos de interrupções temporárias na rede, assegurando a continuidade da comunicação.

2.2 replicação e Consenso

Um dos maiores desafios nos sistemas distribuídos é a replicação da informação. A informação geralmente é replicada para melhorar a confiabilidade ou melhorar o desempenho, o desafio se dá ao tentar manter a consistência entre as réplicas. Informalmente isso quer dizer que quando uma réplica é atualizada é preciso que todas as outras cópias também sejam atualizadas, ou seja as réplicas precisam entrar em um consenso sobre qual informação deve ser a mais atualizada. Manter a consistência acaba se tornando uma tarefa difícil de ser implementada de maneira eficiente, ou seja, de forma escalável [5]. Nessa perspectiva, existem dois motivos para se querer utilizar da replicação.

- **Performance Aprimorada:** A replicação de dados desempenha um papel crucial na melhoria do desempenho em sistemas distribuídos, especialmente quando é necessário escalar o sistema em termos de tamanho ou área geográfica. Quando há um aumento no número de processos que precisam acessar dados gerenciados por um único servidor, a performance pode ser significativamente aprimorada ao replicar o servidor e dividir a carga de trabalho entre os processos que acessam os dados. Além disso, a replicação de dados em caches locais reduz consideravelmente a latência de acesso, já que as informações estão mais próximas dos pontos de uso. A distribuição de carga, utilizando algoritmos como o *round-robin*, otimiza o uso dos recursos e evita a sobrecarga de servidores individuais. No caso de escalonamento

geográfico, a replicação também se torna essencial. Colocar uma cópia dos dados próxima ao processo que os utiliza diminui o tempo de acesso, resultando em uma melhoria de desempenho e percebida por esse processo. No entanto, essa abordagem pode aumentar o consumo de largura de banda da rede para manter todas as réplicas atualizadas, o que deve ser considerado na avaliação do impacto da replicação no desempenho geral do sistema.

- **Tolerância a Falhas:** A replicação de dados em sistemas distribuídos é uma técnica essencial para aumentar a confiabilidade. Ao manter múltiplas réplicas de um dado, o sistema pode continuar operando mesmo que uma das réplicas falhe ou trave, simplesmente redirecionando as operações para outra réplica disponível. Essa redundância não apenas garante a continuidade do serviço, mas também protege contra a corrupção de dados, já que várias cópias são mantidas e sincronizadas regularmente.

  Outro aspecto importante é a consulta entre réplicas, que permite verificar a vigência e a integridade das informações armazenadas. Esse processo, muitas vezes apoiado por algoritmos de consenso, é fundamental para identificar e resolver inconsistências, garantindo que a versão mais confiável e atual dos dados seja utilizada pelo sistema. Assim, a réplicação não só aumenta a resiliência do sistema como também assegura a integridade e consistência dos dados em ambientes distribuídos.

Para a implementação da replicação porém existem algumas dificuldades a serem levadas em consideração, em se tratando da escalabilidade do problema, uma vez que para cada nova réplica adicional é necessário garantir que todas as réplicas sejam iguais ou seja garantir a consistência da informação. Isso significa que uma operação de leitura feita em qualquer réplica deve retornar sempre o mesmo valor, e uma operação de escrita feita em uma das réplicas deve ser propagada para todas as demais réplicas antes que operações subsequentes de leitura sejam feitas.

Esse tipo de problema pode ser resolvido com a atomicidade das transações, porém esse processo se torna mais lento conforme se aumentam os números de réplicas, tornando a solução não escalável para processos que exijam velocidade na resposta. Uma solução para isso seria a criação de um sistema de cache, porém voltaríamos ao problema da sincronização de dados. Uma solução para esse problema seria a implementação de um sistema de votação das réplicas. Dessa forma, considerando um servidor com N replicações, para que uma operação fosse realizada nesse servidor seriam necessários que uma quantidade (por exemplo, a metade) das réplicas aprovarem a operação, que então é realizada e as réplicas que realizarem a operação alteram a versão em que estão para a nova versão na

qual a operação foi realizada. Para denotar qual versão prevalece entre as demais poderia se usar o mesmo sistema para adquirir o consenso entre as réplicas.

O consenso é crucial para a coordenação e a consistência entre múltiplos processos ou nodos. Este envolve alcançar um acordo unânime sobre um determinado valor ou decisão entre todos os componentes envolvidos, mesmo diante de falhas e da falta de sincronização no sistema. O consenso é alcançado quando todos os participantes concordam com um determinado valor ou sequência de ações, sendo uma necessidade fundamental em sistemas que replicam dados para aumentar a disponibilidade e a tolerância a falhas. Este problema é particularmente desafiador quando consideramos a possibilidade de falhas arbitrárias (falhas bizantinas) ou a perda de mensagens na rede.

Existem diversas estratégias para alcançar o consenso em sistemas distribuídos, as quais podem ser categorizadas com base no modelo do sistema (síncrono ou assíncrono) e no tipo de falhas consideradas (crash ou bizantinas). Em sistemas síncronos, os processos operam em rodadas e têm um conhecimento prévio sobre o tempo máximo de resposta. Para sistemas assíncronos, onde não há garantias sobre o tempo de resposta, alcançar o consenso se torna mais complexo.

2.3 Algoritmos de Consenso

Algoritmos de consenso desempenham um papel fundamental em sistemas distribuídos, assegurando a coordenação e a consistência entre os processos em um ambiente distribuído. Para que um algoritmo de consenso seja considerado correto, ele deve satisfazer três condições críticas:

- **Terminação:** Eventualmente, cada processo correto deve chegar a uma decisão.
- **Acordo:** Todos os processos corretos devem concordar com o mesmo valor de decisão.
- **Integridade:** Se todos os processos corretos recebem o mesmo valor de entrada, então todos eles devem decidir sobre o mesmo valor.

A integridade, em particular, pode ter sua interpretação ajustada conforme o contexto; por exemplo, pode-se aceitar a decisão com base no valor proposto por qualquer processo correto, não necessariamente exigindo unanimidade na entrada. No contexto dos Generais Bizantinos, a integridade especifica que, se o comandante (processo iniciador) está correto, então todos os tenentes (outros processos) devem concordar com o valor proposto pelo comandante.

2.3.1 Consenso em Sistemas Síncronos

Em sistemas síncronos, onde existe uma suposição de limites conhecidos para o tempo de transmissão de mensagens e processamento, é mais simples projetar algoritmos de consenso que tolerem falhas. Uma abordagem comum é requerer que cada processo colete valores dos outros processos.

Esse processo de coleta é realizado em rodadas de comunicação. Durante essas rodadas, os processos realizam uma *multicast* dos seus valores atuais para os outros, permitindo que, mesmo na presença de falhas, o consenso seja alcançado entre os processos remanescentes.

Um aspecto crítico em sistemas síncronos é a capacidade de avançar para a próxima rodada apenas quando um processo recebeu suficientes mensagens para garantir que um acordo possa ser alcançado, considerando o número máximo de falhas permitidas. Isso assegura que, apesar das falhas, os processos corretos restantes possam chegar a um acordo sobre um valor de decisão comum, cumprindo assim as condições de terminação e acordo.

2.3.2 Consenso em Sistemas Assíncronos

Em sistemas assíncronos, diferentemente dos sistemas síncronos, não podemos fazer suposições confiáveis sobre o tempo de transmissão de mensagens ou sobre o tempo de processamento nos nodos. Uma consequência direta dessa incerteza é a dificuldade em distinguir entre um processo que falhou e um que está simplesmente respondendo lentamente, já que um processo pode levar um tempo arbitrário para responder.

Essa característica dos sistemas assíncronos leva a um resultado teórico significativo conhecido como o *Teorema da Impossibilidade* de FLP (Fischer, Lynch, e Paterson) [6] que demonstra ser impossível, em tais sistemas, garantir consenso de forma determinista se pelo menos um processo pode falhar. Este teorema implica que não existe um algoritmo determinístico de consenso que sempre termine com sucesso em um ambiente puramente assíncrono e sujeito a falhas.

Apesar deste cenário desafiador, pesquisas subsequentes introduziram o conceito de sistemas *parcialmente síncronos*, nos quais se assume que o sistema pode comportar-se de maneira assíncrona durante certos períodos, mas eventualmente apresentará propriedades síncronas. Em outras palavras, existe um limite (desconhecido a priori) após o qual o sistema garante tempos de resposta e transmissão de mensagens dentro de um espectro conhecido e previsível.

A abordagem parcialmente síncrona oferece um caminho pragmático para alcançar consenso em sistemas distribuídos reais, onde condições de rede e de processamento podem variar, mas geralmente se estabilizam dentro de limites razoáveis. Algoritmos como Paxos e Raft, bem como variantes bizantinas, como Practical Byzantine Fault Tolerance

(PBFT), foram desenvolvidos para operar sob essas premissas parcialmente síncronas, proporcionando uma base sólida para a construção de sistemas distribuídos confiáveis que são capazes de alcançar consenso mesmo na presença de falhas.

2.3.3 Algoritmo Paxos

Devido à importância deste algoritmo para nosso trabalho, ele será explicado de forma aprofundada nesta seção para facilitar o entendimento dos detalhes técnicos envolvidos na implementação realizada.

Histórico e Contexto

O algoritmo de Paxos foi proposto por Leslie Lamport em 1989 e formalmente publicado em 1998 [7]. Inspirado em um sistema legislativo fictício da ilha grega de Paxos, o algoritmo foi desenvolvido para garantir consenso mesmo na presença de falhas, tornando-se uma base para numerosas implementações em sistemas de armazenamento de dados distribuídos.

Definições Importantes

Antes de detalhar o algoritmo de Paxos, é crucial entender os três tipos de agentes envolvidos:

Propositor Sugerem valores para acordo.

Aceitadores Decidem sobre a aceitação dos valores propostos.

Aprendizes Aprendem o valor decidido pelos Aceitadores.

Também é necessário entender que o algoritmo assume algumas características para os processadores:

- Processadores podem falhar, mas são considerados não maliciosos.
- A comunicação é confiável, ainda que possa ocorrer atrasos.
- Processadores operam em velocidades variadas.
- Não existe comportamento malicioso, evitando mentiras ou decepções.

A rede deve satisfazer as condições de:

- Permitir comunicação entre quaisquer dois processadores.
- Operar de maneira assíncrona, com potenciais atrasos indefinidos.

- Mensagens podem ser perdidas, duplicadas ou reordenadas, mas não corrompidas.

Paxos também possui três propriedades essenciais: validade, consistência e terminação, assegurando que valores propostos sejam escolhidos de forma justa e única, e que todos os Aprendizes eventualmente aprendam um valor proposto.

Fluxo do Algoritmo

O funcionamento do algoritmo de Paxos é bem simples sendo operado em duas fases.

Fase 1: Preparação

1. **Preparar**: O Propositor envia uma solicitação de preparação aos aceitadores com um número de proposta único.
2. **Promessa**: Aceitadores comprometem-se a não aceitar propostas com números inferiores.

Fase 2: Proposição

1. **Propor**: Após receber a maioria das respostas, o propositor envia uma proposta com um valor específico.
2. **Aceitação**: Aceitadores aceitam a proposta se ela corresponder ao maior número de proposta ao qual se comprometeram.

Quando a maioria dos aceitadores aceita uma proposta, o valor é considerado escolhido, sendo então notificado aos aprendizes como o valor de consenso.

2.4 Falhas Bizantinas e o Problema dos Generais Bizantinos

O Problema dos Generais Bizantinos, introduzido por Lamport, Shostak e Pease em 1982 [8], é uma das questões mais fundamentais na área de sistemas distribuídos. O problema descreve uma situação em que vários componentes de um sistema distribuído precisam chegar a um consenso, mesmo quando alguns desses componentes podem falhar de maneira arbitrária ou agir de forma maliciosa. Essas falhas, conhecidas como falhas bizantinas, são especialmente desafiadoras porque podem fazer com que diferentes partes do sistema tomem decisões inconsistentes, comprometendo a integridade e a confiabilidade do sistema como um todo.

O cenário clássico do Problema dos Generais Bizantinos envolve um grupo de generais de um exército que cercam uma cidade inimiga. Esses generais precisam decidir de forma

unânime se devem atacar ou recuar. Entretanto, alguns dos generais podem ser traidores, tentando causar discordância entre os demais. O desafio é que, mesmo na presença desses generais desleais, os leais precisam garantir que todos concordem com a mesma decisão para evitar um desastre militar.

Aplicado a sistemas distribuídos, o problema ilustra a complexidade de se alcançar consenso em redes onde pode haver falhas de comunicação, mensagens atrasadas ou componentes defeituosos que enviam informações contraditórias. Quando falhas bizantinas estão presentes, a solução se torna significativamente mais complexa. Algoritmos tradicionais como o Paxos são eficazes para lidar com falhas de parada, mas não são adequados para falhas bizantinas, onde um nó pode se comportar de maneira arbitrária. Para lidar com esse tipo de falha, é necessário implementar algoritmos especificamente projetados para tolerância a falhas bizantinas (Byzantine Fault Tolerance – BFT).

Os algoritmos para falhas Bizantinas requerem pelo menos $3F + 1$ componentes para tolerar a falha de até F deles. Isso assegura que, mesmo com F componentes falhando, os componentes restantes ainda possam formar uma maioria suficiente para chegar a um acordo.

2.5 Assinaturas digitais

A escrita a mão é uma forma comum de comunicação a séculos, dela se derivam as assinaturas que estão entre as principais formas de identificar uma pessoa nos dias de hoje, devido ao comportamento e características únicas de cada individuo [9]. Dessa forma, com a migração crescente do mundo físico para o digital, tornou-se necessário a criação de uma forma de assinar digitalmente confirmando a identidade do autor da mensagem recebida.

Para que uma assinatura digital corresponda ao uso da assinatura física existem duas primitivas que devem ocorrer, a primeira é de que somente você consiga utilizar sua assinatura, e a outra, é assegurar que qualquer um pode confirmar a veracidade da assinatura pertencente a você. Para assegurar isso a assinatura digital utiliza 3 algoritmos. Na Figura 2.1 temos a representação visual do funcionamento desses 3 algoritmos [1].

1. **Geração de par de chaves** utilizado para criar um conjunto de chave privada e pública.
2. **Assinatura** algoritmo que utiliza da chave privada e da mensagem para gerar uma assinatura.
3. **Verificação** algoritmo que a partir da chave pública da mensagem e da assinatura retorna se a assinatura é verdadeira para aquela mensagem ou não.

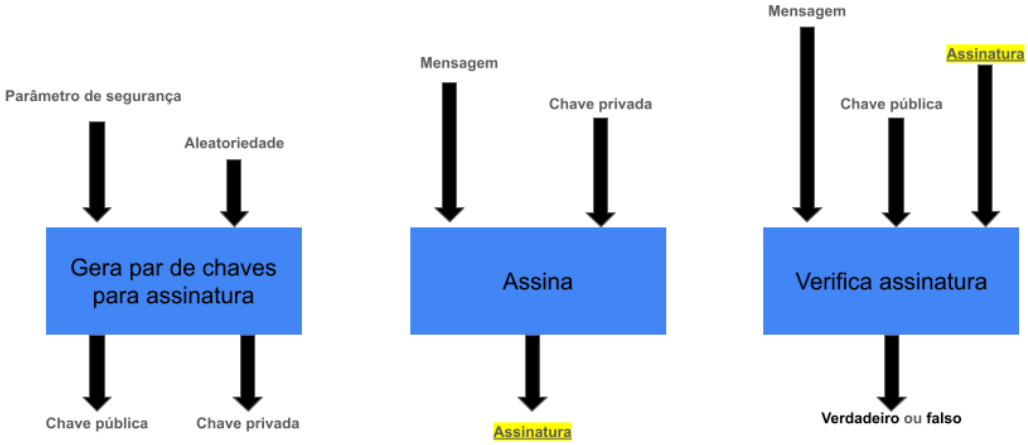

Figura 2.1: Explicação dos algoritmos para assinatura digital. FONTE adaptado de Wong, David - Real-World Criptography [1]

Ao combinar esse conjunto de algoritmos se dá à assinatura digital duas garantias básicas: a de **origem**, que garante que a mensagem vem de quem a assinou, e a de **integridade**, que assegura que, se a mensagem for modificada, a assinatura será inválida para a nova mensagem e precisará passar pelo processo de assinatura novamente. Assim, torna-se possível que dois sistemas que previamente não tinham a garantia de quem enviou a mensagem, agora consigam averiguar o remetente da mensagem, e, caso a verificação falhe, a mensagem pode ser descartada e considerada fraudulenta.

2.6 Conclusões do Capítulo

Este capítulo explorou os fundamentos e os desafios associados aos sistemas distribuídos, com um foco particular em algoritmos de consenso. Através dessa análise, fica evidente que, embora os sistemas distribuídos ofereçam inúmeras vantagens, como melhor desempenho, escalabilidade e tolerância a falhas, eles também apresentam complexidades significativas que precisam ser cuidadosamente gerenciadas.

Um dos maiores desafios identificados é a questão da determinação do estado de outros processos em sistemas assíncronos. A incerteza se um processo falhou ou está apenas respondendo lentamente destaca a dificuldade em alcançar um consenso confiável em tais sistemas. Este desafio é agravado em cenários onde falhas arbitrárias ou maliciosas, como

ilustrado pelo problema dos Generais Bizantinos, introduzem complexidades adicionais na garantia de acordo mútuo entre as partes.

A discussão enfatizou a importância dos algoritmos de consenso em assegurar que, apesar desses desafios, os sistemas distribuídos possam funcionar de maneira confiável e eficiente. A necessidade de um consenso robusto é particularmente crítica em aplicações onde a consistência e a integridade dos dados são fundamentais, como em sistemas financeiros, redes de comunicação críticas.

Para enfrentar esses desafios, foram exploradas soluções como a introdução de sistemas parcialmente síncronos, que oferecem um compromisso pragmático entre os sistemas puramente assíncronos e os síncronos. Algoritmos como Paxos, Raft e PBFT demonstram como, sob certas condições, é possível alcançar consenso mesmo na presença de falhas e incertezas temporais.

Em conclusão, a análise dos algoritmos de consenso em sistemas distribuídos revela um campo de estudo fascinante e desafiador, onde a busca por soluções robustas continua a evoluir. As lições aprendidas e as soluções desenvolvidas para superar esses desafios não apenas permitem a construção de sistemas distribuídos mais confiáveis e eficientes, mas também contribuem para o avanço teórico e prático na área de ciência da computação.

Capítulo 3

Separando Ordenação de Terminação

Neste capítulo, apresentaremos um nova abordagem na replicação de sistemas distribuídos que visa superar algumas das limitações dos protocolos tradicionais de consenso e coordenação, como o Paxos. Esses métodos convencionais, embora eficazes em garantir a consistência das réplicas, frequentemente requerem uma coordenação intensiva que pode comprometer o desempenho geral do sistema aumentando a latência e o uso de recursos dificultando a escalabilidade. Diante da necessidade crítica de manter dados replicados consistentes em ambientes sujeitos a falhas de hardware ou de rede, o artigo [10] propõe o algoritmo NOPaxos (*Network-Ordered Paxos*), um protocolo que simplifica a coordenação entre réplicas utilizando um Multicast Ordenado e Não Confiável (*Ordered Unreliable Multicast* – OUM).

O NOPaxos se destaca por reduzir a sobrecarga associada ao processo de consenso ao depender primariamente da infraestrutura de rede para manter a ordem das mensagens, eliminando a necessidade de coordenação contínua entre as réplicas. Este capítulo detalha tanto a estrutura quanto o funcionamento do NOPaxos, explicando como a inclusão de um sequenciador de rede pode permitir uma replicação consistente com uma latência significativamente reduzida em comparação aos métodos tradicionais.

Além disso, expandimos com outra proposta visando incluir o tratamento de falhas bizantinas, integrando abordagens mencionadas no artigo [11], em especial o serviço USIG, que aumenta a robustez do sistema contra ataques maliciosos ou falhas arbitrárias bizantinas que podem comprometer a ordem ou integridade das mensagens sem a necessidade do aumento no número de réplicas como nos algoritmos para tolerância de falhas bizantinas convencionais.

3.1 Algoritmo para Falhas por Parada

Um dos desafios críticos em sistemas distribuídos é a gestão de falhas por parada, comum em tais ambientes devido à inevitabilidade de falhas de hardware ou de rede. A replicação é amplamente utilizada como estratégia para mitigar perdas causadas por essas falhas, garantindo alta disponibilidade e consistência dos dados. Tradicionalmente, protocolos como Paxos são empregados para assegurar a consistência das réplicas, no entanto, eles exigem coordenação intensiva e podem impactar negativamente o desempenho do sistema [10].

Introduzindo uma inovação nesse contexto, o protocolo NOPaxos apresenta uma abordagem alternativa, reduzindo a sobrecarga do sistema associada ao Paxos. O NOPaxos elimina a necessidade de coordenação contínua entre as réplicas, dependendo apenas da rede para entregar pacotes em uma ordem consistente. Isso é realizado por meio de um sequenciador conectado na rede que ordena as solicitações recebidas antes de distribuí-las às réplicas.

O NOPaxos aproveita a ordenação provida pela rede para alcançar replicação consistente sem coordenação. A rede deve garantir que os pacotes sejam entregues na mesma ordem a todas as réplicas, por meio do sequenciador que marca cada pacote com um número de sequência antes da entrega. Isso reduz drasticamente os problemas de desempenho enfrentados por outros protocolos de consenso, alcançando uma latência mínima. Em casos onde a rede falha em entregar uma mensagem, o sistema recorre a mecanismos tradicionais do Paxos para solicitar a mensagem perdida. Entretanto, se a rede fosse totalmente confiável e funcionar sem interrupções, o uso do protocolo Paxos poderia ser completamente eliminado.

O NOPaxos oferece uma performance notavelmente próxima à de sistemas não replicados, proporcionando uma latência e throughput quase idênticas aos de um sistema sem replicação. Este avanço é especialmente significativo, pois demonstra que a replicação de dados não precisa inerentemente vir acompanhada de um alto custo de desempenho. Adicionalmente, o protocolo é robusto a falhas de rede, mantendo um alto throughput mesmo em condições de rede desfavoráveis, graças ao modelo de OUM que garante uma ordem aos pacotes, mas não que os pacotes sejam entregues, simplificando a complexidade do protocolo de replicação enquanto mantém a consistência dos dados entre as réplicas.

3.1.1 Sequenciador

O sequenciador desempenha um papel crucial na arquitetura do *Multicast* Ordenado e Não Confiável, sendo responsável por atribuir um número sequencial a cada mensagem destinada a um grupo OUM. Um mesmo sequenciador pode administrar vários multicasts

na mesma rede, necessitando assim de um número de ordem para cada novo grupo que administrar. Cada mensagem que chega a um grupo passa pelo sequenciador que registra no cabeçalho da mensagem um número a ser considerado a ordem de processamento das mensagens, eliminando a necessidade de consenso sobre a ordem de processamento das mensagens [10].

Essa alternativa se torna superior ao uso de marcadores de tempo (*timestamps*), pois apesar de serem monotônicos os *timestamps* apresentem intervalos arbitrários entre cada um, tornando impossível detectar se houve uma falha ou se o tempo entre mensagens apenas aumentou, e mesmo assim não é possível afirmar que não houve perda de pacotes, apenas que nenhuma réplica recebeu algum pacote no intervalo entre outros dois. Com pacotes ordenados se torna possível confirmar se a mensagem recebida é a esperada sem a necessidade de consultar às demais réplicas, uma vez que caso haja uma lacuna na sequencia das mensagens pode-se garantir que houve uma perda de pacote.

É notável a eficácia do sequenciador, apesar de sua simplicidade. Podendo processar até 100 milhões de requisições por segundo [10], demonstrando uma capacidade significativa, apesar da escalabilidade limitada. Nossa implementação do sequenciador é restrita a um único contador e, consequentemente, a um único grupo OUM, para simplificar o sistema e focar na robustez contra falhas bizantinas.

Existem três formas de se implementar um sequenciador. A primeira por meio de Switchs, reduzindo bastante a latência causada pelo redirecionamento dos pacotes a um novo dispositivo responsável pela ordem das mensagens. Porém switchs capazes dessa implementação ainda são muito novos e de difícil acesso. A segunda alternativa é implementar um protótipo dos switches usando o hardware disponível no mercado, porém a latência já aumenta devido a necessidade de redirecionar os pacotes de rede ao novo dispositivo, a desvantagem sendo a necessidade de hardware especializado e diminuindo a flexibilidade. Por fim temos a opção que utilizamos, a implementação utilizando sofware e uma computador que irá receber os dados antes de repassa-los, que possibilita uma melhor abstração e flexibilidade, algo que se torna fundamental no nosso trabalho devido as alterações feitas no projeto para a inclusão de tolerância a falhas bizantinas.

Na Figura 3.1 temos o funcionamento do sequenciador para um único grupo OUM. Dois clientes A e B enviam cada um uma mensagem, que será recebida para o sequenciador e transmitida para as réplicas com um número atrelado as mensagens, dessa forma nenhuma réplica necessita consultar para determinar a ordem de processamento das mensagens de cada cliente.

Para manter a tolerância a falhas por parada é essencial que existam mais de um sequenciador, uma vez que na falha de um o outro o substituirá. Isso pode ser feito por meio de um controlador de rede que monitora a disponibilidade do sequenciador e em

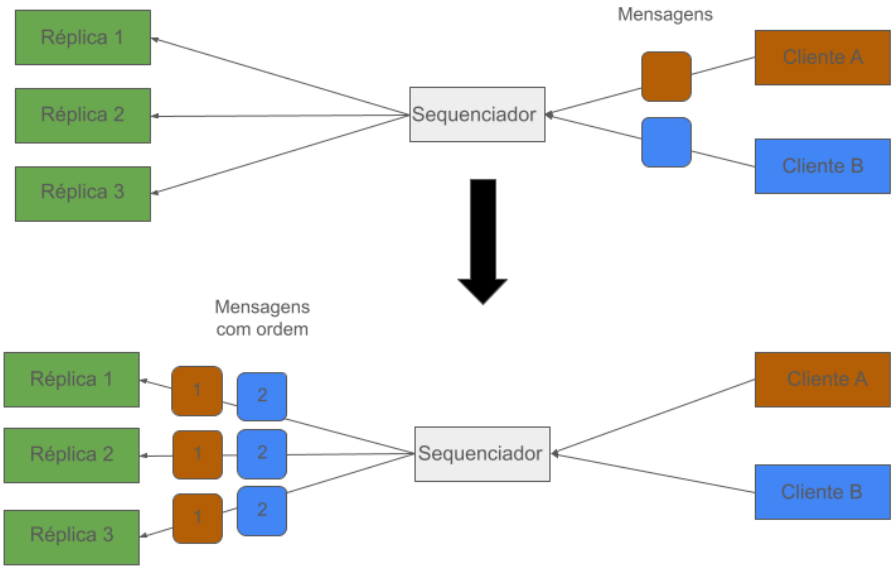

Figura 3.1: Fluxo do sequenciador

caso de falha instancia um novo sequenciador que continua o trabalho normalmente, isso porém apresenta uma falha na continuidade da ordem dos pacotes, após a substituição o novo sequenciador deve continuar a transmitir os dados exatamente de onde o último parou.

Para contornar esse problema podemos atribuir a um sequenciador um número de sessão, para cada sequenciador que falha o número de sessão é incrementado, gerando assim uma tupla que representa a ordem total das mensagens e as réplicas continuam podendo descartar pacotes vindos fora de ordem [10].

3.1.2 O algoritmo NOPaxos

O NOPaxos é um protocolo de replicação que opera sob a premissa de que os pacotes são recebidos de forma ordenada, sem necessariamente exigir a entrega confiável das mensagens. A essência do algoritmo é garantir que todas as mensagens sejam processadas em uma ordem global, mesmo que algumas sejam perdidas, enfrentem problemas durante a entrega ou que ocorram falhas por parada no sistema.

O NOPaxos é construído a partir das garantias dadas pela primitiva de rede OUM. Nesse processo os pacotes tem total garantia de serem ordenados porém podem não ser entregues. Dessa forma as réplicas só precisam entrar em consenso em quais pacotes devem ignorar e quais pacotes devem processar. Isto torna a tarefa de entrar em consenso

muito mais simples, uma vez que só é preciso haver uma comunicação entre as réplicas no caso em que há a perda de pacotes e não a cada pacote recebido.

No momento em que uma réplica detecta uma lacuna na sequencia de pacotes recebidos, sua primeira ação é requisitar às demais réplicas os pacotes faltantes a serem processados, caso isso também falhe as réplicas então precisam entrar em consenso sobre quais pacotes devem ser ignorados para o processamento continuar. Isso pode ser feito elegendo uma réplica líder responsável por realizar essa votação.

Em um processo normal os clientes fazem sua requisição, então passa pelo sequenciador que adiciona a seu cabeçalho o valor atual de seu contador interno e transmite esse novo pacote às réplicas, ao receber esse pacote as réplicas o processam e retornam ao cliente a resposta encontrada. O cliente então espera por $f + 1$ respostas iguais, considerando um sistema com $2f + 1$ réplicas. Caso o cliente não recebe as respostas em um tempo limite, ele tenta refazer a requisição.

3.2 Algoritmo para Falhas Bizantinas

A evolução dos algoritmos de tolerância a falhas bizantinas busca abordagens que reduzam o número de réplicas necessárias para alcançar consenso na presença de componentes maliciosos. Tradicionalmente, tais algoritmos requerem $3f + 1$ réplicas para tolerar até f falhas bizantinas. No entanto, exitem métodos que funcionam eficientemente com apenas $2f + 1$ réplicas . Nosso trabalho atual se inspira nesses desenvolvimentos para aprimorar o algoritmo NOPaxos, integrando o serviço Gerador de Identificador Sequencial Único (Unique Sequential Identifier Generator – USIG) como mecanismo central para garantir a integridade das comunicações dentro do sistema.

O algoritmo proposto utiliza um serviço de assinatura confiável que não apenas assina as mensagens recebidas, mas também as numera de forma sequencial e inalterável, dificultando a execução de ataques que tentem repetir ou alterar a ordem das mensagens.

3.2.1 Fundamentos do Serviço USIG

O USIG é um componente crucial em nosso sistema para a tolerância a falhas bizantinas, oferecendo uma robustez adicional contra ataques que tentam explorar a ordem ou a integridade das mensagens. Este serviço garante que cada mensagem dentro do sistema seja marcada com um identificador único e sequencial, vital para prevenir e mitigar problemas como ataques de repetição ou a inserção de comandos falsos que poderiam desordenar ou invalidar o processo de consenso [11].

A essência do USIG reside na sua capacidade de gerar esses identificadores de forma que sejam tanto únicos quanto monotônicos, isto é, sempre incrementais. Cada identifica-

dor é atribuído de maneira sequencial e estritamente crescente, garantindo que qualquer tentativa de reordenar as mensagens seja facilmente detectável pelas réplicas do sistema.

Inviolabilidade e Segurança:

Para assegurar a integridade e a inviolabilidade deste serviço, o USIG deve ser hospedado em um componente à prova de falhas ou ataques. Idealmente, este componente é um módulo de hardware dedicado chamado TPM (*Trusted Platform Module*). Um dispositivo projetado para executar operações criptográficas de maneira segura, isolada do restante do sistema, e capaz de resistir a uma ampla gama de ataques físicos e lógicos. TPMs tem a habilidade de assinar mensagens utilizando de uma chave primária que não tem contato com o ambiente externo.

A evolução de algoritmos bizantinos para necessitar de apenas $2f + 1$ réplicas em vez de $3f + 1$ réplicas vem acontecendo há alguns anos [12], porém modelos anteriores tinham algumas dificuldades técnicas na implementação ou obtenção de hardware. Porém a utilização do USIG traz a possibilidade de utilizar apenas $2f + 1$ réplicas sem que haja a necessidade de alteração do algoritmo. Isso se deve ao fato das réplicas serem obrigadas a trabalhar em uma sequência incremental monotônica de operações, e como a sequencia é facilmente verificável não é possível que réplicas defeituosas apresentem sequências diferentes para réplicas diferentes, resultando assim na necessidade de apenas $2f + 1$ réplicas para garantir até f falhas.

3.2.2 Implementação e Opções de Segurança

O algoritmo pode ser implementado utilizando várias abordagens dependendo do ambiente operacional e dos requisitos de segurança específicos:

- **Módulo de Plataforma Confiável (TPM):** O uso de TPMs para a implementação do USIG oferece a maior segurança, pois esses dispositivos são projetados para realizar operações criptográficas de forma segura e isolada do resto do sistema. Nessa versão o USIG é implementado utilizando uma camada muito fina de software, sendo apenas uma função em um biblioteca, e pelo TPM.
- **Máquinas Virtuais:** Implementar o USIG em uma máquina virtual diferente da máquina em que o processo principal está sendo executado proporciona uma camada de isolamento, protegendo as funções críticas do algoritmo de falhas ou ataques que afetem o sistema operacional principal. O processo principal é executado na réplica na *VM1* que está conectada à rede e tem todo os softwares não confiáveis, e o USIG é executado na *VM0* que não está conectada à rede e possui o mínimo de programas possível. A comunicação entre o USIG e a réplica pode ser feito utilizando-se *sockets*.

- **Implementação por Software:** Uma abordagem menos segura, mas mais flexível, é implementar o USIG inteiramente em software, utilizando uma classe que emula as propriedades necessárias em métodos para incrementar o contador e retornar o valor em conjunto com a assinatura gerada para mensagem concatenada com o contador. Esta opção pode ser adequada para ambientes de teste ou quando as restrições de hardware impedem o uso de soluções mais robustas.

Implementação do USIG:

A funcionalidade do USIG pode ser implementada de duas formas principais:

USIG-HMAC (*Hash-Based Message Authentication Code*): Esta abordagem utiliza uma chave secreta compartilhada para gerar uma assinatura HMAC das mensagens. Cada mensagem é processada juntamente com o número sequencial para gerar um HMAC único, que é verificável por qualquer parte que possua a chave, garantindo que as mensagens não foram alteradas e mantendo a ordem de sua geração.

USIG-Sign (Assinatura Digital Direta): Alternativamente, o USIG pode utilizar algoritmos de assinatura digital para assinar diretamente cada mensagem junto com seu número sequencial. Esta abordagem proporciona um nível ainda mais alto de segurança, pois as chaves privadas não precisam ser compartilhadas ou conhecidas por outras partes do sistema. A validação da assinatura pode ser feita por qualquer participante que possua a chave pública correspondente, conferindo uma verificação irrefutável da autenticidade e da ordem das mensagens.

Codificação do USIG

Abaixo, é apresentado um pseudocódigo simplificado para ilustrar como o USIG pode ser implementado em duas das abordagens mencionadas anteriormente: USIG-HMAC e USIG-Sign. Este pseudocódigo serve para fornecer uma visão geral do fluxo de geração e validação dos identificadores sequenciais únicos.

Inicialização do USIG:

```
1 inicializar():
2     contador = 0
3     chave_secreta = gerar_chave_secreta()
4     chave_privada, chave_publica = gerar_chaves_de_assinatura()
```

Aplicações do USIG:

Além de ser empregado em sistemas de tolerância a falhas bizantinas, o USIG tem aplicabilidade em outros domínios que requerem integridade e sequencialidade rigorosas das

Geração de Identificador com HMAC:

```
gerar_usig_hmac(mensagem):
    id_mensagem = contador
    hmac = calcular_hmac(chave_secreta, mensagem + str(id_mensagem))
    contador += 1
    return id_mensagem, hmac
```

Validação de Identificador com HMAC:

```
validar_usig_hmac(mensagem, id_mensagem, hmac_recebido):
    hmac_esperado = calcular_hmac(chave_secreta,
                  mensagem + str(id_mensagem))
    if hmac_recebido == hmac_esperado:
        return True
    return False
```

Geração de Identificador com Assinatura Digital:

```
gerar_usig_sign(mensagem):
    id_mensagem = contador
    assinatura = assinar(chave_privada, mensagem + str(id_mensagem))
    contador += 1
    return id_mensagem, assinatura
```

Validação de Identificador com Assinatura Digital:

```
validar_usig_sign(mensagem, id_mensagem, assinatura_recebida):
    sucesso = verificar_assinatura(chave_publica,
              mensagem + str(id_mensagem), assinatura_recebida)
    Se sucesso:
        retorna verdadeiro
    retorna falso
```

mensagens, como sistemas financeiros, redes de comunicação críticas e infraestruturas de votação eletrônica.

Esta abordagem não apenas fortalece o sistema contra ataques externos, mas também aumenta a confiança entre os participantes do sistema, crucial em ambientes distribuídos onde a verificação independente da informação é fundamental.

3.2.3 Protocolo

Abaixo são apresentados os pseudo-códigos do sequenciador, clientes e réplicas do protocolo de consenso proposto para tolerar falhas bizantinas seguindo a abordagem NOPaxos.

O sequenciador inicia sua conexão com cada uma das réplicas. Ao receber uma mensagem ele utiliza o serviço USIG para assinar a mensagem e receber o contador. A nova

Codificação do sequenciador

```
Inicializa():
    Para cada replica:
        iniciar_conexao(replica)

Quando receber pacote/mensagem:
    contador, assinatura = gerar_usig_sign(mensagem)
    nova_mensagem = contador + mensagem
    header.adicionar(assinatura)
    enviar_mensagem_para_replicas(nova_mensagem)
```

mensagem então é gerado usando o contador e a mensagem antiga e a assinatura é inserida no *header*.

Codificação do cliente

```
Inicializa():
     Para cada replica:
        enviar_endereco(replica)
    iniciar_conexao(sequenciador)

Quando_receber_resposta(respota):
    tratar(resposta)
    Realizar_requisicao_ao_receber_resposta()

Realizar_requisicao_ao_receber_resposta():
    requisicao = gerar_requisicao()
    enviar_requisicao(sequenciador)
```

Cada cliente inicia um sinaliza as replica seu endereço e faz uma conexão com o sequenciador. Toda vez que uma nova resposta chega uma nova requisição é gerada e enviada ao sequenciador. O cliente então espera receber uma nova resposta para iniciar novamente o processo de gerar uma nova requisição.

Pseudo-código das réplicas

```
Inicializa():
    contador_replica = 0
    Para cada replica:
        iniciar_conexao(replica)
    iniciar_conexao(sequenciador)

Quando receber requisicao/mensagem:
    Se mensagem.contador > contador_replica entao:
        solicatar_pacotes_perdidos_replicas(contador_replica, mensagem.contador)
    Se mensagem.contador < contador_replica entao:
        retorna 0
    Se validar_usig_sign(mensagem, mensagem.contador, mensagem.assinatura) entao:
        resposta = executar(mensagem)
        enviar_resposta_cliente(resposta)
        contador = contador + 1
    Se nao:
        retorna 0
```

As replicas conectam-se entre si e com o sequenciador. Quando recebem uma mensagem verificam se o contador é maior do o esperado, se for então a réplica solicita os pacotes perdidos para as demais, se for menor ela não faz nada pois o pacote já foi processado, e se for igual valida a mensagem e caso seja valida a mensagem é processada e enviada ao cliente e o contador da replica é incrementado. Se a assinatura não for válida a réplica não faz nada.

3.3 Conclusões do Capítulo

Neste capítulo, exploramos inovações significativas na área de replicação de sistemas distribuídos, focando especialmente na redução dos custos de desempenho associados aos tradicionais protocolos de consenso. A introdução do NOPaxos como uma solução viável para mitigar a necessidade de coordenação intensiva entre réplicas representa um avanço crucial, demonstrando que é possível alcançar replicação consistente e robusta com uma abordagem que minimiza a latência e maximiza o throughput.

O desenvolvimento e a implementação do sequenciador no contexto do NOPaxos ilustra uma técnica eficaz para manter a ordem das mensagens com alta eficiência e baixa sobrecarga, facilitando o processamento rápido e confiável das requisições. Essa inovação é particularmente valiosa em cenários onde a disponibilidade e a consistência são críticas e que necessitam de um bom desempenho, ou não tem muitos recursos para alocar, tornando o desempenho crucial.

Além disso, a integração do serviço USIG em nosso protocolo aprimorado para tolerância a falhas bizantinas reforça a segurança e a integridade do sistema contra ataques maliciosos e falhas, promovendo um ambiente de replicação ainda mais resiliente. Este serviço assegura que todas as mensagens sejam assinadas e sequenciadas de forma única, aumentando a robustez do sistema contra ataques de *replay* e falsificação de mensagens.

Portanto, podemos concluir que o campo dos sistemas distribuídos ainda oferece um vasto território para inovação. A utilização coordenada e harmoniosa de tecnologias existentes, como demonstrado pela combinação de NOPaxos com o serviço USIG, evidencia que melhorias significativas em segurança, eficiência e escalabilidade são alcançáveis. A integração estratégica de soluções já conhecidas, mas ainda não exploradas conjuntamente, pode desbloquear novos níveis de desempenho e robustez para sistemas distribuídos. Assim, este capítulo não apenas destaca os avanços atuais mas também serve como um chamado para a continuação da pesquisa e desenvolvimento que exploram essas sinergias potenciais.

Capítulo 4

Experimentos

Este capítulo descreve a nossa avaliação experimental dos protocolos discutidos no capítulo anterior. Primeiramente, o ambiente e a metodologia empregada nos experimentos são apresentados. Por fim, os resultados obtidos são discutidos.

4.1 Ambiente Experimental

Para a realização dos experimentos, foi utilizado o testbed de rede Emulab[13], que oferece um ambiente flexível e controlado para a extração de dados e simulação do sistema distribuído a ser testado. O Emulab foi escolhido devido à sua capacidade de emular diversos tipos de redes e topologias, permitindo a replicação fiel das condições de um ambiente real em um ambiente experimental.

4.1.1 Topologia Experimental

A topologia escolhida para o experimento consistiu em cinco máquinas: uma máquina destinada à simulação dos clientes, uma máquina destinada ao sequenciador, e três máquinas para três réplicas, tolerando assim uma falha. A escolha dessa configuração visa simular um sistema distribuído onde diversos clientes enviam requisições ao sequenciador, que encaminha essas requisições de maneira ordenada para as réplicas, que por fim retornam os dados aos clientes.

A conexão entre as máquinas foi realizada utilizando a rede interna do Emulab, garantindo baixa latência e alta confiabilidade na comunicação entre os nós. A configuração da rede e a disposição das máquinas foram planejadas para minimizar interferências externas e maximizar a precisão dos resultados experimentais.

4.1.2 Especificações dos recursos utilizados e configuração de rede

Todas as máquinas utilizadas no experimento, incluindo réplicas, sequenciador e clientes, eram modelos referenciados como d430 no Emulab. As principais especificações técnicas dessas máquinas são [14]:

- **CPU:** Dois processadores Intel E5-2630v3 8-Core a 2,4 GHz (Haswell), totalizando 16 núcleos por máquina.
- **Memória RAM:** 64GB de memória ECC (*Error-Correcting Code*) DDR4 a 2133MT/s, distribuídos em 8 módulos de 8GB.
- **Armazenamento:** SSD SATA de 200 GB 6G, garantindo alta velocidade de leitura e escrita.

Essas especificações foram selecionadas para assegurar que os recursos de hardware não se tornassem um gargalo durante os experimentos, permitindo que a análise se concentre nos aspectos da rede e do sistema distribuído.

Outro aspecto essencial para o experimento foram as redes utilizadas, a rede necessitava ser estável e suportar um grande número de requisições para simular o ambiente real em que diversos clientes enviam mensagens simultaneamente. A rede utilizada possui duas componentes principais [14]:

- **Rede Externa:** Utilizada para a conexão remota com as máquinas. Esta rede é configurada para garantir segurança e acesso controlado aos recursos do Emulab.
- **Rede Interna:** Utilizada para realizar os experimentos. Esta rede é caracterizada por interfaces Ethernet de 10/100Mb, 1/10Gb, e é conhecida como "experiment network". Cada nó na rede interna possui pelo menos quatro interfaces, assegurando redundância e alta disponibilidade.

4.1.3 Software Utilizado

Para a criação de cada componente necessário para o funcionamento do algoritmo NOPaxos e da versão tolerante a falhas Bizantinas, foi utilizada a linguagem de programação Java na versão 1.8, também conhecida como Java 8. Esta versão foi escolhida devido à sua ampla adoção e estabilidade, além de possuir diversas ferramentas de suporte que facilitam sua utilização.

Para criar e manter os canais de comunicação do sistema distribuído, foi utilizada a biblioteca Netty, que permite abrir canais de comunicação e realizar o envio de mensagens entre canais de forma eficiente.

Para a serialização e desserialização dos objetos, foi utilizada a interface `Serializable` do Java, que lida com esses aspectos do código. Embora seja possível criar um método próprio para realizar esse processo e reduzir o tamanho dos objetos quando enviados, optamos por utilizar a interface padrão do java devido a sua simplicidade e eficiência.

4.2 Metodologia

Como explicado na seção anterior, o experimento foi realizado utilizando cinco computadores, sendo três para réplicas, um para o sequenciador e um para os clientes. Para evitar a necessidade de criar um novo projeto Java apenas para modificar os IPs das máquinas utilizadas e alterar a quantidade de clientes simulados, foi utilizado um arquivo de configurações. Neste arquivo, eram contidas informações como os IPs das réplicas, a quantidade de clientes que deveriam ser executados e o tamanho do *payload* a ser enviado, cada um desses valores associado a uma chave que era resgatada pelo código Java ao se inicializar.

Para a assinatura e verificação da assinatura foram usadas duas chaves privadas geradas anteriormente a execução do projeto, as chaves foram geradas a partir da biblioteca java *Bouncy Castle* e salvas em dois arquivos, um com a chave pública e outra com a chave privada. O sequenciador então lê o arquivo contendo a chave privada e a armazena em memória, usando-a para assinar os pacotes vindos dos clientes. As réplicas e clientes utilizam a chave pública que é utilizada pelo mesmo processo, o arquivo é lido e salvo em memória para ser usado.

Para cada máquina foi instanciada uma parte do sistema distribuído. Em três máquinas, foi utilizado o código responsável pelo controle das réplicas, cada uma trabalhando de forma independente, mas sincronizada, processando as mensagens recebidas pelo sequenciador na ordem correta. No caso do algoritmo de falhas bizantinas, as mensagens recebidas eram validadas. Uma máquina foi dedicada ao sequenciador, que recebe as mensagens dos clientes e as repassa para as réplicas, indicando a ordem em que devem ser processadas. No caso do algoritmo de falhas bizantinas, também envia a assinatura da mensagem junto com o número sequencial. Por fim, a última máquina fica responsável por instanciar os clientes, podendo simular inúmeros clientes simultaneamente utilizando *threads* para isso. A quantidade de clientes é parametrizável.

Os experimentos foram realizados simulando diferentes quantidades de clientes: 1, 5, 10, 25, 50, 100, 250, para os seguintes diferentes tamanhos de payload: 0, 100, 1024 e 4096 bytes. O processo foi realizado tanto para o algoritmo de *crash* quanto para o de falhas bizantinas. Cada cliente fazia requisições até obter a resposta da requisição 10.000, ou superior, após isso o cliente pára de mandar reaquisições. Um cliente era escolhido

para salvar os tempos de resposta das requisições para calcular a latência. Este cliente também enviar uma mensagem às réplicas avisando que havia finalizado suas requisições. As réplicas então gravavam o total de requisições executadas e o tempo total em que ficaram executando desde o início do teste, para calcular o throughput.

4.3 Resultados

Nessa seção apresentamos os resultados de desempenho encontrados ao avaliar os algoritmos do NoPaxos e o algoritmo modificado para tolerar falhas bizantinas utilizando o USIG. Os experimentos consistiam em micro-benchmarks, utilizando dados coletados a partir da execução do sistema, para avaliar o throughput e a latência dos algoritmos, uma comparação entre os algoritmos avaliando o desempenho com relação aos benefícios adquiridos, e a latência média de assinatura e validação da assinatura no algorítimo de falhas bizantinas.

Micro-benchmarks: Primeiramente iremos iniciar utilizando os resultados de micro-benchmarks comumente usados para avaliar sistemas de replicação de máquinas de estado. Esses benchmarks consistem em serviços "vazios" implementados com os algoritmos mencionados, para calcular o throughput no lado do servidor e a latência no lado do cliente.

Na Figura 4.1 temos os resultados tanto do algoritmo de falha por parada, NOPaxos, quando o algoritmo de falhas bizantinas utilizando o USIG, em ambos os casos foram considerados cargas de dados de tamanhos 0, 100, 1024 e 4096 bytes. Além do payload os pacotes tinham tamanhos mínimos, sendo de 190 bytes sem a assinatura e 536 bytes com a assinatura. Podemos ver que o algoritmo de falha por parada tem uma performance muito superior se comparado ao de falhas bizantinas, isso se deve ao tempo médio que o sequenciador leva para assinar o pacote que é de cerca de 1,81 milissegundos, e para a réplica verificar a validade do pacote a partir da assinatura que é de 0,22 milissegundos, dando um total de 2,03 milissegundos a mais na média, i.e., um tempo de 5 a 10 vezes maior do que as latências mais baixas do algoritmo de crash. Outro fator que impacta nesse acréscimo de tempo é o aumento da carga miníma do pacote que via para 536 bytes quando se trata da comunicação entre sequenciador e réplicas e entre réplicas e outras réplicas em caso de uma falha em que seja necessário requisitar os pacotes faltantes para as demais, o que aumenta o fluxo de bytes na rede resultando em uma comunicação mais lenta.

Utilizando as tabelas 4.1 e 4.2 podemos observar no algoritmo Bizantino que apesar da quantidade de operações processadas por segundo aumentar consideravelmente a latência também aumenta a partir dos 50 clientes, tendo ganhos mínimos no throughput para um aumento muito maior na latência se considerados os ganhos. O mesmo ocorre

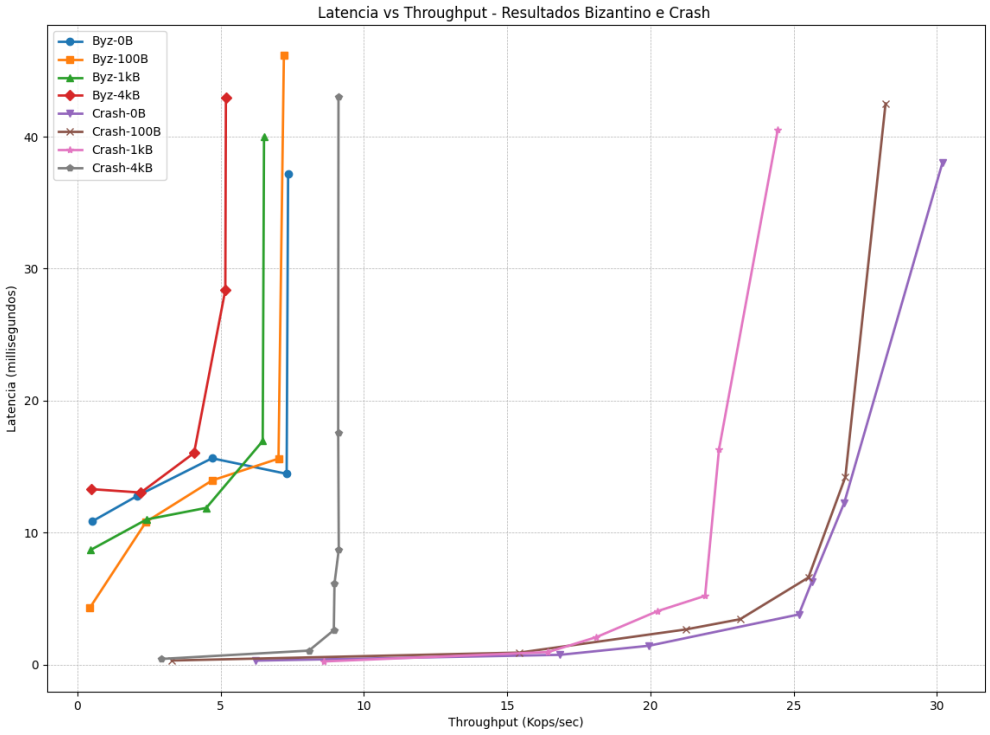

Figura 4.1: Gráfico da relação entre latência e throughput para ambos os algoritmos

para o algoritmo de falhas por parada (tabelas 4.3 e 4.4) porém em uma escala bem mais reduzida e com um impacto bem menor, precisando de um número de clientes consideravelmente maior para apresentar uma diferença significativa na latência, especialmente para os paylods menores.

Tabela 4.1: *Throughput* (operações/segundo) no algoritmo Bizantino, variando o número de clientes e o tamanho do *payload*

	Número de clientes					
Tamanho *payload*	**1**	**5**	**10**	**25**	**50**	**100**
0 bytes	0.516	2.096	4.697	7.299	7.353	7.353
100 bytes	0.434	1.797	4.702	7.018	7.207	7.568
1 kbyte	0.461	2.412	4.497	6.467	6.513	6.467
4 kbytes	0.486	2.212	4.071	4.576	5.182	5.350

Tabela 4.2: Latência (ms) 95 percentil do algoritmo Bizantino para separados por número de clientes e o tamanho do payload

	Número de clientes					
Size	**1**	**5**	**10**	**25**	**50**	**100**
0 bytes	10.85	12.79	15.63	14.45	37.19	77.37
100 bytes	4.30	10.81	13.95	15.60	46.18	76.08
1 kbyte	8.68	10.99	11.87	16.94	40.01	52.56
4 kbytes	13.29	13.03	16.03	28.39	42.96	124.08

Tabela 4.3: *Throughput* (operações/segundo) no algoritmo por Crash para separados por número de clientes e o tamanho do payload

	Número de clientes						
Size	**1**	**5**	**10**	**25**	**50**	**100**	**250**
0 bytes	6.22	16.85	19.96	25.18	25.63	26.76	30.20
100 bytes	3.30	15.43	21.25	23.13	25.52	26.81	28.21
1 kbyte	8.60	16.43	18.11	20.25	21.90	22.39	24.45
4 kbytes	2.93	8.08	8.95	8.97	9.12	9.10	9.11

Tabela 4.4: Latência (ms) 95 percentil do algoritmo Crash para separados por número de clientes e o tamanho do payload

	Número de clientes						
Size	**1**	**5**	**10**	**25**	**50**	**100**	**250**
0 bytes	0.29	0.73	1.42	3.79	6.27	12.26	38.01
100 bytes	0.30	0.9	2.66	3.43	6.59	14.21	42.52
1 kbyte	0.23	0.94	2.07	4.04	5.20	16.30	40.51
4 kbytes	0.42	1.05	2.60	6.14	8.70	17.58	43.03

Capítulo 5

Conclusão

O objetivo principal deste trabalho foi implementar o protocolo NOPaxos, e integrá-lo com o USIG para criar uma versão modificada capaz de tolerar também falhas bizantinas, sem adicionar complexidade significativa à lógica do sistema. Além disso, foram utilizadas chaves públicas e privadas para gerar e validar assinaturas digitais dentro do sistema USIG. A partir dessa implementação, coletamos dados que permitem avaliar a viabilidade de um algoritmo tolerante a falhas bizantinas em comparação com a versão simplificada que lida apenas com falhas por parada.

Embora os resultados sejam promissores vale destacar que a implementação enfrentou limitações de desempenho, principalmente em ambientes em que a baixa latência ou o alto número de operações processadas por segundo sejam necessários. Outra possível limitação seria um gargalo no sequenciador, pois no algoritmo bizantino ele gera uma assinatura e como visto nos resultados esse processo pode ser custoso no ponto de vista computacional e gerar um gargalo no sistema.

Algumas melhorias a serem feitas para reduzir esse impacto seriam a otimização do algoritmo de serialização e desserialização de dados, criando um algoritmo especifico para o pacote enviado, ou uma melhoria no processo de assinatura dos pacotes. Outra alternativa seria implementar um processo de assinatura em *batch* em que vários pacotes teriam a mesma assinatura, não necessitando assim gerar uma nova assinatura para cada pacote.

Acreditamos que a combinação desses algoritmos oferece novas oportunidades para o avanço na área de sistemas distribuídos, abordando tanto falhas por parada quanto falhas bizantinas. Também destacamos os impactos causados por cada abordagem, reforçando a importância de considerar as especificidades de cada sistema ao optar por uma implementação. O código do algoritmo desenvolvido está disponível publicamente no GitHub (`https://github.com/Faustino27/NOpaxos-implementation`), incentivando sua utilização, adaptação e aprimoramento por outros pesquisadores e profissionais interessados na área.

Referências

[1] Wong, David: *Real-World Criptography*. Manning, second edition, 2021. ix, 13, 14

[2] Schneider, Fred B.: *Implementing fault-tolerant services using the state machine approach: A tutorial*. ACM Computing Surveys (CSUR), 22(4):299–319, 1990. 1

[3] Tel, Gerard: *Introduction to Distributed Algorithms*. Cambridge University Press, second edition, 2000. 4

[4] Coulouris, George, Jean Dollimore, Tim Kindberg e Gordon Blair: *Distributed Systems: Concepts and Design*. Pearson Education, Inc., fith edition, 2011. 5

[5] Steen, Maarten van e Andrew S. Tanenbaum: *Distributed Systems*. Maarten van Steen, third edition, 2020. 7

[6] Fischer, Michael J., Nancy A. Lynch e Michael S. Paterson: *Impossibility of distributed consensus with one faulty process*. Journal of the ACM, 32(2):374–382, 1985. 10

[7] Lamport, Leslie: *The part-time parliament*. ACM Transactions on Computer Systems, 16(2):133–169, may 1998. 11

[8] Lamport, Leslie, Robert Shostak e Marshall Pease: *The byzantine generals problem*. ACM Transactions on Programming Languages and Systems, 4(3):387–401, 1982. 12

[9] Okawa, Manabu: *Online signature verification using locally weighted dynamic time warping via multiple fusion strategies*. IEEE Access, 10:40806–40817, 2022. 13

[10] Jialin, Li, Michael Ellis, Naveen Kr. Sharma, Szekeres Adriana e Dan R. K. Ports: *Just say no to paxos overhead: Replacing consensus with network ordering*. USENIX Symposium on Operating Systems Design and Implementation, 1(16), 2016. 16, 17, 18, 19

[11] Veronese, Giuliana Santos, Miguel Correia, Alysson Neves Bessani, Lau Cheuk Lung e Paulo Verissimo: *Efficient byzantine fault tolerance*. IEEE Transactions on Computers, 60(4):491–504, 2011. 16, 20

[12] Correia, M., N. F. Neves e P. Verissimo: *How to tolerate half less one byzantine nodes in practical distributed systems*. Em *Proceedings of the 23rd IEEE Symposium on Reliable Distributed Systems*, páginas 174–183. IEEE, October 2004. 21

[13] Emulab: *Emulab network testbed*. `https://www.emulab.net`. Ultimo acesso em: 20-ago-2024. 26

[14] Emulab Documentation: *Emulab documentation*. `https://docs.emulab.net/`. Acesso em: 19-ago-2024. 27

[15] Castro, Miguel e Barbara Liskov: *Practical byzantine fault tolerance*. Em *Proceedings of the Third Symposium on Operating Systems Design and Implementation (OSDI)*, páginas 173–186, Cambridge, MA, 1999. USENIX Association, USENIX Association. Massachusetts Institute of Technology, Laboratory for Computer Science, 545 Technology Square, Cambridge, MA 02139.