Suporte à Comunicação e ao Gerenciamento de Estado em Replicação por Máquina de Estados

Tese de Doutorado submetida à
Faculdade de Informática da Università della Svizzera Italiana (USI)
em um acordo de *cotutelle de thèse* com a
Escola Politécnica da Pontifícia Universidade Católica do Rio Grande do Sul (PUCRS)
em cumprimento parcial dos requisitos para a obtenção do grau de
Doutor em Filosofia

apresentada por
Eliã Rafael de Lima Batista

sob a supervisão de
Fernando Pedone (USI) e Fernando Luís Dotti (PUCRS)

Fevereiro de 2026

Comitê de aprovação

Alysson Bessani	Universidade de Lisboa, Portugal
Antonio Carzaniga	Universidade da Suíça Italiana, Suíça
Luiz Gustavo Leão Fernandes	Pontificia Universidade Católica Rio Grande do Sul, Brasil
Patrick Thomas Eugster	Universidade da Suíça Italiana, Suíça
Valerio Schiavoni	Universidade de Neuchâtel, Suíça

Tese recebida em 23 Fevereiro 2026

Orientador

Fernando Pedone (USI)

Co-Orientador

Fernando Luís Dotti (PUCRS)

Diretor do Programa de Doutorado

Walter Binder

Declaro que, exceto nos casos em que o devido reconhecimento foi dado, o trabalho apresentado nesta tese é de autoria exclusiva do autor; que este trabalho não foi previamente submetido, no todo ou em parte, para a obtenção de qualquer outro título acadêmico; e que o conteúdo desta tese é resultado de atividades realizadas desde a data oficial de início do programa de pesquisa aprovado.

Eliã Rafael de Lima Batista
Lugano, 23 Fevereiro 2026

Dedicado à minha doce filha Ellie

Consagre ao Senhor tudo o que você faz, e os seus planos serão bem-sucedidos.

Proverbios 16:3

Resumo

A Replicação por Máquina de Estados (State Machine Replication – SMR) é uma abordagem fundamental para a construção de sistemas distribuídos tolerantes a falhas, garantindo que múltiplas réplicas mantenham um estado consistente apesar de falhas. No entanto, alcançar simultaneamente forte consistência e alto desempenho continua sendo um desafio, devido à sobrecarga de coordenação entre réplicas e de sincronização de estado. Esta tese aborda esses desafios ao propor novos mecanismos de comunicação e gerenciamento de estado para sistemas SMR.

Apresentamos o FlexCast, um protocolo de multicast atômico genuíno baseado em overlay, que combina localidade geográfica com entrega eficiente de mensagens por meio de um overlay DAG completo. Ele reduz a latência em implantações geograficamente distribuídas e elimina a sobrecarga de comunicação. Além disso, introduzimos *quiescência*, uma nova propriedade de multicast atômico proposta neste trabalho, que complementa a *minimalidade* ao garantir que os processos cessem a comunicação quando não estão mais envolvidos em nenhum multicast, reforçando assim tanto a eficiência quanto a genuinidade. Adicionalmente, introduzimos um mecanismo de reconfiguração para o FlexCast, que adapta dinamicamente seu overlay à localidade da carga de trabalho, mantendo alto desempenho sob condições de rede variáveis.

Também abordamos a transferência e a verificação eficientes de estado em sistemas replicados. Propomos duas estruturas de dados clusterizadas e autovalidáveis distintas, que possibilitam sincronização de estado eficiente, paralela e independentemente verificável, reduzem latências de cauda, melhoram os processos de checkpointing e recuperação, e permanecem robustas sob falhas Bizantinas. A integração em um framework SMR prático demonstra sua eficácia tanto em cenários normais quanto adversariais em redes de longa distância.

De forma geral, esta tese apresenta um conjunto abrangente de técnicas para melhorar o desempenho, a escalabilidade e a tolerância a falhas de sistemas SMR, combinando comunicação eficiente, adaptação dinâmica e gerenciamento avançado de estado para atender às demandas de aplicações distribuídas modernas.

Agradecimentos

Antes de tudo, agradeço a Deus por me conceder vida, saúde e força ao longo de toda esta jornada.

Expresso minha mais profunda gratidão à minha esposa, Larissa. Seu amor infinito, paciência e apoio foram meu alicerce em cada desafio. Obrigado por acreditar em mim, por estar sempre ao meu lado e por assumir tantas responsabilidades para que eu pudesse perseguir este sonho. À minha preciosa filha, Eloise, que enche meus dias de alegria e me lembra do que realmente importa.

Aos meus pais, Carlos e Mari, por seu amor incondicional, sacrifícios e orações que moldaram quem eu sou hoje. Obrigado por me ensinarem, desde cedo, o valor da dedicação, da fé e da perseverança. À minha família e amigos, que me apoiaram e incentivaram de inúmeras formas, especialmente à minha irmã, Eliadi, e ao seu marido, Emerson, pelo carinho e apoio constantes. Estendo também minha gratidão aos meus sogros, Djalma e Hilza, e aos cunhados, Thiago e Leticia, e aos seus filhos, Livia e Felipe, pelo incentivo e pelas orações.

Sou sinceramente grato aos meus orientadores, Prof. Fernando Pedone e Prof. Fernando Dotti, pela orientação, conhecimento e confiança ao longo de toda a pesquisa. Sua mentoria foi inestimável para meu desenvolvimento acadêmico e pessoal. Agradeço ainda ao Prof. Eduardo Alchieri e ao Prof. Paulo Coelho pelas colaborações, discussões enriquecedoras e ideias, que contribuíram significativamente para o desenvolvimento da maioria das contribuições apresentadas nesta tese. Também agradeço a Michele Cattaneo por seu trabalho no projeto e na implementação da árvore B+AVL. Por fim, agradeço aos membros da banca examinadora pelo tempo dedicado, pela leitura cuidadosa e pelos comentários construtivos, que ajudaram a melhorar a qualidade e a clareza da versão final desta tese. Agradeço também a todos os meus colegas do Distributed Systems Laboratory (DSLab), liderado pelo Prof. Pedone, pelo ambiente de pesquisa estimulante, pelas discussões valiosas e pelas trocas diárias que enriqueceram minha experiência de doutorado.

Por fim, esta conquista não é apenas minha, mas um testemunho do amor, apoio e conhecimento que recebi de cada um de vocês. Muito obrigado.

Prefácio

Todos os esforços de pesquisa que realizei durante meu doutorado, incluindo as contribuições apresentadas nesta tese e as colaborações com outros membros do nosso grupo de pesquisa, resultaram nas seguintes publicações:

- E. Batista, P. Coelho, E. Alchieri, F. Dotti, and F. Pedone. *Robust and efficient replication with CAV data structures*. Submitted to the ACM Symposium on Cloud Computing (SOCC) 2026.
- L. Martignetti, E. Batista, G. Cugola, and F. Pedone. *TRAM: Tree-based atomic multicast on RDMA*. Submitted to the ACM Symposium on Cloud Computing (SOCC) 2026.
- M. Cattaneo, E. Batista, and F. Pedone. *B+AVL trees: towards data structures for robust and efficient blockchain state synchronization*. Proceedings of the 44th International Symposium on Reliable Distributed Systems, 2025.
- E. Batista, P. Coelho, E. Alchieri, F. Dotti, and F. Pedone. *Reconfiguring atomic multicast*. Submitted to IEEE Transactions on Dependable and Secure Computing, 2024.
- E. Batista, P. Coelho, E. Alchieri, F. Dotti, and F. Pedone. *FlexCast: Genuine Overlay-based Atomic Multicast*. Proceedings of the 24th International Middleware Conference, 2023, Pages 288-300.
- E. Batista, E. Alchieri, F. Dotti, and F. Pedone. *Early Scheduling on Steroids: Boosting Parallel State Machine Replication*. Journal of Parallel and Distributed Computing, Volume 163, 2022, Pages 269-282.

Índice

Capítulo 1

Introdução

Sistemas distribuídos modernos são projetados para fornecer serviços ininterruptos mesmo na presença de falhas. Desde infraestruturas de nuvem em escala global até redes blockchain e bancos de dados replicados, as aplicações atuais dependem da replicação para garantir disponibilidade, tolerância a falhas e desempenho. A replicação permite que múltiplos servidores mantenham cópias dos mesmos dados ou serviços, de modo que, se um falhar, outros possam continuar operando. No entanto, garantir que essas réplicas se comportem de forma consistente está longe de ser trivial.

O desafio reside no fato de que sistemas distribuídos devem operar sob incerteza: mensagens podem ser atrasadas, perdidas ou reordenadas; máquinas podem falhar ou se comportar de forma maliciosa; e, ainda assim, os clientes esperam que o sistema se comporte como se houvesse apenas um único servidor confiável. Essa tensão entre tolerância a falhas e consistência tornou a replicação um dos problemas centrais e mais desafiadores da computação distribuída.

A Replicação por Máquina de Estados (State Machine Replication – SMR) oferece uma solução fundamentada para esse problema. Ela permite que um sistema tolere falhas ao replicar uma máquina de estados determinística em múltiplos nós, garantindo que todas as réplicas corretas executem as mesmas operações na mesma ordem e, assim, alcancem o mesmo estado [65, 90]. A SMR tem sido implantada com sucesso em diversos sistemas do mundo real, como bancos de dados distribuídos [26, 92], serviços de coordenação em larga escala [91, 93], e plataformas blockchain [15, 94], demonstrando sua versatilidade e importância como base para a computação distribuída confiável.

Apesar de sua ampla adoção, implementar SMR de forma eficiente continua sendo um desafio significativo. A coordenação entre réplicas exige protocolos de comunicação capazes de garantir acordo sobre a ordem das operações mesmo na presença de falhas. Além disso, à medida que os estados do sistema crescem ou evoluem dinamica-

mente, manter e sincronizar esses estados de forma eficiente entre as réplicas torna-se cada vez mais complexo. Esses desafios impactam diretamente o desempenho, a escalabilidade e a recuperação de falhas, aspectos essenciais para implantações modernas que abrangem redes de longa distância e infraestruturas geograficamente distribuídas.

O trabalho apresentado nesta tese aborda esses desafios ao propor novas técnicas para comunicação entre réplicas, gerenciamento de estado e sincronização em sistemas SMR. Em particular, investiga-se como otimizar overlays de comunicação para escalabilidade e localidade, e como projetar estruturas de dados que possibilitem sincronização de estado rápida e verificável. Em conjunto, essas contribuições visam melhorar o desempenho e a resiliência de sistemas SMR, preservando ao mesmo tempo suas fortes garantias de correção.

1.1 Da replicação à Replicação por Máquina de Estados

Tendo apresentado a motivação para a replicação e o papel da Replicação por Máquina de Estados (SMR) em sistemas confiáveis, fornecemos agora uma breve visão geral dos conceitos fundamentais por trás da SMR e dos desafios que ela aborda.

1.1.1 Replicação e acordo

A replicação tem como objetivo melhorar tanto a disponibilidade quanto a confiabilidade de sistemas distribuídos por meio da manutenção de múltiplas cópias de dados ou serviços. Quando todas as réplicas devem se comportar de forma consistente, elas precisam concordar sobre a sequência de operações a serem executadas, um problema conhecido como *consenso*. Alcançar consenso não é trivial porque ambientes distribuídos são inerentemente não confiáveis: mensagens podem ser atrasadas, perdidas ou reordenadas, e processos podem falhar a qualquer momento. Resultados clássicos, como a impossibilidade FLP [37], demonstram que mesmo acordos simples não podem ser garantidos sob assincronia completa, o que motivou décadas de pesquisa em protocolos práticos tolerantes a falhas que assumem hipóteses realistas sobre temporização.

1.1.2 Replicação por Máquina de Estados

A Replicação por Máquina de Estados (SMR) fornece uma abordagem geral para a implementação de serviços tolerantes a falhas por meio da replicação [65, 90]. A ideia é modelar o serviço como uma máquina de estados determinística replicada em múltiplos nós. Cada réplica inicia no mesmo estado inicial e executa a mesma sequência de

operações na mesma ordem. Como resultado, todas as réplicas corretas permanecem consistentes, mesmo que algumas réplicas falhem ou se recuperem posteriormente.

A correção da SMR baseia-se em duas propriedades fundamentais: *segurança* e *vivacidade* [90]. A segurança garante que todas as réplicas corretas executem as mesmas operações na mesma ordem, prevenindo estados divergentes. A vivacidade assegura que toda operação solicitada por um cliente correto seja eventualmente executada por todas as réplicas corretas. Uma propriedade relacionada, a *linearizabilidade* [54], define o comportamento observável do sistema, exigindo que toda operação pareça ocorrer atomicamente entre sua invocação e sua resposta.

1.1.3 Comunicação e ordenação

A coordenação eficiente entre réplicas requer mecanismos de comunicação que garantam uma ordem total ou parcial consistente das operações. Protocolos como *atomic broadcast* e *atomic multicast* são blocos fundamentais da SMR. Eles garantem que todas as réplicas entreguem o mesmo conjunto de mensagens na mesma ordem, mesmo na presença de falhas e atrasos de rede. Projetar esses protocolos é desafiador, pois eles devem alcançar simultaneamente alta tolerância a falhas, baixa latência e escalabilidade. Para enfrentar esses desafios, pesquisadores propuseram comunicação baseada em overlay, multicast genuíno e protocolos de multicast tolerantes a falhas (por exemplo, [22, 50, 51, 89]), cada um oferecendo diferentes compromissos entre desempenho e confiabilidade. Esses mecanismos formam a base para as otimizações de comunicação exploradas no Capítulo 3.

1.1.4 Gerenciamento de estado e recuperação

Manter o estado consistente das réplicas é igualmente crucial. Quando uma réplica falha e posteriormente se recupera, ou quando novas réplicas ingressam no sistema, elas devem sincronizar seu estado com outras réplicas sem afetar significativamente o desempenho. Estruturas eficientes de gerenciamento de estado possibilitam sincronização rápida e validação de estados grandes, o que é particularmente importante em sistemas com cargas de trabalho dinâmicas ou grandes volumes de dados. Estruturas de dados como árvores de Merkle [76] e variantes clusterizadas (por exemplo, [24, 41, 46, 57]) permitem representações de estado compactas e verificáveis. O Capítulo 4 introduz novas estruturas de dados que se baseiam nesses princípios para fornecer transferência de estado eficiente, verificável e paralelizável.

Em resumo, a SMR oferece uma base teórica robusta para tolerância a falhas, mas sua realização prática em ambientes de larga escala ou geograficamente distribuídos

permanece desafiadora. Esta tese explora como superar esses desafios por meio de overlays de comunicação otimizados e mecanismos eficientes de gerenciamento de estado que melhoram o desempenho e a resiliência sem comprometer a correção.

1.2 Contribuições desta tese

Apesar dos avanços em protocolos de comunicação e estruturas de gerenciamento de estado, sistemas SMR ainda enfrentam gargalos que limitam a vazão, aumentam a latência ou dificultam a escalabilidade para implantações maiores. Coordenação eficiente, multicast tolerante a falhas e sincronização de estado continuam sendo áreas ativas de pesquisa. Esta tese aborda essas lacunas ao propor novas técnicas de comunicação e gerenciamento de estado que melhoram o desempenho preservando a segurança (isto é, linearizabilidade) e a vivacidade.

As principais contribuições desta tese incluem novos protocolos de multicast atômico e mecanismos de comunicação baseados em overlay para coordenação eficiente entre réplicas; estruturas eficientes de gerenciamento de estado com integração prática em sistemas SMR; e uma avaliação abrangente que demonstra melhorias de desempenho, tolerância a falhas e escalabilidade em relação às abordagens existentes. Em mais detalhes, as quatro contribuições são as seguintes.

FlexCast Na primeira contribuição, propusemos o FlexCast, um protocolo de multicast atômico genuíno baseado em overlay que combina conectividade reduzida e localidade geográfica por meio do uso de um overlay DAG completo, no qual cada grupo toma decisões locais de ordenação. Também introduzimos uma nova propriedade, quiescência, que refina a propriedade de minimalidade do multicast atômico para evitar comunicação desnecessária e garantir a integridade de protocolos de multicast atômico genuínos. Nossa avaliação mostra que o FlexCast é sensível à escolha do overlay em termos de latência, mas supera consistentemente um protocolo genuíno distribuído e apresenta desempenho comparável ou superior ao de um protocolo hierárquico em cargas de trabalho típicas. Diferentemente de protocolos hierárquicos, o FlexCast não introduz sobrecarga de comunicação, ao mesmo tempo em que fornece ordenação acíclica global a partir de decisões locais por meio de um mecanismo baseado em histórico. Seu projeto, implementação e avaliação utilizando o benchmark gTPC-C, que estende o TPC-C com distribuição geográfica, demonstram reduções significativas de latência em implantações geograficamente distribuídas.

FlexCast reconfiguration Na segunda contribuição, estendemos o FlexCast com um mecanismo de reconfiguração. O esquema de reconfiguração proposto permite que o

FlexCast adapte dinamicamente seu overlay de comunicação a mudanças na localidade da carga de trabalho, melhorando o desempenho em condições variáveis e tornando o sistema adequado para ambientes dinâmicos. A avaliação experimental mostra que o protocolo se ajusta de forma eficaz às cargas de trabalho atuais, oferecendo melhor desempenho com novas configurações.

B+AVL trees No que diz respeito ao gerenciamento de estado, nossa terceira contribuição abordou o desafio da transferência eficiente de estado em sistemas blockchain, particularmente para a recuperação de nós fora de sincronia. Soluções existentes baseadas em snapshots e validação via Merkle frequentemente enfrentam limitações em eficiência de cluster e tamanho das provas. Para superar esses problemas, propusemos as B+AVL trees, uma nova estrutura de dados que combina as propriedades de balanceamento e verificação das árvores AVL com a eficiência de espaço das B+Trees. Diferentemente de abordagens anteriores, as B+AVL trees simplificam a manutenção de clusters ao impedir o movimento de folhas entre clusters e fornecem provas de validação mais compactas. Resultados experimentais mostram que as B+AVL trees mantêm desempenho consistente em inserções e buscas, otimizam a utilização de cache por meio do armazenamento contíguo de chaves e produzem provas de Merkle menores e mais estáveis. Elas também alcançam eficiência espacial superior à maioria das alternativas, particularmente para clusters menores. Na sincronização de estado, as B+AVL trees superam árvores de referência tanto em cenários não Bizantinos quanto Bizantinos, graças à serialização eficiente, ao layout compacto de memória e a clusters autovalidáveis que permitem detecção e recuperação rápidas sob ataque.

SVCSkipList Por fim, nossa quarta contribuição introduziu o conceito de estruturas de dados clusterizadas autovalidáveis como uma abordagem fundamentada para aprimorar o gerenciamento de estado em replicação por máquina de estados tolerante a falhas Bizantinas (BFT SMR). Essa ideia é concretizada no SVCSkipList, uma nova estrutura de dados que possibilita transferência de estado eficiente, paralela e independentemente verificável. Ao integrar fortemente a representação e a validação de estado ao framework de replicação, o SVCSkipList aborda limitações críticas de bibliotecas SMR existentes, particularmente na presença de falhas Bizantinas. A avaliação experimental, conduzida em um ambiente de rede de longa distância (WAN) com integração ao framework BFT-SMaRt [12], demonstra melhorias robustas de desempenho em todas as fases de execução. Durante a execução normal, o SVCSkipList alcança vazão comparável ou superior à linha de base, com redução de até 60% nas latências de cauda para estados de grande porte. Os tempos de checkpointing são reduzidos pela metade para estados grandes (até 4GB) devido à multiversionamento eficiente. A sin-

cronização de estado é mais rápida e consistente, tanto em operação normal quanto sob ataque Bizantino, com tempos de sincronização até 64% menores para grandes grupos de réplicas. A validação independente de clusters e o processamento paralelo possibilitam recuperação rápida sob ataque, evitando as tentativas em cascata que afetam implementações de referência.

1.3 Organização da tese

O restante da tese está organizado da seguinte forma: o Capítulo 2 introduz o modelo de sistema e conceitos de base. O Capítulo 3 foca na comunicação em sistemas SMR, incluindo protocolos de multicast atômico, o FlexCast e mecanismos de reconfiguração. O Capítulo 4 apresenta o gerenciamento de estado em sistemas SMR, detalhando as B+AVL trees, o SVCSkipList e sua avaliação. Por fim, o Capítulo 5 conclui a tese com um resumo das contribuições e possíveis direções futuras.

Capítulo 2

Modelo de Sistema e Fundamentação

Este capítulo introduz nosso modelo de sistema e revisita definições fundamentais da Replicação por Máquina de Estados (State Machine Replication – SMR), com ênfase particular em dois aspectos centrais às nossas contribuições: comunicação e gerenciamento de estado.

A Replicação por Máquina de Estados (SMR) é tipicamente definida como um sistema distribuído baseado em troca de mensagens, composto por um conjunto ilimitado de processos clientes $C = \{c_1, c_2, \ldots\}$ e um conjunto limitado de processos servidores $S = \{p_1, p_2, \ldots, p_n\}$. Um processo cliente representa uma entidade externa que interage com o serviço replicado submetendo requisições (por exemplo, comandos da aplicação) e recebendo respostas. Os clientes não participam dos protocolos de replicação ou tolerância a falhas; em vez disso, confiam nos servidores para executar suas requisições de forma consistente. Os servidores são responsáveis por coordenar entre si para garantir a emulação correta de uma máquina de estados tolerante a falhas. Os clientes emitem requisições ao sistema, enquanto os servidores executam essas requisições de forma determinística e retornam respostas, emulando assim o comportamento de uma única máquina de estados tolerante a falhas [65, 90].

Um processo é dito *correto* se segue o protocolo e nunca falha, e *falho* caso contrário. Processos falhos podem apresentar dois tipos fundamentais de falhas: *falhas por parada (crash)*, nas quais um processo interrompe sua execução prematuramente e não realiza mais nenhuma ação, e *falhas Bizantinas*, nas quais um processo se comporta de forma arbitrária, potencialmente desviando do protocolo prescrito de maneira imprevisível ou até maliciosa. Falhas por parada são geralmente mais fáceis de tolerar, enquanto falhas Bizantinas representam um modelo adversarial estritamente mais geral e desafiador.

Assumimos um modelo de comunicação por troca de mensagens no qual os processos interagem invocando as primitivas $\mathsf{send}(m, q)$ e $\mathsf{receive}(m)$. Um processo p invoca

send(m, q) para transmitir uma mensagem m a um processo destino q, e um processo q invoca receive(m) para obter uma mensagem enviada a ele. Os canais de comunicação são ponto a ponto e assíncronos, o que significa que os atrasos na transmissão de mensagens são finitos, porém ilimitados, e as mensagens podem ser entregues fora de ordem.

No contexto desta tese, adotamos ambos os modelos de falha, dependendo da natureza da contribuição. Especificamente, nas contribuições relacionadas à comunicação, focamos na tolerância a falhas por parada, refletindo o modelo comumente assumido em sistemas SMR de alto desempenho. Em contraste, nas contribuições relacionadas ao gerenciamento de estado, também consideramos falhas Bizantinas, uma vez que mecanismos de transferência e recuperação de estado devem permanecer corretos e seguros mesmo em ambientes adversariais. Essa perspectiva dupla nos permite explorar tanto eficiência quanto robustez em diferentes componentes da SMR.

Assumimos canais de comunicação confiáveis entre processos corretos. Em particular, o sistema satisfaz a seguinte propriedade: se um processo correto p envia uma mensagem m a um processo correto q, então q eventualmente recebe m. Mensagens não são criadas nem duplicadas pelo subsistema de comunicação; isto é, toda mensagem recebida foi previamente enviada por algum processo, e cada mensagem é recebida no máximo uma vez. Essa suposição é padrão em modelos de sistemas SMR e permite que o protocolo foque no tratamento de falhas de processos, em vez de omissões de comunicação.

Assumimos que o sistema é parcialmente síncrono [35]: ele é inicialmente assíncrono e eventualmente se torna síncrono. O instante em que o sistema se torna síncrono é chamado de Tempo Global de Estabilização (Global Stabilization Time – GST), e é desconhecido pelos processos. Antes do GST, não há limites para atrasos de comunicação e processamento; após o GST, tais limites existem, mas são desconhecidos.

Na SMR, as interações entre clientes e réplicas servidoras devem ser coordenadas para garantir que todas as réplicas corretas exibam a mesma evolução de estado. Um sistema SMR deve satisfazer as propriedades de segurança e vivacidade. A propriedade de segurança possibilita a implementação de serviços fortemente consistentes, satisfazendo critérios de consistência forte, enquanto a vivacidade garante o progresso do sistema. Formalmente, essas propriedades podem ser definidas da seguinte forma:

- *Segurança:* Um sistema distribuído satisfaz segurança se nada de ruim jamais acontece. Formalmente, propriedades de segurança afirmam que todas as execuções do sistema evitam um conjunto de estados "ruins". A *linearizabilidade* é o critério de consistência utilizado para implementar a propriedade de segurança, garantindo que cada operação pareça surtir efeito atomicamente em algum instante entre sua invocação e sua resposta, de forma que a ordem em tempo

real de operações não sobrepostas seja preservada e o histórico resultante seja consistente com a especificação sequencial do objeto [54].

- *Vivacidade:* Um sistema distribuído satisfaz vivacidade se algo de bom eventualmente acontece. Formalmente, propriedades de vivacidade afirmam que, em toda execução, progresso é eventualmente realizado — isto é, toda requisição que deve ser tratada será eventualmente tratada.

A implementação dessas propriedades requer o seguinte [90]:

- *Estado inicial:* Todas as réplicas corretas iniciam no mesmo estado.
- *Determinismo:* Todas as réplicas corretas que recebem a mesma entrada no mesmo estado produzem saídas e estados resultantes idênticos.
- *Coordenação:* Todas as réplicas corretas processam a mesma sequência de operações.

Protocolos de coordenação de réplicas implementam em seu núcleo um algoritmo de *multicast de ordem total* e/ou *consenso* [53]. O determinismo pode impactar o desempenho do sistema, pois restringe o uso de múltiplas threads e núcleos [4]. Para superar essa limitação, muitas abordagens focam na Replicação por Máquina de Estados Paralela (Parallel State Machine Replication – PSMR), cujo objetivo é paralelizar a execução de requisições independentes, mantendo a correção. A ideia central é identificar operações não conflitantes que possam ser executadas concorrentemente nas réplicas, explorando assim arquiteturas multicore para melhorar a vazão.

Diversas estratégias foram propostas para alcançar execução paralela em PSMR. Em CBASE [62], por exemplo, as réplicas são estendidas com um escalonador determinístico. Com base na semântica da aplicação, ele serializa a execução de requisições conflitantes respeitando a ordem de entrega e despacha requisições não conflitantes para execução paralela. A detecção de conflitos é feita por meio de um grafo de dependências, que organiza as requisições para serem processadas o mais cedo possível (sempre que as dependências forem executadas).

Em [74], os autores evitam um escalonador central ao mapear estaticamente requisições para diferentes grupos de multicast. Requisições não conflitantes são multicastadas para grupos distintos, que ordenam parcialmente as requisições entre as réplicas. Requisições conflitantes são multicastadas para o mesmo grupo. Em uma réplica, cada thread é associada a um grupo e processa requisições à medida que elas chegam. Requisições entregues por diferentes grupos são executadas concorrentemente. O sistema Eve [59] utiliza abordagens otimistas que podem levar a sobrecarga adicional em alguns casos, nos quais as réplicas comparam os resultados da execução otimista usando

consenso e, caso haja divergência, realizam rollback e reexecutam as requisições de forma conservadora.

Em [5], os autores apresentam uma técnica de escalonamento antecipado, na qual parte das decisões de escalonamento é tomada antes da ordenação das requisições. Após as requisições serem ordenadas, o escalonamento deve respeitar essas restrições. Em [7], apresentamos uma análise de utilização de recursos dessa técnica de escalonamento antecipado, avaliando o impacto de tais restrições e mostrando que threads podem permanecer ociosas enquanto requisições independentes pendentes estão disponíveis para execução, levando a uma utilização ineficiente do processador. Assim, em [8], estendemos a técnica de escalonamento antecipado incorporando work-stealing e outros mecanismos de sincronização para alcançar uma carga de trabalho mais equilibrada e melhorar o desempenho de execução.

Outra abordagem para melhorar o desempenho da SMR é enfraquecer o requisito de ordem total das requisições nas réplicas. Em particular, apenas requisições conflitantes precisam ser ordenadas de forma consistente entre as réplicas. Generalized Paxos [68], Generic Broadcast [83], Egalitarian Paxos [77], e Mencius [73] são exemplos de protocolos que adotam essa abordagem.

Ao explorar essas técnicas, a PSMR pode aumentar significativamente a vazão enquanto preserva as garantias de segurança e linearizabilidade da SMR tradicional. No entanto, o projeto deve equilibrar cuidadosamente paralelismo e determinismo, pois concorrência excessiva pode levar a sincronizações complexas e sobrecarga de rollback. Essas otimizações são ortogonais às abordagens descritas nesta tese e podem ser combinadas com elas para melhorar ainda mais o desempenho.

2.1 Multicast atômico

O multicast atômico é uma abstração fundamental de comunicação em sistemas distribuídos confiáveis, encapsulando a complexidade de propagar e ordenar mensagens de forma consistente entre subconjuntos de processos. Com o multicast atômico, um cliente pode multicastar uma mensagem para diferentes grupos de servidores, e a abstração garante que todos os destinos corretos entreguem a mensagem em uma ordem consistente. A seguir, formalizamos essas garantias de confiabilidade e ordenação.

Para implementar multicast atômico tolerante a falhas, os servidores são organizados em *grupos de processos*. Cada grupo se comporta como um único processo lógico do ponto de vista da abstração de multicast, mascarando falhas individuais de servidores dentro do grupo. A replicação dentro de um grupo garante que o grupo como um todo possa continuar participando do multicast mesmo que alguns de seus membros falhem. Formalmente, definimos o conjunto de grupos de servidores como

$\Gamma = \{G_A, G_B, \ldots, G_N\}$, onde, para todo $g \in \Gamma$, $g \subseteq S$. Além disso, os grupos são não vazios e disjuntos [22, 50, 51, 89], garantindo que cada servidor pertença exatamente a um grupo.

Um cliente multicast atômico uma mensagem de aplicação m para um conjunto de grupos invocando a primitiva multicast(m), onde $m.sender$ denota o processo que chama multicast(m), $m.id$ é o identificador único da mensagem, e $m.dst$ é o conjunto de grupos de destino para os quais m é multicastada. Um servidor entrega uma mensagem m invocando a primitiva deliver(m). Se $|m.dst| = 1$, dizemos que m é uma mensagem *local*; se $|m.dst| > 1$, dizemos que m é uma mensagem *global*.

Definimos a relação $\prec$ sobre o conjunto de mensagens entregues por processos servidores da seguinte forma: $m \prec m'$ se, e somente se, existe um processo que entrega m antes de m'. Se $m \prec m'$ ou $m' \prec m$, dizemos que há uma dependência entre m e m'.

O multicast atômico satisfaz as seguintes propriedades [52]:

- *Validade*: Se um processo correto p multicastar uma mensagem m, então eventualmente todos os processos servidores corretos $q \in g$, onde $g \in m.dst$, entregam m.
- *Acordo*: Se um processo p entrega uma mensagem m, então eventualmente todos os processos servidores corretos $q \in g$, onde $g \in m.dst$, entregam m.
- *Integridade*: Para qualquer processo p e qualquer mensagem m, p entrega m no máximo uma vez, e somente se $p \in g$, $g \in m.dst$, e m foi previamente multicastada.
- *Ordem de prefixo*: Para quaisquer duas mensagens m e m' e quaisquer dois processos servidores p e q tais que $p \in g$, $q \in h$ e $\{g,h\} \subseteq m.dst \cap m'.dst$, se p entrega m e q entrega m', então ou p entrega m' antes de m, ou q entrega m antes de m'.
- *Ordem acíclica*: A relação $\prec$ é acíclica.

Em um protocolo de multicast atômico genuíno, apenas o remetente e os destinos de uma mensagem coordenam para ordená-la. Um protocolo de multicast atômico genuíno não depende de um grupo fixo de processos e não envolve processos desnecessariamente. Mais precisamente, um algoritmo de multicast atômico genuíno deve garantir a seguinte propriedade [51].

- *Minimalidade*: Se um processo p envia ou recebe uma mensagem em uma execução R, então alguma mensagem m é multicastada em R, e p é $m.sender$ ou pertence a um grupo em $m.dst$.

2.2 Gerenciamento de estado

Satisfazer o estado inicial em um sistema SMR não é trivial em cenários práticos nos quais réplicas falham (ou são desligadas para manutenção) e posteriormente se recuperam com um estado possivelmente desatualizado. Consequentemente, o gerenciamento de estado da replicação é um componente vital, garantindo o tratamento consistente e confiável do estado entre as réplicas para dar suporte à tolerância a falhas. O gerenciamento de estado também possibilita a implementação de durabilidade [11] e reconfigurações [12] que incorporam novas réplicas ao sistema. Além disso, réplicas atrasadas podem utilizar esse componente como um mecanismo de atualização (catch-up) para sincronizar seu estado diretamente a partir de outras réplicas, eliminando assim a necessidade de executar todas as operações perdidas.

O gerenciamento de estado requer métodos como registro (logging), checkpointing e sincronização de estado. As réplicas mantêm um log das operações executadas, permitindo restaurar um estado consistente por meio da reexecução do log obtido de outras réplicas. Para evitar crescimento ilimitado do log, as réplicas criam periodicamente um checkpoint e truncam o log, descartando operações ocorridas antes do checkpoint. Para atualizar seu estado, uma réplica instala um checkpoint válido e processa comandos registrados obtidos de outras réplicas após o checkpoint. Essas funcionalidades são expostas por meio de uma interface que permite às aplicações gerenciar o estado criando checkpoints (ou snapshots) e/ou restaurando o estado da réplica. Normalmente, desenvolvedores interagem com essas capacidades por meio de APIs de bibliotecas SMR, que simplificam essas operações. Por exemplo, na biblioteca BFT-SMaRt [11, 12], o gerenciamento de estado da replicação faz parte de sua arquitetura modular. As aplicações devem implementar uma interface para salvar e carregar o estado, que é utilizada pelo módulo de gerenciamento de estado para gerar periodicamente snapshots que capturam o estado atual da aplicação.

Garantir transferência de estado consistente e correta na presença de falhas Bizantinas é desafiador, pois algumas réplicas podem enviar dados de estado incorretos, incompletos ou desatualizados. O BFT-SMaRt aborda esse problema utilizando validação baseada em quórum para garantir que apenas snapshots e logs de estado válidos sejam aplicados às réplicas que executam a sincronização de estado. Nesse processo, uma réplica r que executa a sincronização de estado solicita o estado a um quórum (tipicamente $f+1$ réplicas para f falhas Bizantinas). Uma réplica envia o snapshot de estado, enquanto outras fornecem porções do log e hashes criptográficos do mesmo snapshot para validação [11, 12]. A réplica r então verifica o hash do snapshot em relação aos enviados pelo quórum. Se os hashes não coincidirem, o estado é rejeitado, prevenindo adulteração maliciosa. Essa abordagem explora a garantia de que pelo menos $2f+1$ réplicas são não falhas, assegurando que a réplica r aplique apenas dados de estado

corretos, ignorando respostas maliciosas.

No entanto, essa abordagem possui uma limitação fundamental em cenários Bizantinos: a réplica r só consegue detectar um estado corrompido após baixá-lo completamente. Isso impacta significativamente o desempenho, pois downloads repetidos a partir de réplicas maliciosas desperdiçam largura de banda e prolongam o tempo de sincronização de estado. Além disso, como todo o estado é obtido de uma única réplica, a velocidade de transferência é limitada pela largura de banda e pela responsividade dessa réplica específica, amplificando ainda mais o custo de tentativas de sincronização malsucedidas.

Uma abordagem promissora, comumente empregada em blockchains (uma forma especializada de SMR), envolve integrar o estado da aplicação diretamente à biblioteca de replicação e representá-lo por meio de estruturas de dados clusterizadas e autovalidáveis. Esse método também alivia o programador da aplicação do ônus do gerenciamento de estado, ao fornecer uma interface direta para armazenar e manipular os dados que constituem o estado. Além disso, tais estruturas de dados clusterizadas fornecem um meio eficiente de organizar e validar dados, garantindo que, mesmo na presença de réplicas maliciosas ou falhas, o sistema consiga manter um estado consistente.

Capítulo 3

Comunicação em Sistemas SMR

Sistemas de Replicação por Máquina de Estados (State Machine Replication – SMR) dependem de primitivas de comunicação para garantir consistência entre réplicas na presença de falhas. Esses mecanismos de comunicação sustentam o requisito fundamental da SMR: que todas as réplicas corretas executem a mesma sequência de comandos na mesma ordem, apesar da assincronia e de falhas no sistema [66, 67, 90]. Este capítulo examina as principais abstrações de comunicação usadas em SMR, enfatizando seu papel em garantir ordenação e entrega de mensagens consistentes e eficientes, e constrói sobre esses fundamentos para introduzir novos protocolos de comunicação que abordam limitações existentes.

Começamos discutindo *Atomic Broadcast*, a abstração clássica que garante a entrega de mensagens em ordem total para todos os participantes. Embora o atomic broadcast seja suficiente para muitos cenários de replicação, ele impõe sobrecarga desnecessária em sistemas nos quais comandos precisam ser entregues apenas a subconjuntos de réplicas. Para tratar esses cenários, *Atomic Multicast* estende a abstração de broadcast ao permitir que mensagens sejam enviadas a grupos específicos de processos, mantendo ordenação consistente entre grupos sobrepostos. Descrevemos algoritmos fundamentais que implementam atomic multicast, incluindo o algoritmo de Skeen [13] e ByzCast [22], e apresentamos uma classificação de protocolos existentes de atomic multicast com base em suas abordagens de projeto e modelos de falha.

Em seguida, apresentamos FlexCast [9], nosso protocolo proposto e o primeiro atomic multicast genuíno baseado em overlay para sistemas SMR. A genuinidade captura a essência do atomic multicast no sentido de que apenas o remetente de uma mensagem e seus destinos coordenam para ordenar a mensagem, levando a protocolos eficientes [51]. Protocolos baseados em overlay restringem como grupos de processos podem se comunicar. Limitar a comunicação leva a protocolos mais simples e reduz a quantidade de informação que cada processo precisa manter sobre o

restante do sistema. O FlexCast implementa atomic multicast genuíno usando um overlay em grafo acíclico direcionado completo (C-DAG). Além disso, introduzimos uma nova propriedade, quiescência, que refina a propriedade de minimalidade para evitar comunicação desnecessária e garantir a integridade de protocolos de atomic multicast genuínos. Avaliamos experimentalmente o FlexCast em um ambiente geograficamente distribuído usando nosso gTPC-C proposto, uma variação do benchmark TPC-C que leva em conta distribuição geográfica e localidade. Mostramos que, ao explorar genuinidade e localidade da carga de trabalho, o FlexCast supera protocolos de atomic multicast bem estabelecidos sem a sobrecarga de comunicação inerente a protocolos multicast não genuínos de ponta.

Como continuidade do FlexCast, apresentamos seus mecanismos de reconfiguração, que estendem ainda mais os benefícios do atomic multicast genuíno baseado em overlay. O projeto do FlexCast não apenas garante menor sobrecarga de comunicação, como também permite que o sistema se adapte dinamicamente a mudanças na localidade da carga de trabalho por meio da reconfiguração do overlay. Modelamos o processo de reconfiguração como um problema de otimização e implementamos um protocolo para ajustar o overlay em resposta a mudanças de carga, aumentando a adequação do sistema a ambientes dinâmicos. Nossa avaliação experimental demonstra que, ao reconfigurar o overlay, nosso protocolo de reconfiguração pode reduzir substancialmente a latência para grupos de destino que representam a maior parte da carga de trabalho.

A implementação do FlexCast, incluindo sua extensão de reconfiguração, está publicamente disponível para apoiar reprodutibilidade e experimentação adicional. O código-fonte completo pode ser acessado por meio do repositório do autor no GitHub.[1]

3.1 Atomic broadcast

O atomic broadcast é melhor compreendido introduzindo primeiro primitivas de broadcast mais simples que fortalecem incrementalmente as garantias de comunicação. Essa progressão evidencia como propriedades de confiabilidade e ordenação culminam na ordem total fornecida pelo atomic broadcast. Discutimos como o atomic broadcast está intimamente ligado ao consenso, já que a capacidade de ordenar totalmente mensagens entre processos possibilita o acordo sobre um único valor de decisão em sistemas distribuídos.

[1] https://github.com/elbatista/flexcast

3.1.1 Primitivas básicas de broadcast

Abstrações de comunicação por broadcast são definidas por duas primitivas básicas:

- `broadcast(m)`: um processo envia a mensagem m para todos os processos do sistema.
- `deliver(m)`: um processo entrega a mensagem m.

Diferentes abstrações de comunicação constroem-se sobre essas primitivas ao fortalecer suas garantias [52].

Reliable broadcast Reliable broadcast garante que mensagens transmitidas por processos sejam eventualmente entregues por todos os processos corretos, sem duplicação ou fabricação. Ele fornece as seguintes propriedades:

- **Validade:** Se um processo correto realiza broadcast de uma mensagem m, então ele eventualmente entrega m.
- **Acordo:** Se um processo entrega uma mensagem m, então todos os processos corretos eventualmente entregam m.
- **Integridade:** Para qualquer mensagem m, todo processo entrega m no máximo uma vez, e somente se m tiver sido previamente enviada em broadcast por m.`sender`.

FIFO broadcast FIFO broadcast estende reliable broadcast garantindo que mensagens enviadas pelo mesmo remetente sejam entregues na ordem em que foram enviadas. Além das propriedades de reliable broadcast, ele garante:

- **Ordem FIFO:** Se um processo realiza broadcast da mensagem m antes de m', então nenhum processo entrega m' a menos que tenha entregue m.

Causal broadcast Causal broadcast fortalece FIFO broadcast preservando a causalidade entre mensagens, conforme definida pela relação happens-before [65]. Uma execução de uma primitiva de broadcast ou deliver por um processo é chamada de *evento*. Dizemos que um evento e precede causalmente um evento f, denotado $e \rightarrow f$, se: *(i)* um processo executa e e f nessa ordem; ou *(ii)* e é o broadcast da mensagem m e f é a entrega de m; ou *(iii)* existe um evento h tal que $e \rightarrow h$ e $h \rightarrow f$.

A abstração de causal broadcast garante:

- **Ordem causal:** Se o broadcast de uma mensagem m precede causalmente o broadcast de uma mensagem m', então nenhum processo entrega m' a menos que tenha previamente entregue m.

3.1.2 Definição de atomic broadcast

Atomic broadcast estende reliable broadcast ao fornecer uma ordenação total de mensagens, garantindo que todos os processos entreguem as mensagens na mesma ordem. Formalmente, atomic broadcast fornece:

- **Ordem total:** Se os processos p e q entregam ambas as mensagens m e m', então p entrega m antes de m' se, e somente se, q entrega m antes de m'.

Atomic broadcast garante que todos os processos corretos entreguem mensagens na mesma ordem, independentemente da ordem em que foram enviadas. Essa ordem total é crucial para garantir consistência em sistemas distribuídos, especialmente em replicação por máquina de estados. Atomic broadcast é tipicamente implementado sob a suposição de um modelo de sistema parcialmente síncrono, no qual o sistema pode inicialmente se comportar de forma assíncrona — com atrasos ilimitados de mensagens e velocidades de execução de processos — mas, após algum Tempo Global de Estabilização (GST) desconhecido, passam a existir limites para comunicação e processamento.

3.2 Atomic multicast

Atomic multicast é uma abstração de comunicação chave em sistemas distribuídos confiáveis, garantindo que mensagens enviadas a múltiplos grupos sejam entregues de forma consistente. Atomic broadcast pode ser visto como um caso especial de atomic multicast no qual toda mensagem é multicastada para todos os processos do sistema. Portanto, atomic broadcast pode ser usado de forma trivial para implementar atomic multicast ao realizar broadcast de toda mensagem para o sistema e restringir sua entrega aos destinatários pretendidos. No entanto, tal solução não tem interesse prático devido à ineficiência.

Assim, caracterizamos agora protocolos de atomic multicast com base em três aspectos: genuinidade e quiescência (§3.2.1), tolerância a falhas (§3.2.2), e comunicação entre processos (§3.2.3). Concluímos esta seção classificando protocolos de atomic multicast com base nesses aspectos (§3.2.4).

3.2.1 Atomic multicast genuíno e quiescente

Atomic multicast também pode ser implementado delegando a responsabilidade de ordenar mensagens a um grupo distinto de processos. Toda mensagem m multicastada atomicamente é primeiro enviada ao grupo distinto, que ordena m em relação a outras mensagens multicast, e então propaga m para seus grupos de destino. Embora simples, essa solução não captura o espírito do atomic multicast.

Em vez disso, estamos interessados em protocolos de atomic multicast que sejam "genuínos". Em um protocolo de atomic multicast genuíno, apenas o remetente e os destinos de uma mensagem coordenam para ordenar a mensagem. Um protocolo de atomic multicast genuíno não depende de um grupo fixo de processos e não envolve processos desnecessariamente. Mais precisamente, um algoritmo de atomic multicast genuíno deve garantir a seguinte propriedade [51].

- **Minimalidade**: Se um processo p envia ou recebe uma mensagem em uma execução R, então alguma mensagem m é multicastada em R, e p é $m.\texttt{sender}$ ou pertence a um grupo em $m.\texttt{dst}$.

A minimalidade restringe a comunicação aos processos envolvidos no multicast de uma requisição como remetente ou destino da requisição. Uma vez que um processo é remetente ou destino de uma requisição, no entanto, ele pode se comunicar livremente com qualquer outro processo sem violar a minimalidade. Esse comportamento obviamente contradiz o espírito de minimalidade e genuinidade. Evitamos tais comportamentos indesejados introduzindo a seguinte propriedade adicional.

- **Quiescência**: Se existe um tempo t_1 em uma execução R após o qual nenhuma requisição r é multicastada tal que p seja $r.\texttt{sender}$ ou um destino em $r.\texttt{dst}$, então existe um tempo $t_2 \geq t_1$ tal que, após t_2, p não envia nem recebe quaisquer mensagens.

Note que minimalidade e quiescência são complementares: enquanto a minimalidade define quando processos podem trocar mensagens, a quiescência estabelece quando processos devem parar de se comunicar. Nosso protocolo FlexCast proposto (§3.3) é um protocolo de atomic multicast genuíno que satisfaz ambas as propriedades, como demonstramos em §3.3.3.

3.2.2 Atomic multicast tolerante a falhas

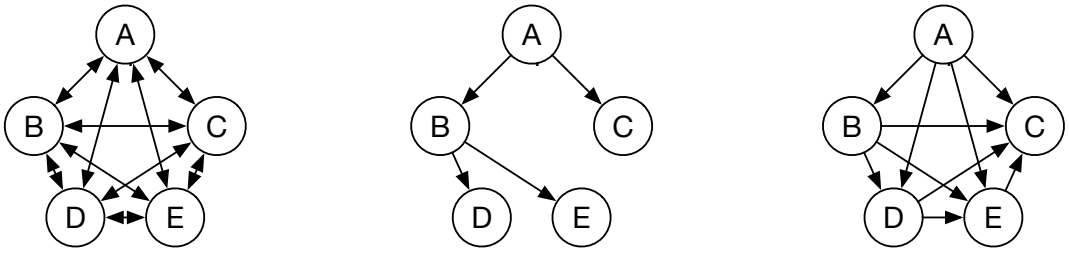
O primeiro protocolo de atomic multicast é atribuído a D. Skeen e é conhecido como "protocolo de Skeen" [13]. O protocolo é genuíno, mas não tolera falhas. Uma mensagem m multicastada é primeiro propagada para todos os seus destinos. Ao receber a mensagem, um destino atribui à mensagem um timestamp local e envia o timestamp para os demais destinos da mensagem. Quando um destino recebe todos os timestamps locais da mensagem, ele calcula o timestamp final da mensagem como o máximo entre todos os timestamps locais da mensagem. Uma mensagem só é entregue após possuir um timestamp final, e mensagens são entregues na ordem de seus timestamps finais.

Diversos protocolos de atomic multicast tolerantes a falhas baseiam-se na técnica de ordenação do Skeen de troca de timestamps (por exemplo, [23, 39, 51, 86]). Nesses protocolos de atomic multicast tolerantes a falhas, mensagens são endereçadas a conjuntos de grupos de processos, em vez de a um conjunto de processos, como no protocolo original de Skeen. Embora alguns processos em um grupo possam falhar, cada grupo atua como uma entidade confiável, cuja lógica é replicada dentro do grupo usando replicação por máquina de estados [90].

3.2.3 Atomic multicast baseado em overlay

A maioria dos protocolos de atomic multicast assume que processos podem se comunicar diretamente entre si. Em outras palavras, o *grafo de comunicação*, um grafo direcionado no qual vértices representam grupos de processos servidores replicados e uma aresta do grupo g para o grupo h significa que g pode enviar mensagens para h, é completo. Uma abordagem alternativa é restringir a comunicação por meio de um overlay que determina como grupos podem se comunicar.

Restringir a comunicação pode levar a algoritmos de atomic multicast mais simples, como em um overlay em árvore. Além disso, se a comunicação precisa ser autenticada, como em protocolos tolerantes a falhas Bizantinas, um grafo de comunicação parcialmente conectado requer menos chaves para serem mantidas e trocadas entre processos. Por fim, um grafo de comunicação completo é uma suposição razoável em sistemas que executam dentro do mesmo domínio administrativo (por exemplo, o Spanner do Google [26]). Em alguns contextos, no entanto (por exemplo, sistemas descentralizados), múltiplas entidades de diferentes domínios administrativos colaboram, mas não desejam estabelecer conexões com todos os demais domínios.



(a) Complete graph (b) Tree (c) Complete DAG

Figure 3.1. Três grafos de comunicação usados em algoritmos de atomic multicast envolvendo grupos $A, B, ..., E$: (a) um grafo completo (a abordagem mais comum), (b) uma árvore, e (c) um grafo acíclico direcionado completo, ou C-DAG (a abordagem que propomos mais adiante). Nos grafos, a aresta direcionada $g \rightarrow h$ significa que o grupo g pode enviar mensagens ao grupo h, e h pode receber mensagens de g mas não enviar mensagens a g.

A Figura 3.1 mostra três casos de interesse. Todos os algoritmos de atomic multicast genuínos de que temos conhecimento assumem um grafo de comunicação completo (Figura 3.1 (a)). Uma árvore (Figura 3.1 (b)) é o overlay mínimo necessário para que qualquer protocolo de atomic multicast suporte uma carga de trabalho arbitrária (isto é, mensagens podem ser multicastadas para qualquer conjunto de grupos), pois remover uma aresta do overlay resulta em um grafo particionado. O ByzCast [22] é um protocolo de atomic multicast que utiliza um overlay em árvore. Para ordenar uma mensagem m, m é primeiro ordenada dentro do grupo de menor ancestral comum (`lca`) entre os grupos em m.`dst`, no pior caso a raiz da árvore de overlay. Em seguida, m é sucessivamente ordenada pelos grupos inferiores na árvore até alcançar todos os grupos em m.`dst`. O principal invariante do ByzCast é que os grupos inferiores na árvore preservam a ordem induzida por grupos superiores. Embora simples, o ByzCast não é genuíno, pois uma mensagem pode precisar ser ordenada por um grupo que não está no conjunto de destinos da mensagem. Por exemplo, na Figura 3.1 (b), uma mensagem multicastada para os grupos B e C será primeiro ordenada em A e então propagada e ordenada por B e C.

C-DAG: uma discussão sobre genuinidade Para motivar a abordagem apresentada mais adiante, argumentamos informalmente que um grafo acíclico direcionado completo (C-DAG) é uma condição necessária para genuinidade em sistemas parcialmente síncronos baseados em overlay. Para entender por quê, observe que, em um C-DAG, cada vértice possui uma aresta de entrada ou de saída com qualquer outro vértice. Se não há aresta entre os grupos g e h no grafo, então uma mensagem endereçada a g e h precisa envolver um ou mais grupos que não fazem parte do conjunto de destinos da mensagem, o que viola a minimalidade. [2]

De forma semelhante a um overlay em árvore, em um C-DAG uma mensagem multicastada pode ser inicialmente ordenada no grupo de menor ancestral comum (`lca`) g entre os destinos da mensagem e então ser propagada por g aos demais destinos. Em um C-DAG, no entanto, o `lca` de uma mensagem pode propagar a mensagem diretamente a todos os destinos restantes da mensagem, respeitando a minimalidade. Diferentemente de um overlay em árvore, em um C-DAG um grupo não pode entregar uma mensagem imediatamente após recebê-la de seu ancestral, pois grupos podem ter mais de uma aresta de entrada e mensagens podem chegar a um grupo a partir de diferentes ancestrais. Um grupo com múltiplas arestas de entrada deve determinar a ordem correta para entregar mensagens vindas de seus ancestrais.

[2] Esta observação assume que, para ordenar uma mensagem endereçada a dois ou mais grupos, esses grupos precisam se comunicar. É fácil ver que isso vale em um sistema parcialmente síncrono: se g e h não se comunicam, então eles não conseguem concordar sobre a ordem de mensagens endereçadas tanto a g quanto a h.

Por exemplo, na Figura 3.1 (c), o grupo C pode receber mensagens de A e B. Em uma execução na qual a mensagem m_1 é endereçada aos grupos A, B e C, e a mensagem m_2 é endereçada aos grupos B e C, o grupo C receberá m_1 de A e m_2 de B. Se B entrega m_2 antes de m_1, então C deve respeitar essa ordem, mesmo que possa receber m_1 antes de m_2. Caso contrário, C violaria a propriedade de ordem acíclica do atomic multicast.

3.2.4 Uma classificação de protocolos de atomic multicast

Como observado anteriormente, muitos protocolos de atomic multicast se baseiam na técnica de ordenação do Skeen, estendendo-a para prover tolerância a falhas [23], [39], [50], [69], [86]. Como, em todos esses protocolos, processos se comunicam diretamente uns com os outros, referimo-nos a essa classe de protocolos como protocolos de atomic multicast *distribuídos* (ver Tabela 3.1).

Outra classe de protocolos baseia-se na suposição de que processos se comunicam por meio de um grupo distinto de processos, que ordena mensagens e as propaga aos seus destinos (por exemplo, [22, 42]). Nessa classe, um invariante importante é que grupos inferiores na árvore preservam a ordem induzida por grupos superiores. Daqui em diante, referimo-nos a essa classe de protocolos como protocolos de atomic multicast *hierárquicos*.

Classe	**Tipo**	**Exemplos**
Distribuido	genuino	[13, 23, 30, 39, 50, 69, 86]
Hierarquico	nao-genuino	[22, 42, 63]
C-DAG overlay	genuino	FlexCast [9] (nossa abordagem)

Table 3.1. Diferentes classes de protocolos de atomic multicast.

Por fim, nossa abordagem proposta FlexCast é um protocolo de atomic multicast genuíno que utiliza um overlay em grafo acíclico direcionado completo (C-DAG), em vez de uma árvore como em todos os demais protocolos conhecidos da classe *hierárquica*. No FlexCast, uma mensagem multicastada é inicialmente ordenada no grupo de menor ancestral comum entre os grupos em m.`dst` e então propagada pelo `lca` aos destinos restantes de m. Diferentemente de outros protocolos hierárquicos, no FlexCast o `lca` de uma mensagem pode propagar a mensagem diretamente a todos os destinos restantes da mensagem, respeitando a minimalidade. Além disso, no FlexCast, um grupo com múltiplas arestas de entrada deve determinar a ordem correta para entregar mensagens vindas de seus ancestrais, o que requer mecanismos sofisticados. Na próxima seção, apresentamos tais mecanismos em mais detalhes.

3.3 FlexCast: atomic multicast genuíno baseado em overlay

Como discutido nas seções anteriores, um protocolo de atomic multicast genuíno garante que apenas o emissor e os destinos da mensagem se comuniquem para ordenar uma mensagem multicast [51]. Em cenários geograficamente distribuídos, um protocolo de atomic multicast genuíno consegue explorar melhor a localidade do que um protocolo não genuíno, pois mensagens endereçadas a grupos próximos não introduzem comunicação com grupos remotos. Além disso, como um grupo só recebe mensagens que lhe são endereçadas, em um protocolo de atomic multicast genuíno os grupos não incorrerem em sobrecarga de comunicação ao repassar mensagens para os destinos. Isso é importante em ambientes geograficamente distribuídos, onde a comunicação por enlaces de longa distância representa um custo relevante (por exemplo, Amazon Web Services).

A maioria dos protocolos de atomic multicast assume que os processos podem se comunicar diretamente entre si. Alternativamente, os processos se comunicam seguindo um *overlay*, que determina quais processos podem trocar mensagens com quais outros processos. Impor limites à comunicação traz vantagens. Por exemplo, overlays podem representar a estrutura de domínios administrativos, simplificar o projeto de protocolos e reduzir a quantidade de informação que cada processo precisa manter sobre o restante do sistema (por exemplo, gerenciamento de chaves em protocolos tolerantes a falhas Bizantinas [22]).

No entanto, combinar genuinidade e overlays é desafiador. Os protocolos existentes de atomic multicast focam em um aspecto ou no outro, mas não em ambos. Por exemplo, todos os protocolos genuínos existentes de atomic multicast assumem um overlay totalmente conectado. Protocolos hierárquicos, que estruturam a comunicação entre grupos como uma árvore, não são genuínos. Por exemplo, em ByzCast [22], uma mensagem multicast é primeiro enviada ao menor ancestral comum dos destinos da mensagem e, então, segue descendo pela árvore até alcançar todos os destinos. A lógica de ByzCast é simples e os processos em um grupo precisam manter informação apenas sobre seu pai e seus filhos. Entretanto, ele não é genuíno, pois mensagens endereçadas aos filhos do grupo g, mas não a g, são primeiro enviadas a g e depois propagadas aos filhos de g, violando a genuinidade.

A Figura 3.2 quantifica a sobrecarga de comunicação de ByzCast, calculada como um menos a razão entre o número de mensagens que um grupo entrega (isto é, mensagens endereçadas ao grupo) e o número de mensagens que o grupo recebe como parte da comunicação imposta pelo overlay em árvore, e expressa como uma porcentagem. Os dados deste experimento foram coletados durante a execução do benchmark

gTPC-C com a árvore T_1 e 90% de localidade (mais detalhes na Seção 3.3.5). Em média, os grupos incorrem em quase 10% de sobrecarga de comunicação. Alguns grupos, no entanto, são mais penalizados do que outros, dependendo de sua posição na árvore. Em particular, cerca de 23% e 36% da comunicação dos grupos 5 e 9, respectivamente, é sobrecarga. Isso contrasta com protocolos genuínos de atomic multicast, que não têm sobrecarga de comunicação.

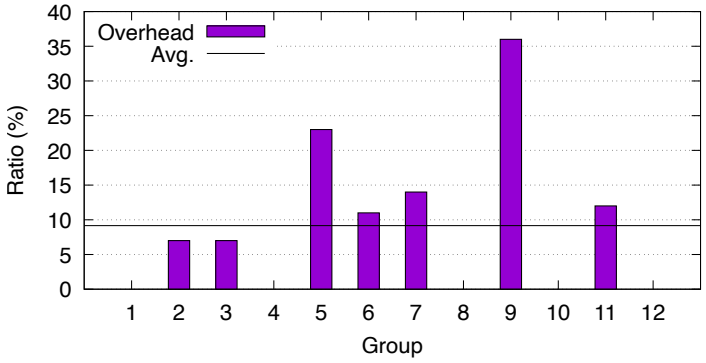


Figure 3.2. Sobrecarga de comunicação em um protocolo hierárquico, expressa como uma porcentagem, calculada para cada grupo como 1 menos a razão entre o número de mensagens entregues e o número de mensagens recebidas pelo grupo.

Portanto, propomos FlexCast, o primeiro protocolo de atomic multicast genuíno baseado em overlay. FlexCast assume um overlay em grafo acíclico direcionado completo (C-DAG). Mensagens multicast são enviadas ao menor ancestral comum (`lca`) dos destinos da mensagem. O `lca` então propaga a mensagem para todos os demais destinos em um único passo de comunicação, sem envolver quaisquer grupos que não sejam destino da mensagem. FlexCast utiliza um protocolo sofisticado, baseado em histórico, para ordenar mensagens. Primeiro, cada processo constrói um histórico com todas as mensagens que o processo entregou. Esse histórico é propagado para outros processos no C-DAG, de modo que os processos possam garantir consistência (por exemplo, que dois processos não ordenem duas mensagens de maneira diferente).

Entretanto, simplesmente seguir os históricos de outros processos não é suficiente para garantir uma ordem consistente devido a dependências indiretas. Dependências indiretas acontecem por alguns motivos. Por exemplo, se o processo x ordena a mensagem m_1 antes da mensagem m_2 e o processo y ordena m_2 antes da mensagem m_3, então o processo z precisa ordenar m_1 antes de m_3 como consequência das dependências criadas pelos processos x e y envolvendo m_2, uma mensagem não endereçada a z. FlexCast é bem adequado para equipar sistemas geograficamente replicados, pois explora a localidade.

No restante desta seção, descrevemos a ideia central por trás de FlexCast, fornecemos algoritmos detalhados, uma prova de correção, considerações práticas de implementação e uma avaliação abrangente do nosso protocolo em comparação com abordagens de ponta.

3.3.1 Ideia geral

Os grupos em FlexCast são estruturados como um grafo acíclico direcionado completo (C-DAG), como no exemplo da Figura 3.1 (c). Assumimos que existe uma ordem total entre os grupos. Cada grupo recebe um posto (rank) único em $0..(n-1)$, onde n é o número de grupos. A topologia do C-DAG é tal que há uma aresta direcionada de cada grupo com posto i para cada grupo com posto j se $i < j$. Nesse grafo, os *ancestrais* de i têm posto menor do que i e os *descendentes* de i têm posto maior do que i.[3] A Figura 3.1 (c) mostra um C-DAG com nós ordenados do mais baixo para o mais alto como: A, B, D, E, C.

Um *processo cliente* é uma entidade externa que interage com o sistema submetendo requisições a serem multicast para um ou mais grupos. Clientes não pertencem a nenhum grupo no C-DAG; em vez disso, eles usam a estrutura de grupos para disseminar suas mensagens de forma consistente. Um cliente realiza atomic multicast de uma mensagem m enviando m ao menor ancestral comum (`lca`) de m. O `lca` de uma mensagem multicast é o grupo com o menor posto entre os destinos da mensagem. No seu `lca`, m é diretamente entregue e propagada para os demais grupos destino de m (por definição, o `lca` tem arestas diretas para cada outro grupo destino em m.`dst`). De forma semelhante a um atomic multicast baseado em árvore, em um C-DAG um grupo precisa respeitar as dependências criadas por seus ancestrais e propagar dependências para seus descendentes. Em um C-DAG, porém, um grupo pode ter múltiplos ancestrais e dependências podem ser criadas por qualquer um deles. Um desafio importante é garantir que as dependências sejam comunicadas adequadamente ao longo do C-DAG sem violar a propriedade de minimalidade do atomic multicast genuíno. FlexCast usa três estratégias para alcançar isso, como explicado a seguir.

Estratégia (a): Primeiro, cada grupo acompanha um *histórico*, um grafo em que mensagens são vértices e suas ordens relativas são arestas. Um vértice contém o id e os destinos de uma mensagem. Mensagens entregues em um grupo são registradas em seu histórico e constroem uma ordem total dentro do grafo. Quando um grupo propaga uma mensagem para outro, seu histórico é incluído. O grupo destino estende

[3] Usamos os termos "grupos mais baixos" e "grupos mais altos" para denotar posições relativas nessa ordenação por posto, e "grupo mais baixo" e "grupo mais alto" de um subconjunto de grupos, também referindo-se a essa ordenação. "Ancestrais" de um grupo g denotam o conjunto de grupos abaixo de g, enquanto "descendentes" denotam, respectivamente, os grupos acima.

seu histórico com os históricos que recebe de outros grupos e com as mensagens que entrega. O histórico então torna-se um grafo. Mais especificamente, como a ordenação é respeitada (discutido a seguir), o histórico é um DAG. Grupos destino usam o histórico para garantir que as mensagens sejam entregues de forma consistente em todo o sistema.

Para entender a necessidade de trocar históricos, considere o cenário ilustrado na Figura 3.3 (a), em que o grupo A é o `lca` das mensagens m_1 (multicast para A e C) e m_2 (multicast para A e B), e o grupo B é o `lca` de m_3 (multicast para B e C). Como A entrega m_1 antes de m_2 (isto é, $m_1 \prec m_2$) e B entrega m_2 antes de m_3 (isto é, $m_2 \prec m_3$), C deve entregar m_1 antes de m_3 para evitar um ciclo entre mensagens entregues. Mas C recebe m_3 de B antes de receber m_1 de A. Ao receber o histórico de B, C sabe que deve entregar m_1 e depois m_3 para evitar ciclos.

Infelizmente, incluir históricos em mensagens encaminhadas não é suficiente para garantir uma ordem consistente. Intuitivamente, isso ocorre porque nem todas as dependências são capturadas na comunicação de mensagens de aplicação entre grupos. Há dois casos a considerar, dependendo de o grupo que cria a dependência estar ciente de que deve propagar a dependência a seus descendentes ou não.

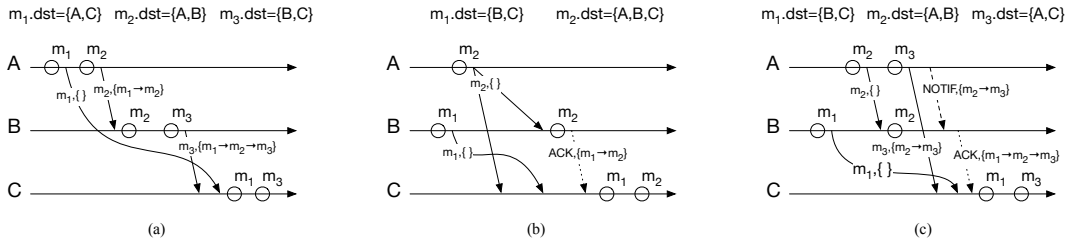


Figure 3.3. Execuções de FlexCast ilustrando o uso de (a) históricos, (b) mensagens ack, e (c) mensagens notif em um overlay onde $A \rightarrow B$,$A \rightarrow C$ e $B \rightarrow C$. (Legenda: uma seta cheia é a propagação de uma mensagem de aplicação, um círculo é a entrega de uma mensagem, uma seta pontilhada é uma mensagem ack, e uma seta tracejada é uma mensagem notif).

Estratégia (b): Para motivar o caso em que um grupo está ciente de que deve enviar dependências a seus descendentes, considere a execução na Figura 3.3 (b). Nesse caso, B entrega m_1 antes de m_2, e C recebe m_2 de A (com um histórico vazio) e então recebe m_1 de B (com um histórico vazio, já que B não sabia sobre m_2 quando enviou m_1 a C). Ainda assim, C deve entregar m_1 antes de m_2. FlexCast garante a ordem adequada nesses casos da seguinte forma. Se o grupo g e seu descendente h estão no destino de uma mensagem m e g não é o `lca` de m, então g envia uma mensagem ACK a h com o histórico de g. Por outro lado, se h recebe uma mensagem m e h tem um ancestral que está no destino de m, mas não é o `lca` de m, então h espera pela mensagem ACK de g.

Estratégia (c): Para motivar o caso em que um grupo não está ciente de que deve enviar dependências a seus descendentes, considere a execução na Figura 3.3 (c). Nesse caso, o grupo A envia m_3 e seu histórico (isto é, m_2 precede m_3) para C, e B envia m_1 e um histórico vazio para C (isto é, porque a dependência entre m_1 e m_2 ocorre em B após B se comunicar com C). B não envia a C a informação de que m_1 precede m_2 porque m_2 não é endereçada a C. Ainda assim, C deve entregar m_1 antes de m_3. Para lidar com esse caso, quando um grupo determina que um descendente d deve encaminhar seu histórico ao longo do C-DAG, ele envia uma mensagem NOTIF a d para que d possa comunicar suas dependências a outros grupos.

Mais precisamente, quando um grupo g, o `lca` de uma mensagem (ou outro destino em m.`dst`) está prestes a encaminhar a mensagem m (respectivamente, uma mensagem ACK referente a m) e existe um grupo h tal que: (i) h não está em m.`dst`; (ii) h é descendente de g e ancestral de um grupo r em m.`dst`; e (iii) existe uma mensagem no histórico de g endereçada a h, então g envia uma mensagem NOTIF referente a m para h. Se o grupo h recebe uma mensagem NOTIF referente a m, ele envia mensagens ACK a todos os seus descendentes $k \in m$.`dst`. Além disso, indutivamente, se houver uma mensagem endereçada ao grupo h' no histórico de h com as mesmas restrições acima, h notifica h'. Essa indução termina naturalmente, pois há uma ordem total sobre os grupos.

Por que é genuíno Para argumentar que FlexCast é genuíno, primeiro observe os seguintes aspectos discutidos nas *Estratégias (a)* e *(b)*:

- quando m é multicast, ela entra no overlay em m.`lca`() (ver Algoritmo 1), que por definição é um destino de m;
- m.`lca`() propaga m para seus demais destinos em m.`dst`; e
- cada destino d (exceto m.`lca`()) envia mensagens ACK para grupos em m.`dst` acima de d.

Do acima, segue que a comunicação descrita envolve exclusivamente grupos em m.`dst`.

Agora, considere a *Estratégia (c)* e observe que:

- um grupo $g \in m$.`dst` pode enviar uma mensagem NOTIF para um grupo $h \notin m$.`dst` desde que g tenha enviado anteriormente uma mensagem para h, isto é, alguma mensagem foi multicast para h na execução R; e
- indutivamente, h notifica h' apenas se alguma mensagem foi multicast de h para h' na execução R.

Do acima, segue que grupos fora de $m.\texttt{dst}$ trocam mensagens apenas se eles já haviam se comunicado na execução R, preservando minimalidade (ver definição na Seção 3.2.1).

3.3.2 Protocolo detalhado

O Algoritmo 1 apresenta as estruturas de dados básicas usadas em FlexCast. Cada grupo conhece a topologia do C-DAG e possui um canal de comunicação para cada grupo descendente (isto é, um enlace ponto-a-ponto confiável FIFO). Como consequência, cada processo possui uma fila de entrada para cada canal de entrada proveniente de grupos ancestrais. Cada fila contém mensagens ainda não entregues enviadas pelos respectivos ancestrais.

Algorithm 1 Tipos e estruturas de dados, para cada grupo g

1: **Tipo** $Message$: toda mensagem m possui:
2: $m.id$ {*m's id global único*}
3: $m.dst$ {*destinos de m, um subconjunto de grupos*}
4: $m.payload$ {*fornecido pela aplicação*}
5: $m.acks \leftarrow \emptyset$ {*um conjunto de acks recebidos*}
6: $m.notifList \leftarrow \emptyset$ {*um conjunto de grupos notificados*}
7: $m.lca() : func$ {*retorna o lca em* $m.dst$}
8: **Tipo (histórico)** H: {*um histórico é* }
9: $H = (M, D, lastDlvd)$ {*mensagens, dependências, a última*}
10: M : conjunto de *Message* {*um par* $(m_1, m_2) \in D$ *significa ...*}
11: $D : M \times M$ {m_1 *ordenada antes de* m_2*:* m_2 *depende de* m_1}
12: $lastDlvd : M \cup \{\perp\}$ {*a última mensagem entregue*}
13: **Variáveis do grupo** g**:**
14: $queues \leftarrow [\emptyset, \ldots, \emptyset]$ {*uma fila vazia por ancestral*}
15: $hst \leftarrow H(\emptyset, \emptyset, \perp)$ {*o histórico inicial do grupo g*}
16: $deliveredInG \subseteq hst.M$ {*as mensagens em hst entregues em g*}
17: $pendNotif \leftarrow \emptyset$ {*um conjunto de notificações pendentes*}
18: $\forall\ h$ acima de $g, hst(h) \leftarrow H(\emptyset, \emptyset, \perp)$ {*o histórico de g informado a cada h até o momento*}

Uma mensagem tem um id único, um conjunto de grupos destino e um payload arbitrário, fornecido pela aplicação. O protocolo armazena mensagens pendentes juntamente com um conjunto de respectivas mensagens ACK e um conjunto de grupos notificados, ambos detalhados mais adiante. A função $m.\texttt{lca}()$ retorna o grupo mais baixo em $m.\texttt{dst}$.

Um grupo g tem o histórico que aprende de cada um de seus ancestrais e as mensagens que entrega. O conjunto de mensagens entregues em g é um subconjunto das mensagens no histórico. O histórico constrói um DAG com dependências em $hst.D$. Como mensagens de notificação podem não ser entregues imediatamente de acordo

com critérios a serem detalhados mais adiante, um grupo também possui um conjunto de mensagens de notificação pendentes.

Quando o grupo g se comunica com um grupo descendente h, g informa apenas a diferença no histórico de g em relação à última mensagem que g enviou a h. Portanto, para cada descendente h, g mantém o controle de qual parte do seu histórico ele já enviou a h.

Para realizar atomic multicast de uma mensagem m, um cliente envia m a $m.\texttt{lca}()$. O Algoritmo 2 apresenta os eventos acionados em um grupo ao receber cada um dos três tipos de mensagens em nosso protocolo: (i) MSG é uma mensagem de cliente; (ii) ACK é uma mensagem de reconhecimento; e (iii) NOTIF é uma mensagem de notificação.

Algorithm 2 Eventos, para cada grupo g

```
1:  ao receber [MSG, m, history] ∧ g = m.lca()
2:      a-deliver(m)                                   {o lca pode entregar m imediatamente}
3:  ao receber [MSG, m, history] ∧ g ≠ m.lca()
4:      update-hst(history)                            {atualiza o histórico local com o histórico recebido}
5:      queues[m.lca()].enqueue(m)                     {enfileira m na fila do lca }
6:      reprocess-queues()                             {reprocessa todas as filas de ancestrais}
7:  ao receber [ACK, m, history] do ancestral a        {ao receber uma mensagem ACK }
8:      update-hst(history)                            {atualiza o histórico local}
9:      queues[m.lca()].get(m.id).acks.add([ACK de a]) {adiciona o ACK de a a m}
10:     queues[m.lca()].get(m.id).notifList.merge(m.notifList)  {atualiza a lista de notificação de m}
11:     reprocess-queues()                             {reprocessa todas as filas}
12: ao receber [NOTIF, m, history]                     {ao receber uma mensagem NOTIF }
13:     update-hst(history)                            {atualiza o histórico local}
14:     deps ← open-dependencies()                     {calcula possíveis dependências}
15:     if deps ≠ ∅ then
16:         pendNotif.add([NOTIF, m, deps])            {se dependências forem encontradas, adiciona NOTIF ao
            conjunto pendente}
17:     else
18:         send-descendants(m, ACK)                   {caso contrário, envia ACK a todos os descendentes em m.dst}
```

Em FlexCast, o `lca` entrega uma mensagem multicast assim que recebe a mensagem. Assim, o `lca` impõe sua ordem de entrega a todos os seus grupos descendentes por meio de informações disseminadas em históricos e mensagens auxiliares. No Algoritmo 2, ao receber uma MSG m, se g é o `lca`, ele pode entregar m imediatamente. Quando grupos que não são `lca` recebem uma MSG, eles primeiro atualizam seu histórico local com o histórico recebido junto com m, enfileiram m na fila do ancestral correspondente e reprocessam todas as filas de ancestrais, pois essa mensagem pode carregar a informação necessária para entregar outras mensagens.

Ao receber uma mensagem ACK, g atualiza seu histórico local e associa o ACK à MSG m correspondente na fila do `lca` que originou o ACK. Como um ACK pode identificar grupos adicionais a serem notificados, a lista de grupos notificados da mensagem é atualizada de acordo. O grupo g então reprocessa todas as filas.

Ao receber uma mensagem NOTIF, g atualiza seu histórico local, envia as mensagens ACK necessárias e possivelmente envia mensagens de notificação para seus descendentes também, como detalhado mais adiante. No entanto, se o histórico local contém uma mensagem m' endereçada a g que ainda não foi entregue, então g espera até entregar m' antes de enviar as mensagens ACK, e adiciona a NOTIF ao conjunto de notificações pendentes, evitando propagar dependências incompletas.

Algorithm 3 Funções principais, para cada grupo g

```
1:  a-deliver (m : Message)                          {quando uma mensagem m pode ser entregue}
2:      hst-add(m)                                   {adiciona m ao histórico local}
3:      if g = m.lca() then                          {se g é o lca de m }
4:          send-descendants(m, MSG)                 {envia m a todos os descendentes em m.dst}
5:      else
6:          queues[m.lca()].dequeue()                {remove m da fila do lca }
7:          send-descendants(m, ACK)                 {possivelmente envia ACK a todos os descendentes em m.dst}
8:      if ∃[NOTIF, n, deps] ∈ pendNotif | m ∈ deps then{verifica se desbloqueia alguma NOTIF pendente}
9:          deps ← deps \ m                          {remove m das dependências}
10:         if deps = ∅ then                         {se não há mais dependências, envia ACK }
11:             pendNotif ← pendNotif \[NOTIF, n, deps]   {remove NOTIF do conjunto pendente}
12:             send-descendants(n, ACK)             {envia ACK aos descendentes correspondentes}
13: update-hst (ah : H)                              {histórico do ancestral ah}
14:     hst.M ← hst.M ∪ ah.M                         {mensagens e dependências são}
15:     hst.D ← hst.M ∪ ah.D                         {integradas ao hst do grupo}
16: reprocess-queues ()
17:     faça:                                        {enquanto mensagens forem entregues}
18:         delivered ← false
19:         for all q ∈ queues do                    {itera sobre todas as filas q}
20:             if can-deliver(q.head()) then        {verifica condições para entregar o topo de q}
21:                 a-deliver(q.head())              {chama a função de entrega para o topo de q}
22:                 delivered ← true
23:     while delivered
24: open-dependencies (): conjunto de Messages       {calcula dependências}
25:     return {∀ m ∈ hst.M | g ∈ m.dst ∧ m ∉ deliveredInG }   {mensagens em hst ainda não
        entregues}
26: send-descendants (m : Message, mType ∈ {MSG, ACK})
27:     send-notifs(m)                               {envia possíveis mensagens NOTIF para descendentes}
28:     for all descendant d ∈ m.dst do
29:         send [mType, m, diff-hst(d)] to d        {envia MSG ou ACK aos descendentes com histórico
            diferencial}
```

Nos Algoritmos 3 e 4, apresentamos a lógica usada por cada grupo para entregar mensagens. A ordem total das mensagens entregues é construída fazendo com que a nova mensagem dependa da última mensagem entregue. Usamos o conjunto *delivered-InG* para identificar mensagens entregues em g, o que é um subconjunto de *hst.M* e é usado para identificar possíveis dependências abertas no histórico. Uma dependência aberta ocorre quando uma mensagem endereçada a g está incluída no histórico de g mas ainda não foi entregue. A operação *diff-hst* é uma otimização: apenas as novas partes de um histórico são enviadas a cada descendente. A operação *depend* computa a possível dependência transitiva de m em relação a m' em hst.

Algorithm 4 Funções auxiliares, para cada grupo g

```
1:  hst-add (m : Message)
2:      hst.M ← hst.M ∪ {m}                          {adiciona m, se ainda não estiver em hst}
3:      hst.D ← hst.D ∪ {(hst.lastDlvd, m)}                          {constrói ordem total em}
4:      hst.lastDlvd ← m                                              {msgs entregues neste grupo}
5:      deliverdInG ← deliverdInG ∪{m}
6:  diff-hst(h : um grupo acima) : H          {histórico de g ainda não informado a h até aqui}
7:      seja hstTmp.M ← hst.M \ hst(h).M
8:      seja hstTmp.D ← hst.D \ hst(h).D
9:      seja hstTmp.lastDlvd ← hst.lastDlvd
10:     hst(h) ← hst          {o histórico enviado a h é atualizado para o histórico atual de g}
11:     return hstTmp
12: depend (m, m' : Message): boolean                          {computa dependências transitivas}
13:     return (m', m) ∈ hst.D ∨ ∃m'' | (m', m'') ∈ hst.D ∧ depend(m, m'')
14: send-notifs (m : Message)                                  {envia NOTIF para grupos}
15:     for all descendant d | d ∉ m.dst do
16:         if ∃d' ∈ m.dst | d é ancestral de d' e hst.containsMsgTo(d) then
17:             send [NOTIF, m, diff-hst(d)] to d
18:             m.notifList.append(d)                          {m carrega os grupos notificados}
19: can-deliver (m : Message)
20:     if ancestors-to-ack(m) ⊈ ancestors-that-acked(m) then   {verifica se todos os ancestrais
        reconheceram}
21:         return false
22:     if ∃ m' ∈ hst.M | g ∈ m'.dst ∧ m' ∉ deliveredInG ∧
                          depend(m,m') then                    {verifica dependências}
23:         return false
24:     return true
25: ancestors-to-ack (m : Message): conjunto de grupos
26:     return (ancestrais de g em m.dst \ m.lca()) ∪
                queues[m.lca()].get(m.id).notifList   {ancestrais em m.dst e grupos notificados}
27: ancestors-that-acked (m : Message): conjunto de grupos
28:     return queues[m.lca()].get(m.id).acks
```

Quando uma mensagem pode ser entregue, o grupo adiciona a mensagem ao seu histórico local. Um grupo `lca` envia a mensagem aos seus descendentes, enquanto grupos que não são `lca` removem a mensagem da fila do ancestral e enviam as mensagens ACK correspondentes aos seus descendentes. Todos os grupos verificam se a entrega dessa mensagem pode desbloquear notificações pendentes. A função *send-descendants()* faz parte das *Estratégias (a)* e *(b)* discutidas na Seção 3.3.1. Para enviar MSG m (ou ACK m), o `lca` (ou um descendente), primeiro envia possíveis mensagens de notificação aos seus descendentes que não estão em m.`dst`. A função *send-notifs()* implementa a *Estratégia (c)*: ela procura mensagens passadas e avalia se notificações são necessárias, incluindo os grupos notificados na lista de notificação de m. Então, m é enviada a todos os demais destinos em m.`dst`, carregando a lista de grupos notificados junto com o histórico com a informação necessária para cada destino (*diff-hst*).

A função *reprocess-queues()* é chamada ao receber mensagens MSG e ACK (ver Algoritmo 2). Em ambos os casos, ela itera pelas filas dos ancestrais e tenta entregar mensagens. Ela continua iterando enquanto mensagens puderem ser entregues devido a informações de dependência atualizadas. A entrega de mensagens em grupos que não são `lca` é definida na função *can-deliver(m)*. A primeira condição verifica se g recebeu ACK de todos os ancestrais necessários: (i) todos os ancestrais (exceto o `lca`) em m.`dst`; (ii) todos os ancestrais (fora de m.`dst`) notificados sobre a mensagem m, que foram informados a g via MSG ou ACK. Lembre que um grupo notificado, além de enviar ACK, pode ainda notificar outros grupos. No Algoritmo 2, *notifList* acumula todos os ancestrais notificados que precisam reconhecer m. A lista de ancestrais que reconheceram é mantida em *ancestors-that-acked*. Tendo a informação completa sobre m, a segunda condição garante que qualquer mensagem m' que precede m e é endereçada a g já tenha sido entregue antes da entrega de m.

3.3.3 Prova de correção

Para garantir que FlexCast opere corretamente sob todas as condições definidas pelo nosso modelo de sistema, fornecemos uma prova de correção detalhada. Primeiro delineamos as principais suposições subjacentes ao protocolo, seguidas de argumentos formais demonstrando como ele satisfaz as propriedades de atomic multicast.

FlexCast assume que:

1. processos são organizados em grupos disjuntos e cada grupo é tolerante a falhas;

2. os grupos são totalmente ordenados e a topologia de comunicação possui canais FIFO de cada grupo para todos os grupos acima.

3. quando clientes enviam uma mensagem multicast m para grupos destino em $m.\texttt{dst}$, m é enviada ao menor grupo em $m.\texttt{dst}$, chamado de grupo menor ancestral comum (`lca`). Usamos $m.\texttt{lca}()$ para denotar o menor grupo em $m.\texttt{dst}$.

Na discussão a seguir, a comunicação é considerada no nível de grupos, e não de processos individuais. Assim, quando nos referimos a um grupo enviando, recebendo ou entregando uma mensagem, isso denota que a maioria de seus processos executou a ação correspondente.

Definition 1 *Ordem de mensagens: para qualquer par de mensagens $m \neq m'$, dizemos que $m < m'$ sse:*

- *m e m' são entregues em um grupo comum, e m é entregue antes de m';*
- *ou por transitividade: $m < m'' \wedge m'' < m' \implies m < m'$.*

Lemma 1 *Para qualquer mensagem m enviada via atomic multicast a múltiplos grupos, m é recebida em todos e apenas nos grupos destino $d \in m.dst$.*

PROVA: Pela Assunção 3, o cliente envia m a $m.\texttt{lca}()$. Pela Assunção 2, qualquer subconjunto $m.\texttt{dst}$ de destinos é alcançado diretamente por $m.\texttt{lca}()$. De acordo com o Algoritmo 3, linhas 3-4, quando $m.\texttt{lca}()$ recebe m, ele incondicionalmente executa $send\text{-}descendants(m)$ para todos os demais destinos em $m.\texttt{dst}$ e apenas para esses. Como grupos e canais são tolerantes a falhas, eventualmente todo grupo destino recebe m, e nenhum outro grupo a recebe. □

Lemma 2 *Sejam m e m' mensagens tais que $m.dst \cap m'.dst \neq \emptyset$. Existe um único grupo que atribui uma ordem relativa a m e m', a ser seguida por todos os grupos acima.*

PROVA: Pela Assunção 3, m e m' entram no overlay por seus respectivos `lca`s e, pelo Lema 1, ambas são recebidas em seus respectivos destinos. Como os grupos são totalmente ordenados (Assunção 2) e $m.dst \cap m'.dst \neq \emptyset$, na interseção existe um único menor grupo que trata tanto m quanto m'. Chamamos esse grupo de menor destino comum dessas mensagens, $lcd(m, m')$. Como os canais de mensagem são direcionados apenas para grupos acima, a ordem relativa de m e m' é atribuída em $lcd(m, m')$ e seguida nos grupos acima. □

Lemma 3 *Para qualquer mensagem m enviada via atomic multicast, a informação completa de dependências para entregar m é eventualmente recebida em cada grupo em $m.dst$. A informação completa de dependências para entregar m em um grupo g é a informação sobre qualquer mensagem m' entregue antes de m, isto é, $m' < m$ em cada grupo abaixo de g.*

PROVA: Pelos Algoritmos 2, 3 e 4:

1. cada grupo g mantém um histórico registrando a ordem das mensagens que entregou e, para cada mensagem m entregue, as mensagens anteriores m' entregues em grupos abaixo de g, tais que $m' < m$;

2. toda mensagem carrega o histórico do grupo emissor, o que enriquece o histórico de cada grupo receptor ao recebê-la;

3. cada grupo g em $m.\texttt{dst} \setminus m.\texttt{lca}()$ envia ACKs a grupos acima em $m.\texttt{dst}$; e

4. sempre que qualquer grupo g em $m.\texttt{dst}$ tenha enviado anteriormente mensagens a um grupo h abaixo de outros em $m.\texttt{dst}$, g envia NOTIF para h. Cada grupo notificado h reage enviando ACKs a grupos acima em $m.\texttt{dst}$ e, indutivamente, se comporta como g para notificar outros grupos. Como os grupos possuem uma ordem total, essa indução termina.

Do Lema 1 e dos fatos acima, segue que cada grupo em $m.\texttt{dst}$ recebe o histórico de cada grupo abaixo que está envolvido em mensagens ordenadas antes de m. □

Lemma 4 *Para qualquer mensagem m enviada via atomic multicast, qualquer grupo destino em $m.dst$ sabe quando a informação completa de dependências foi recebida.*

PROVA: Pelo Lema 1, cada grupo em $m.\texttt{dst}$ recebe m; pela Assunção 2, ele sabe quais são os grupos abaixo em $m.\texttt{dst}$ e aguarda seus respectivos ACKs. Cada ACK também informa se o grupo emissor notificou outros grupos, dos quais ACKs adicionais são aguardados (ver Lema 3, fatos 3 e 4). Assim, a partir das mensagens recebidas, qualquer destino de m consegue detectar se recebeu ACKs de todos os grupos com mensagens ordenadas antes de m. □

Proposition 1 *(Genuinidade) Um protocolo de multicast é genuíno se, em uma execução R, apenas o emissor e os destinos da mensagem se comunicam para propagar e ordenar uma mensagem multicast.*

PROVA: Pelo Algoritmo 2, quando m é enviada via multicast, há três tipos de mensagens possíveis no overlay: MSG, ACK e NOTIF. Mensagens MSG e ACK são trocadas exclusivamente entre grupos em $m.\texttt{dst}$, isto é, os destinos de m. Uma mensagem NOTIF só pode ser enviada de um grupo $g \in m.dst$ para h se existe uma mensagem anterior m' na execução R e $\{g,h\} \in m'.dst$. Assim, segue que apenas destinos de mensagens em R se comunicam para propagar e ordenar suas mensagens. □

Proposition 2 *(Quiescência) Um protocolo de multicast é quiescente se um processo que foi, mas não é mais, emissor ou destino de requisições em uma execução R, eventualmente para de se comunicar em R.*

PROVA: FlexCast segue um projeto reativo: um grupo só envia uma mensagem em reação ao recebimento de alguma mensagem. Considere que após o tempo t_1, um grupo g não é mais emissor nem destino de requisições. Após o tempo t_1, g ainda pode receber mensagens NOTIF de grupos abaixo e reagir enviando ACK ou outras mensagens NOTIF para grupos acima. Argumentamos que após t_1, g eventualmente deixa de receber mensagens NOTIF e, portanto, não reage mais gerando mensagens ACK e NOTIF.

Considere grupos l e h, respectivamente abaixo e acima de g. Assuma que l envia uma requisição r_1 para h após o tempo t_1. Nesse caso, l envia tanto uma mensagem de dados para h quanto uma mensagem NOTIF para g. Essas mensagens incluem o histórico de l, e l registra o prefixo de histórico encaminhado para cada grupo acima. Como, por hipótese, g não é mais destino, na requisição subsequente r_2 de l envolvendo h, l detecta que nenhuma mensagem envolvendo g aparece além do prefixo de histórico já enviado em r_1. Isso significa que não há mensagem envolvendo g para decidir ordem e, portanto, nenhuma mensagem NOTIF para g é necessária. Supondo que todos os grupos abaixo periodicamente enviem requisições via multicast para grupos acima, excluindo g, o tempo t_2 ocorre (isto é, g torna-se quiescente) após a primeira requisição de cada grupo abaixo para cada grupo acima ter sido multicast ($t_2 > t_1$). Se nenhuma dessas mensagens for enviada, como FlexCast é reativo, g é quiescente a partir de t_1 ($t_2 = t_1$). □

Proposition 3 *(Validade e Acordo)*

- **Validade**: Se um processo correto p realiza multicast de uma mensagem m, então eventualmente todos os processos servidor corretos $q \in g$, onde $g \in m.\texttt{dst}$, entregam m.
- **Acordo**: Se um processo p entrega uma mensagem m, então eventualmente todos os processos servidor corretos $q \in g$, onde $g \in m.\texttt{dst}$, entregam m.

PROVA: Devido à Assunção 1, aos Lemas 1, 3 e 4, e pelos Algoritmos 3 e 4, temos que todos os grupos em $m.\texttt{dst}$ eventualmente têm m e conseguem passar na avaliação da primeira condição da função *can-deliver(m)*. Resta verificar se existe alguma mensagem m' que deva ser entregue antes de m. Se não existir m', então o grupo pode entregar m. Se existir tal m', ela deve ser entregue primeiro. Assumindo ordem acíclica, o que é discutido mais adiante, os argumentos acima e por indução sobre dependências

entre mensagens, sempre haverá uma mensagem sem dependências pendentes para entregar, o que então habilitará outras mensagens a serem entregues, de forma que m possa ser entregue. Portanto, validade vale. Pelos mesmos argumentos, acordo vale. □

Proposition 4 *(Integridade)*

- **Integridade**: Para qualquer processo p e qualquer mensagem m, p entrega m no máximo uma vez, e apenas se $p \in g$, $g \in m.\texttt{dst}$, e m foi previamente enviada via multicast.

PROVA: Pelo Lema 1, uma mensagem m enviada via multicast alcança todos e apenas seus grupos destino. Qualquer outra possível mensagem (ACKS ou NOTIFS) não carrega mensagens a serem entregues. Assim, um grupo g entrega m apenas se $g \in m.dst$ e m foi enviada via multicast anteriormente. □

Proposition 5 *(Ordem de prefixo)*

- **Ordem de prefixo**: Para quaisquer duas mensagens m e m' e quaisquer dois processos servidor p e q tais que $p \in g$, $q \in h$ e $\{g,h\} \subseteq m.\texttt{dst} \cap m'.\texttt{dst}$, se p entrega m e q entrega m', então ou p entrega m' antes de m ou q entrega m antes de m'.

PROVA: Do Lema 2, existe um grupo único, $lcd(m,m')$, que atribui a ordem relativa entre m e m'. Dos Lemas 3 e 4, qualquer grupo adicional em $h \in m.dst \cap m'.dst$ recebe e preserva a ordem atribuída por $lcd(m,m')$. Assim, ordem de prefixo vale. □

Proposition 6 *(Ordem acíclica)*

Já definimos a relação $\prec$ no conjunto de mensagens que processos servidor entregam: $m \prec m'$ sse existe um processo que entrega m antes de m'.

- **Ordem acíclica**: A relação $\prec$ é acíclica.

Argumentamos que FlexCast garante ordem acíclica por contradição. Assuma que existe um ciclo C: $m_1 < m_2 < ... < m_k < m_1$. Seja C tal que $m_k < m_1$ ocorre no grupo h (isto é, h entrega m_k e então m_1), onde h é o grupo mais alto no overlay. Isso é possível porque o overlay induz uma ordem total sobre os grupos.

Seja q o grupo lcd que entrega as mensagens m_1 e m_2. Consideramos todas as combinações de `lca` para m_1 e m_2 (na Figura 3.4, casos a, b, c e d). Afirmamos que existe um caminho causal P desde a entrega de m_2 em q até a recepção da mensagem

m_k no processo p. Como processos entregam mensagens seguindo suas dependências causais, mostrar que existe um caminho causal P significa que, antes de p entregar m_k, ele sabe que m_1 precede m_k, o que leva a uma contradição, pois p não entregará m_k antes de entregar m_1.

PROVA: A prova da afirmação é por indução no tamanho k do ciclo C.

Passo base ($k = 2$): Este caso corresponde aos quatro padrões envolvendo as mensagens m_1 e m_2 (ver Figura 3.4), com $r = p$. Para os padrões (a) e (b), a afirmação segue diretamente. Para os padrões (c) e (d): como m_2 é endereçada a q e p, e p está abaixo de q no overlay, ao entregar m_2, de acordo com o Algoritmo 3, q envia uma mensagem ACK a p (com todas as dependências de q) e, portanto, existe um caminho causal.

Passo indutivo: Assuma que existe um caminho causal entre $m_2 < m_3 < ... < m_k$. Mostramos que existe um caminho de mensagens causal de m_1 até m_k, onde q entrega as mensagens m_1 e m_2, e r é um dos destinos de m_2 (junto com q e possivelmente outros processos).

Há cinco possibilidades de como q cria uma dependência entre m_1 e m_2, e de onde r está posicionado em relação a q no overlay de comunicação (ver Figura 3.4).

- Casos (a) e (b). Nesses casos, r está necessariamente abaixo de q no overlay, pois q envia m_2 via multicast e, caso contrário, r não seria destino de m_2. Nesses casos, o multicast de m_2 por q para r cria um caminho causal de m_1 para m_2 em r. Pela hipótese de indução, isso leva a um caminho causal até m_k.

- Casos (c) e (d). Nesses casos, consideramos que r está abaixo de q no overlay. Como q e r são destinos de m_2 e r está abaixo de q, pelos Algoritmos 3 e 4, q envia uma mensagem ACK para r e r espera pela mensagem ACK antes de entregar m_2. Isso cria um caminho causal entre a entrega de m_1 e m_2 em q e a entrega de m_2 em r. Pela hipótese de indução, segue que existe um caminho causal até a entrega de m_k em p.

- Caso (e). r está posicionado acima de q no overlay de comunicação. Como existe um caminho causal P entre a entrega de m_2 em r e o recebimento de m_k em p, é o caso que r enviou uma mensagem em P, digamos m_3. Quanto à geração de m_3, também pode ocorrer que $r = t$. Quanto à geração de m_1, também pode ocorrer que $s = q$.

  Como r sabe que esteve envolvido em m_2 com q, abaixo de r no overlay, r envia uma mensagem NOTIF para q e, como resposta, q envia uma mensagem ACK no caminho P para grupos em $m_3.dst$ abaixo de q (completando a informação de

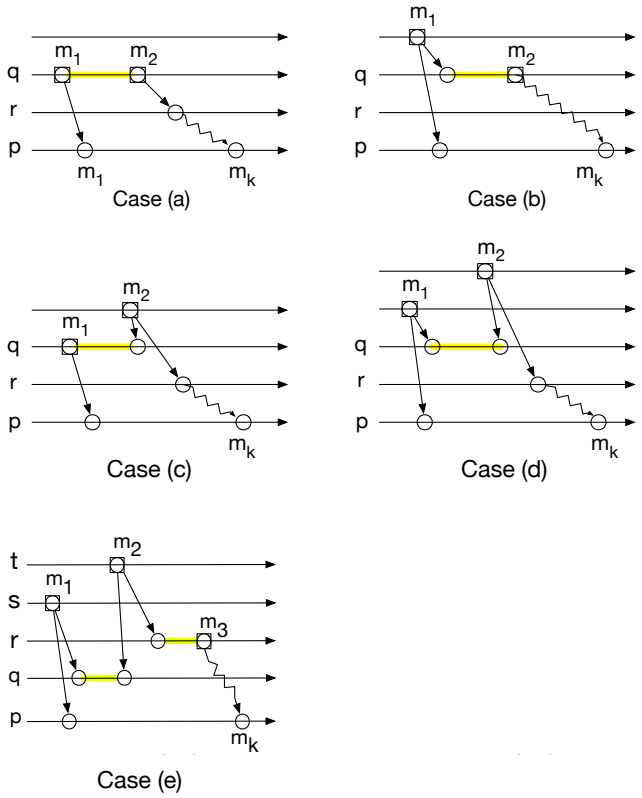


Figure 3.4. Cinco formas pelas quais um processo q pode criar uma dependência entre as mensagens m_1 e m_2. Quadrados denotam eventos de envio, círculos denotam eventos de entrega e setas denotam destinos de mensagens. Destaques em amarelo mostram dependências entre m_1 e m_2, enquanto setas em zigue-zague indicam caminhos causais.

que m_1 está no passado de m_3). Como grupos só podem entregar m_3 após a chegada desses ACKs, mensagens posteriores a m_3 constroem um caminho P até m_k em p iniciando em m_2 em r. Pela hipótese de indução, existe um caminho de m_1 até m_k. □

3.3.4 Considerações práticas

Coleta de lixo O protocolo como descrito até aqui não inclui coleta de lixo. No nosso protótipo de FlexCast, no entanto, podamos históricos locais associados a cada grupo ancestral. Um processo distinto periodicamente envia via multicast uma mensagem *flush* a todos os grupos. Uma vez que um grupo entrega essa mensagem, ele sabe que todas as mensagens que precedem *flush* podem ser coletadas. A intuição por trás desse

mecanismo é que, para entregar uma mensagem m de um ancestral específico, todas as dependências antes de m devem ser resolvidas e não precisam ser reavaliadas no futuro. Para reduzir ainda mais a comunicação, os históricos enviados com mensagens não incluem o histórico do sistema em constante crescimento. FlexCast envia apenas um *diff* do histórico para cada grupo descendente. A ideia é implementada mantendo-se o controle da última mensagem do histórico local enviada a cada descendente d e, em mensagens subsequentes para d, enviando um histórico que contém apenas as mensagens mais novas adicionadas desde a última comunicação com d.

Tolerando falhas FlexCast assume a mesma abordagem usada em outros protocolos de atomic multicast para tolerar falhas (por exemplo, [23], [39], [50], [69], [86], [22]), isto é, processos dentro de um grupo são mantidos consistentes usando replicação por máquina de estados [67, 90]. Isso significa que processos em um grupo podem falhar desde que processos suficientes permaneçam operacionais dentro do grupo. Consequentemente, grupos não falham como um todo e devem permanecer conectados (isto é, sem partições de rede). Tolerar a falha de um grupo requer suposições adicionais do sistema [89].

As implicações dessa abordagem no número de processos corretos por grupo e na comunicação entre processos dependem do protocolo de consenso particular usado para implementar replicação por máquina de estados dentro de um grupo. Por exemplo, Paxos [67] requer uma maioria de processos corretos dentro de cada grupo e pode tolerar perdas de mensagens. No nosso protótipo, simplesmente nos apoiamos em conexões TCP para garantir comunicação confiável.

3.3.5 Avaliação do FlexCast

Nesta seção apresentamos a avaliação do nosso protocolo. Começamos explicando a justificativa da avaliação, depois descrevemos o ambiente e os benchmarks usados e apresentamos os resultados.

Justificativa da avaliação Comparamos FlexCast a um protocolo distribuído de atomic multicast e a um protocolo clássico hierárquico de atomic multicast usando grupos de processo único (isto é, nenhuma falha é tolerada) nos três protocolos. Ao fazer isso, nossa avaliação foca nos custos inerentes de três classes de protocolos de atomic multicast (ver Tabela 3.1) e evita a sobrecarga introduzida pela replicação. Usamos o protocolo de Skeen como atomic multicast distribuído porque seu mecanismo de ordenação é usado por vários outros protocolos (por exemplo, [23], [39], [50], [69], [86]). Além disso, quando os grupos contêm um único processo, os protocolos FastCast [23] e Whitebox [50] de atomic multicast se comportam como no protocolo de Skeen. O

protocolo de Skeen é genuíno, pode ordenar mensagens em dois passos de comunicação, o que foi mostrado ser ótimo [88], e assume que quaisquer dois grupos podem se comunicar. Escolhemos ByzCast como um protocolo clássico hierárquico de atomic multicast. ByzCast não é genuíno e impõe um overlay em árvore à comunicação, o overlay mínimo que garante um sistema conectado. Em grupos de processo único, ByzCast não introduz qualquer sobrecarga particular para tolerar comportamento malicioso. Implementamos protótipos de todos os protocolos em Java.

Nossa avaliação experimental busca entender o comportamento dos protocolos considerados em implantações geograficamente distribuídas sujeitas a cargas de trabalho realistas. Nossa carga de trabalho estende o benchmark bem estabelecido TPC-C para acomodar localidade, uma propriedade comum em sistemas geo-distribuídos. Nesses cenários, buscamos responder às seguintes perguntas:

1. Qual é o impacto de diferentes overlays em FlexCast e em protocolos hierárquicos?

2. Quão rapidamente um protocolo consegue ordenar mensagens endereçadas a dois ou mais grupos?

3. Qual é a sobrecarga de comunicação de protocolos hierárquicos baseados em overlay?

4. Qual é o custo de comunicação de protocolos de atomic multicast?

Ambiente e implantação A configuração experimental foi montada com 12 máquinas servidoras e 24 máquinas clientes, conectadas por uma rede comutada de 1 Gbps, no CloudLab [34]. As máquinas são equipadas com oito núcleos ARMv8 de 64 bits a 2.4 GHz, e 64 GB de RAM. O software instalado nas máquinas foi Linux Ubuntu 20.04 (64 bits) e a máquina virtual Java de 64 bits versão 11.0.3. As máquinas se comunicam via TCP.

Consideramos uma rede de longa distância emulada que modela a Amazon Web Services (AWS). Cada grupo representa uma região da AWS e experimentamos uma implantação de 12 regiões da AWS, como ilustrado na Figura 3.5 (a). As latências emuladas entre regiões são baseadas em medições reais na AWS [21]. Processos clientes suficientes (para saturar nossa implementação de FlexCast) são distribuídos uniformemente nas 24 máquinas clientes que representam cada região/grupo, e eles enviam requisições ao grupo mais próximo. Ao entregar uma mensagem, cada destino da mensagem responde ao emissor da mensagem (cliente).

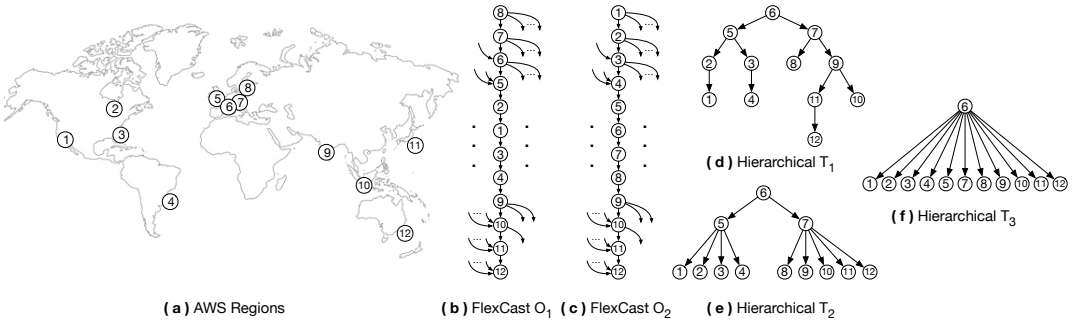


Figure 3.5. Regiões da AWS e diferentes overlays usados em nossa avaliação experimental.

Benchmark gTPC-C Desenvolvemos gTPC-C, um benchmark geograficamente distribuído inspirado no benchmark bem estabelecido TPC-C [29]. Traduzimos os armazéns do TPC-C em grupos, implantados em uma ou mais regiões da AWS, e as transações do TPC-C em mensagens multicast para seus respectivos armazéns.

De acordo com o benchmark TPC-C, clientes podem gerar as seguintes transações (com uma dada probabilidade): new order (45%), payment (43%), order status (4%), delivery (4%), ou stock level (4%). As últimas três transações são de único armazém (locais), resultando em uma mensagem multicast para o armazém base (home warehouse) do cliente. Como todos os protocolos de multicast têm o mesmo desempenho ao ordenar uma mensagem multicast para um único grupo, em nossas medições de latência consideramos apenas transações globais, que resultam em mensagens endereçadas a múltiplos armazéns. Consequentemente, esta carga de trabalho contém apenas transações new order e payment, sempre envolvendo dois ou mais armazéns. Transações new order podem ter de 5 a 15 itens, em que cada item tem 2% de probabilidade de ser emitido para um armazém que não é o armazém base do cliente, como definido pelo TPC-C.

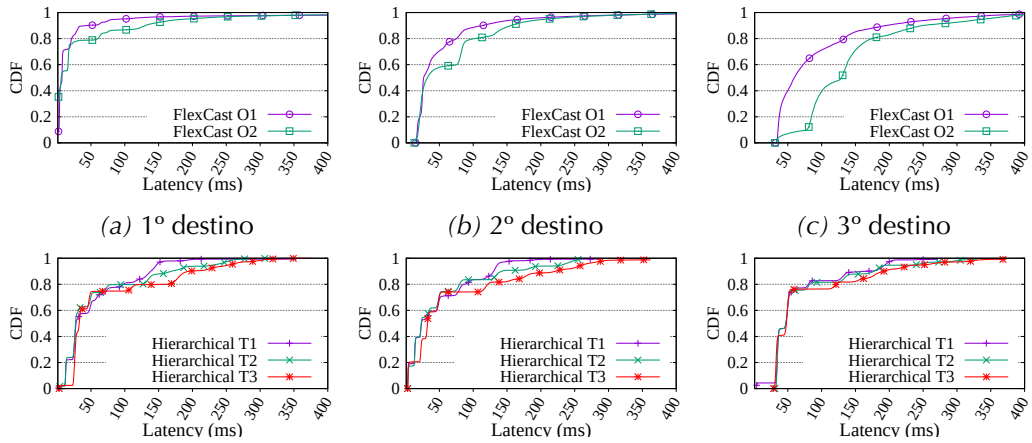
Para capturar localidade, ao escolher um armazém adicional além do armazém base do cliente, o cliente escolhe o armazém mais próximo do seu armazém base com uma alta probabilidade configurável, a taxa de *localidade*; caso contrário, o cliente escolhe o próximo armazém mais próximo, e assim por diante, até o armazém mais distante do seu armazém base. Nosso critério para definir localidade é inspirado por uma política comum de fornecedores no atacado em que, quando um item não está disponível no armazém mais próximo de um cliente (isto é, o armazém base), ele é enviado a partir do armazém mais próximo que tenha o item. Essa especificação de localidade implica que a maioria das mensagens é endereçada a apenas dois armazéns (como no TPC-C padrão), e algumas a três. Muito poucas são endereçadas a mais de

três grupos; portanto, não consideramos essas mensagens em nossos experimentos.

Clientes operam em um ciclo fechado emitindo uma transação por vez e são implantados na mesma região que seu armazém base. Cada experimento dura por um período de aproximadamente um minuto, no qual clientes coletam e armazenam dados de latência. Descartamos os primeiros e os últimos 10% dos dados coletados durante o experimento para evitar dados possivelmente ruidosos durante o aquecimento e o fim da execução.

O efeito dos overlays No primeiro conjunto de experimentos, investigamos o papel dos overlays em FlexCast e em protocolos hierárquicos. Comparamos a latência percebida pelos clientes em dois overlays do FlexCast e em três overlays hierárquicos (árvores), como ilustrado na Figura 3.5.

As árvores T_1, T_2 e T_3 contêm diferentes números de nós internos. Em princípio, um maior número de nós internos fornece uma melhor distribuição da sobrecarga de comunicação entre esses nós. Árvores com muitos nós internos, no entanto, podem levar a passos adicionais de comunicação ao ordenar mensagens. Para os overlays O_1 e O_2, inicialmente selecionamos um nó inicial (isto é, o nó central 8 em O_1 e o nó mais à esquerda 1 em O_2). Então, o nó mais próximo do inicial, o nó mais próximo do segundo nó escolhido e assim por diante. Como O_1 e O_2 são DAGs completos, um nó é conectado a todos os nós que o sucedem (por exemplo, o primeiro nó é conectado a todos os nós).



(a) 1º destino *(b)* 2º destino *(c)* 3º destino

(d) 1º destino *(e)* 2º destino *(f)* 3º destino

Figure 3.6. Latência por grupo destino ao variar overlays em FlexCast e em um protocolo hierárquico, gTPC-C com 90% de localidade.

A Figura 3.6 e a Tabela 3.2 apresentam os resultados. Relatamos a latência por

grupo endereçado pela mensagem. A latência do primeiro (respectivamente, segundo e terceiro) destino corresponde à primeira (respectivamente, segunda e terceira) resposta que o cliente recebe dos grupos endereçados pela mensagem. O_1 apresenta melhor desempenho do que O_2 para todos os destinos. Isso acontece porque O_1 explora melhor a localidade: nós mais altos no DAG têm as menores latências na distribuição geográfica. Daqui em diante, avaliamos FlexCast usando o overlay O_1.

		Destino								
		1º			2º			3º		
	Ov.	90p	95p	99p	90p	95p	99p	90p	95p	99p
FC	O_1	144.0	279.0	1403.1	398.0	829.0	2243.42	1406.0	2195.0	4542.5
	O_2	156.0	350.0	790.22	416.0	652.0	2006.83	1028.0	1681.5	3112.9
HR	T_1	229.0	267.0	311.0	261.0	288.0	403.0	307.0	386.0	408.0
	T_2	233.0	269.0	311.0	215.0	249.1	351.0	261.0	338.0	375.28
	T_3	311.0	398.0	544.0	381.0	480.0	622.0	397.0	531.6	621.0

Table 3.2. Percentis de latência em milissegundos para cada grupo destino ao variar o overlay (Ov.) em FlexCast (FC) e a árvore no protocolo hierárquico (HR), gTPC-C com 90% de localidade.

Diferentemente de FlexCast, cujo desempenho depende fortemente do overlay, um protocolo hierárquico não é tão sensível à árvore escolhida (mas ver também a discussão na Seção 3.3.5), embora as árvores tenham impacto no desempenho. T_1 apresenta desempenho ligeiramente melhor em todos os destinos do que T_2 e T_3. Isso se deve à sobrecarga de comunicação (discutida mais adiante na Seção 3.3.5) de envolver grupos que não são destinos e também ao efeito de gargalo de envolver a raiz da árvore em T_3 para todas as mensagens do sistema. A partir desses resultados, selecionamos T_1 para representar um protocolo hierárquico no restante da avaliação.

Vazão No segundo conjunto de experimentos, avaliamos o desempenho global do nosso gTPC-C padrão, medido em milhares de operações por segundo (kops), incluindo mensagens locais e globais, quando implantado em uma configuração com taxa de localidade de 99%.

Realizamos múltiplos experimentos aumentando gradualmente o número de clientes. A Figura 3.7 apresenta os resultados. Embora FlexCast tenha sido projetado para otimizar latência, ele consegue manter a mesma vazão dos outros protocolos até seu ponto de saturação. Esse efeito pode ser observado pela inflexão da curva de vazão de FlexCast a partir de 960 clientes. Esse comportamento é explicado pelo design orientado a latência do FlexCast. Como discutido a seguir, FlexCast apresenta desempenho particularmente bom sob cargas de trabalho de alta localidade, como gTPC-C,

em que a maioria das mensagens é endereçada a apenas dois destinos. À medida que o número de clientes aumenta, uma fração maior de mensagens tem como alvo mais de dois destinos, aumentando o número de mensagens auxiliares necessárias para a ordenação. Essa coordenação adicional atrasa a entrega em destinos mais baixos, levando, por fim, a uma saturação mais cedo.

Nos experimentos apresentados a seguir, consideramos configurações com 240 clientes. Isso se justifica pelo fato de que nenhum dos algoritmos está sujeito a efeitos de enfileiramento, o que interferiria em sua latência inerente.

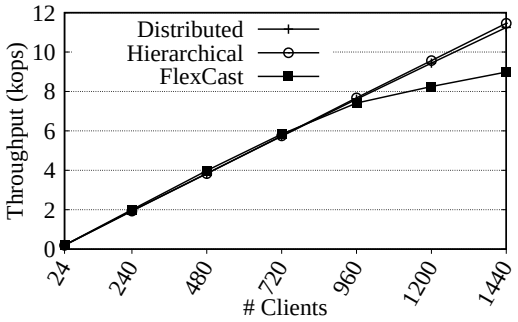
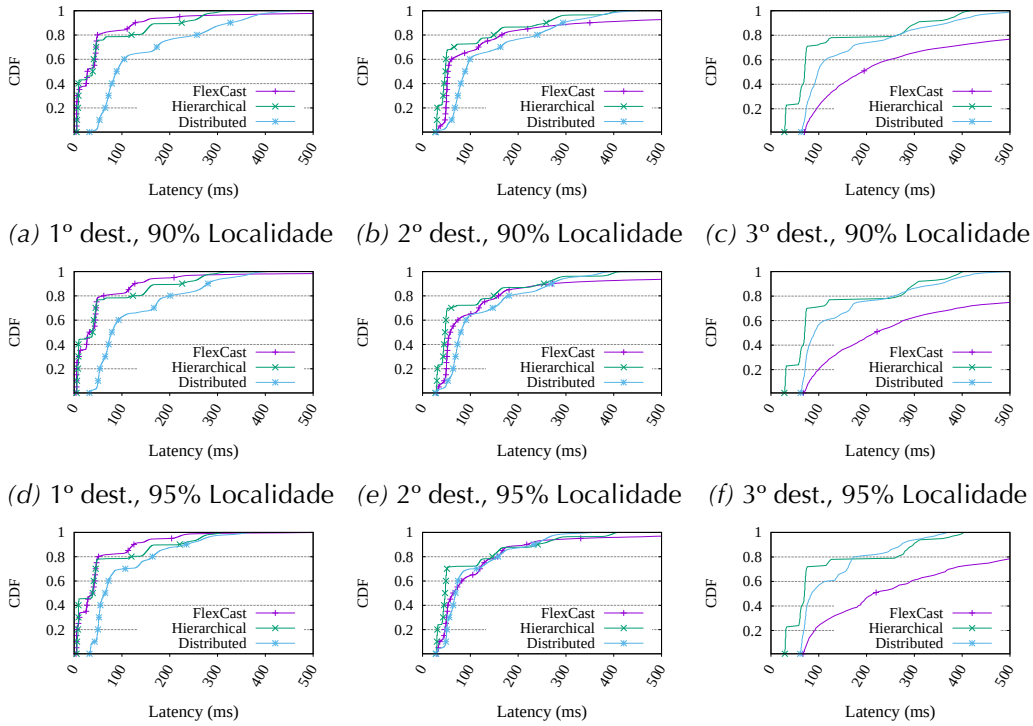


Figure 3.7. Vazão vs. número de clientes com 99% de localidade.

Latência No terceiro conjunto de experimentos, aumentamos a taxa de localidade e medimos a latência percebida pelos clientes ao receber uma resposta de cada um dos destinos de uma mensagem multicast global.

A Figura 3.8 e a Tabela 3.3 apresentam os resultados. FlexCast supera tanto o protocolo distribuído quanto o protocolo hierárquico na latência do primeiro grupo destino para as três taxas de localidade experimentadas. Atribuímos esse comportamento ao fato de que FlexCast se beneficia de dois aspectos que reduzem o custo de ordenar mensagens no primeiro destino em um cenário distribuído: *(i) Passos de comunicação:* enquanto em um protocolo distribuído os grupos endereçados por uma mensagem precisam trocar timestamps antes que um grupo destino possa entregar a mensagem, em FlexCast o primeiro grupo destino no DAG (isto é, o `lca` da mensagem) pode entregar a mensagem assim que a recebe de um cliente; o protocolo hierárquico também se beneficia desse aspecto, porém, em ByzCast, o `lca` de uma mensagem pode não ser um destino da mensagem, já que não é um protocolo genuíno. *(ii) Taxa de localidade:* ter uma carga de trabalho com alta taxa de localidade aumenta o número de mensagens que FlexCast consegue entregar usando menos passos de comunicação do que ambos os outros protocolos. Isso dá vantagem ao FlexCast, já que o custo de um passo de comunicação pode levar dezenas de milissegundos em cenários geográficos.



(a) 1º dest., 90% Localidade *(b)* 2º dest., 90% Localidade *(c)* 3º dest., 90% Localidade

(d) 1º dest., 95% Localidade *(e)* 2º dest., 95% Localidade *(f)* 3º dest., 95% Localidade

(g) 1º dest., 99% Localidade *(h)* 2º dest., 99% Localidade *(i)* 3º dest., 99% Localidade

Figure 3.8. Latência por destino (dest.) ao variar a taxa de localidade.

No segundo destino, FlexCast tem desempenho pior do que o protocolo hierárquico e supera o protocolo distribuído. Como discutido acima, protocolos hierárquicos requerem apenas um passo adicional de comunicação para ordenar uma mensagem no segundo destino, desde que ambos os destinos estejam diretamente conectados (condição que tipicamente se mantém devido à localidade), enquanto o protocolo distribuído, além de exigir que os grupos destino se comuniquem, também está exposto ao efeito de comboio (convoy effect), o que retarda ainda mais a entrega de mensagens [50]. No terceiro destino, a latência de FlexCast aumenta e a simplicidade do algoritmo de um protocolo hierárquico compensa. Em ambos o segundo e o terceiro destinos, FlexCast pode precisar de passos adicionais de comunicação para receber as mensagens ACK necessárias para entregar uma mensagem multicast m, avaliar possíveis dependências e aguardar que dependências sejam resolvidas (isto é, aguardar a entrega de mensagens anteriores ordenadas antes de m em grupos ancestrais). Embora FlexCast tenha desempenho pior do que ambos os protocolos hierárquico e distribuído no terceiro destino, mensagens endereçadas a três (ou mais) grupos são raras em gTPC-C, uma característica herdada de TPC-C.

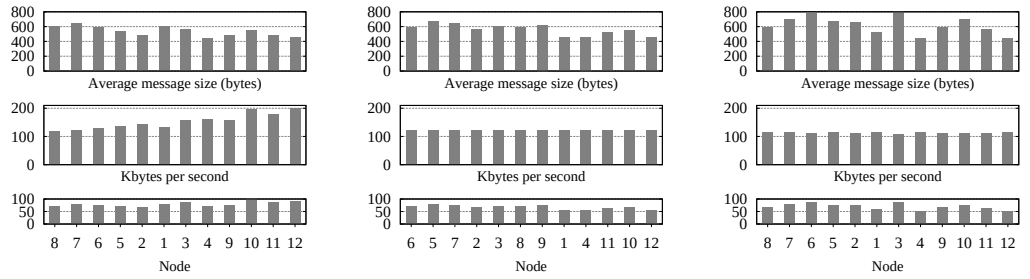
		Destino								
		1º			2º			3º		
	Loc.	90p	95p	99p	90p	95p	99p	90p	95p	99p
FC	90%	144.0	279.0	1403.1	398.0	829.0	2243.4	1406.0	2195.0	4542.5
	95%	131.0	217.0	1146.0	288.0	671.4	2192.6	1307.2	2231.6	4211.5
	99%	132.0	218.0	764.0	227.0	458.0	1562.0	1404.9	1975.7	3583.9
HR	90%	229.0	267.0	311.0	261.0	288.0	403.0	307.0	386.0	408.0
	95%	226.0	265.0	307.0	255.0	286.0	403.0	306.0	381.0	405.0
	99%	224.0	264.0	303.0	243.0	284.0	402.0	303.0	376.2	406.8
DT	90%	335.0	377.0	452.0	299.0	367.0	444.0	373.0	423.0	527.7
	95%	284.0	349.0	417.0	275.0	339.0	406.98	365.0	407.0	528.0
	99%	241.0	279.0	370.0	238.0	263.0	355.0	309.5	367.0	415.3

Table 3.3. Percentis de latência em milissegundos para cada destino ao variar a taxa de localidade (Loc.) para os três protocolos: FlexCast (FC), Hierárquico (HR) e Distribuído (DT).

Como consequência do overlay C-DAG de FlexCast e do fato de que cada cliente no benchmark gTPC-C está associado ao armazém mais próximo, clientes enviam a maioria de suas mensagens ao seu armazém base e ao próximo armazém mais próximo. A taxa em que esse fenômeno ocorre é regulada pela localidade configurada. Portanto, a maioria das mensagens na carga de trabalho tem conjunto de destinos disjunto. Isso aumenta a vantagem de FlexCast sobre um protocolo distribuído quando mensagens são endereçadas a dois grupos se os grupos estiverem colocados consecutivamente no C-DAG. O protocolo hierárquico também se beneficia da localidade, embora, por não ser genuíno, ele introduza sobrecarga de comunicação, quantificada na Seção 3.3.5. A taxa de localidade também ajuda a reduzir o número de mensagens auxiliares (isto é, ACK e NOTIF) necessárias para FlexCast garantir consistência na ordem total global, já que interdependências serão relativamente menores em tal cenário. A Tabela 3.3 mostra os percentis de latência (90, 95 e 99) de todos os destinos ao variar a taxa de localidade para todas as técnicas. Embora o protocolo hierárquico mostre, em média, melhor desempenho ao agregar as latências de todos os destinos, FlexCast é mais sensível à localidade. No primeiro destino, FlexCast reduz a latência 90p em 9% ao aumentar a localidade de 90% para 99%, enquanto o protocolo hierárquico reduz em 3%. Apesar de sua latência mais alta, o protocolo distribuído reduz a latência em até 29% ao aumentar a localidade de 90% para 99%.

O custo da troca de históricos Também avaliamos a quantidade de informação exigida por cada protocolo para implementar atomic multicast. Todos os protocolos propagam o payload da mensagem, como definido por gTPC-C, e informações específicas do pro-

tocolo, que no caso de FlexCast incluem históricos. A Figura 3.9 apresenta nossos resultados. Os gráficos no topo apresentam o número de mensagens recebidas por cada nó por segundo. Os gráficos no meio mostram o tamanho médio das mensagens por nó. Ao contrário dos outros protocolos, com tamanhos médios fixos, FlexCast mostra um aumento no tamanho médio das mensagens conforme os nós sobem na topologia C-DAG (ver Figura 3.5, overlay C-DAG O_1). Isso se deve ao fato de nós mais altos exigirem mais dados de histórico de seus ancestrais. Os gráficos na parte inferior mostram o volume total de dados trocados pelos nós por segundo.



(a) FlexCast *(b)* Hierárquico *(c)* Distribuído

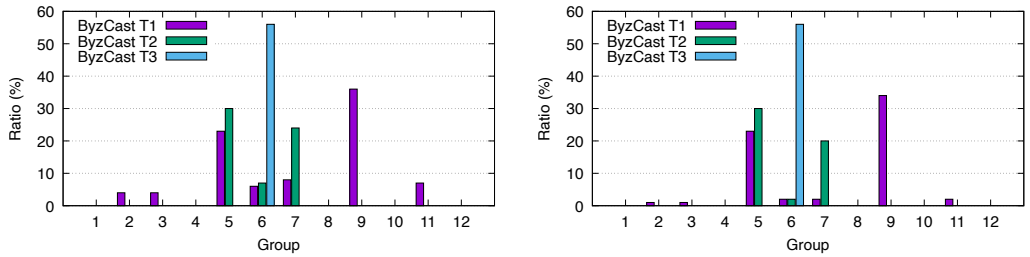
Figure 3.9. A quantidade de informação trocada por cada protocolo (99% de localidade, 720 clientes).

Em resumo, nossos experimentos indicam que FlexCast exibe um comportamento distinto, com nós mais altos no C-DAG de FlexCast trocando uma quantidade maior de dados do que nós mais baixos. Isso resulta em mensagens maiores em comparação com os outros protocolos. Em média, um nó troca 68.5 Kbytes por segundo no protocolo distribuído, 66 Kbytes por segundo no protocolo hierárquico e 79 Kbytes por segundo em FlexCast.

A sobrecarga da não genuinidade Também avaliamos a sobrecarga de comunicação de protocolos hierárquicos não genuínos (Figuras 3.2 e 3.10). Intuitivamente, sobrecarga de comunicação captura a quantidade de comunicação envolvendo um grupo devido a mensagens multicast que não são endereçadas ao grupo. Expressamos a sobrecarga de comunicação como uma porcentagem e a definimos como 1 menos a razão entre o número de mensagens de payload entregues por um grupo e o número de mensagens de payload recebidas pelo grupo durante uma execução do protocolo. Focamos em mensagens de payload, pois elas são tipicamente maiores do que mensagens auxiliares usadas em um protocolo.

A sobrecarga entre grupos depende do overlay em árvore e da carga de trabalho. Mas, enquanto todos os grupos internos em uma árvore estão potencialmente sujeitos

a sobrecarga de comunicação, grupos folha não têm sobrecarga, pois eles sempre estão nos destinos das mensagens que recebem. Localidade também desempenha um papel na sobrecarga de comunicação. Uma árvore pode se beneficiar da localidade conectando diretamente grupos que estão próximos entre si. Essa é a motivação por trás da árvore T_1: à medida que a localidade aumenta, a sobrecarga de T_1 diminui, pois a comunicação terá maior probabilidade de envolver grupos diretamente conectados (ver Tabela 3.4).



(a) 95% de Localidade *(b)* 99% de Localidade

Figure 3.10. Sobrecarga de comunicação de cada grupo em protocolos hierárquicos com 95% e 99% de localidade.

A árvore T_3 tem menor sobrecarga de comunicação do que T_1, mas isso vem ao custo de penalizar o grupo 6 (isto é, a raiz de T_3), que precisa suportar 56% de sobrecarga. Em T_1, os grupos 5 e 9 apresentam alta sobrecarga por serem raízes (menores ancestrais comuns) de diferentes subárvores que representam regiões geográficas separadas (América e Ásia). A raiz da árvore não tem muita sobrecarga, pois a localidade é alta entre grupos dentro da região Europa. O mesmo é observado em T_2, em que os grupos 5 e 7 de subárvores disjuntas apresentam as maiores sobrecargas.

As Tabelas 3.2 e 3.4 sugerem um trade-off: árvores com as menores latências estão sujeitas a maior sobrecarga, em média, enquanto árvores com pior desempenho têm menor sobrecarga de comunicação, em média.

3.4 Reconfiguração de atomic multicast

Na seção anterior, apresentamos o FlexCast, um protocolo de atomic multicast de estado da arte que assume um overlay em grafo acíclico direcionado completo (C-DAG) [9].

O desempenho de um protocolo de atomic multicast baseado em overlay é sensível às características da carga de trabalho e à localidade. A carga de trabalho determina os destinos das requisições e sua distribuição; a localidade da carga de trabalho define

Overlay	Localidade	Sobrecarga média	Máx
T_1	90%	9.16% (11.18)	36%
	95%	7.33% (11.12)	36%
	99%	5.41% (11.06)	34%
T_2	90%	5.75% (11.31)	30%
	95%	5.08% (10.50)	30%
	99%	4.33% (9.90)	30%
T_3	90%	4.66% (16.16)	56%
	95%	4.66% (16.16)	56%
	99%	4.66% (16.16)	56%

Table 3.4. Sobrecarga média, desvio padrão e sobrecarga máxima em árvores hierárquicas ao variar a taxa de localidade.

a proximidade geográfica entre clientes e destinos, o que afeta a latência de comunicação. Como demonstrado na Figura 3.6 na seção anterior, executar um protocolo baseado em overlay usando diferentes overlays, com a mesma carga de trabalho e localidade, pode resultar em desempenhos distintos. Para evitar uma reconfiguração estática, com as consequências de desempenho associadas à parada e reinicialização do sistema, propomos um mecanismo de reconfiguração dinâmica para alterar espontaneamente a configuração do overlay do FlexCast.

Definimos um modelo de otimização que captura o custo da ordenação de requisições e propõe um overlay mais adequado à carga de trabalho em execução no sistema. Como tanto a carga de trabalho quanto a localidade podem mudar ao longo da execução do sistema, o modelo determina periodicamente se um overlay alternativo deve ser adotado e, em caso afirmativo, dispara a reconfiguração.

Estendemos o FlexCast com um esquema de reconfiguração e o avaliamos em um ambiente WAN, conforme descrito a seguir. Mostramos que o protocolo de reconfiguração do FlexCast pode reduzir substancialmente a latência e aumentar a vazão ao transicionar seu overlay de uma configuração não ótima para uma configuração otimizada em um ambiente geograficamente distribuído. Por exemplo, alguns de nossos resultados demonstram que a vazão aumenta em aproximadamente 260% após uma reconfiguração, enquanto a latência diminui cerca de 83% para grupos que representam a maior parte da carga de trabalho, permanecendo estável ou sofrendo uma leve degradação para grupos menos frequentemente endereçados.

3.4.1 Protocolo de reconfiguração

O FlexCast apresenta melhor desempenho com cargas de trabalho que exibem algum grau de localidade e, à medida que a carga de trabalho muda, seu desempenho pode ser afetado. Portanto, é importante adaptar-se a possíveis mudanças de localidade. A maioria das abordagens na literatura adapta-se a mudanças de localidade migrando objetos entre partições, com o objetivo de converter operações entre partições em operações de partição única (por exemplo, [55]). Consideramos essas abordagens ortogonais ao FlexCast e focamos aqui na definição e reconfiguração do overlay de comunicação utilizado, um dos principais aspectos do FlexCast.

O FlexCast utiliza um overlay C-DAG construído sobre uma ordem total de grupos. Quando os grupos destino são vizinhos nessa ordem, a coordenação para entregar requisições é mínima. Caso contrário, a entrega pode ser atrasada dependendo do número de grupos intermediários envolvidos e das latências experimentadas pelos enlaces de rede que conectam esses grupos. Assumimos que todos os processos, incluindo servidores, clientes e o coordenador de reconfiguração (introduzido mais adiante), iniciam com o mesmo overlay C-DAG inicial, construído com base em uma ordem total compartilhada. Esse overlay é denominado configuração inicial. Clientes rotulam suas requisições com a configuração que conhecem, começando pela configuração inicial.

A motivação geral para a reconfiguração é minimizar a latência total de entrega para a carga de trabalho em execução. Isso significa que, com base em *(i)* o overlay atual, *(ii)* a latência média entre qualquer par de grupos, e *(iii)* a taxa observada de requisições para cada possível subconjunto de destinos, que caracteriza a carga de trabalho, um novo overlay é calculado. O novo overlay é obtido usando a função de minimização apresentada a seguir.

O Algoritmo 5 recebe como entrada o conjunto de grupos destino N, as latências lat entre grupos e uma representação da carga de trabalho *WL* que inclui a frequência de requisições para cada destino possível. A função *permutations* recebe o conjunto de grupos destino N e computa todas as possíveis ordens totais estritas[4] que podem ser derivadas de N. A função *maxPathLat* recebe como entrada uma ordem total possível o, um conjunto de destinos d, as latências dos enlaces lat entre grupos e retorna o maior caminho (isto é, a latência máxima entre o `lca` e o grupo mais alto dentro de d) de acordo com a ordem em o e as latências em lat. Por fim, a função de otimização identifica a ordem total em N que minimiza a soma das latências dos caminhos mais longos para cada destino, ponderada pela sua frequência na carga de trabalho *WL*.

Para obter o parâmetro de entrada *WL*, cada grupo registra a frequência de conjuntos de grupos destino para todas as requisições recebidas. Para coordenar o protocolo de reconfiguração, implementamos um processo separado denominado coordenador

[4]Uma ordem total estrita é irreflexiva.

Algorithm 5 Reconfiguração do overlay

1: $Input : N$ {*o conjunto de grupos destino*}
2: $Input : Lat(N \times N) \rightarrow \mathbb{R}$ {*latências entre pares de grupos*}
3: $Input : WL = \{(dest, freq) \mid dest \in 2^N \wedge freq \in \mathbb{N}\}$ {*para cada destino possível, uma frequência de requisições*}
4: **Função de otimização** (N, Lat, WL) **:**
5: **Minimizar:** {*encontra, entre todas as ordens totais em N*}
6: $o \in$ **permutations**(N) {*aquele que tem o menor*}
7: $\sum_{\forall l \in WL}$ **maxPathLat**$(o, l.dest, Lat) \times l.freq$ {*soma do maior caminho por destino ponderada pela sua frequência em WL*}
8: **Função permutations**(n : conjunto de grupos destino) :
9: **retorna** o conjunto de todas as ordens totais estritas com todos os elementos de n
10: **Função maxPathLat**(o : uma ordem total em N, $d : 2^N$, $lat : Lat$) : $\mathbb{R}$
11: **retorna** a latência do caminho mais longo entre os destinos em d
12: seguindo a ordem o e as latências em lat

de reconfiguração (RC), que é um cliente especial conectado a todos os grupos de servidores. O RC periodicamente realiza multicast de uma consulta CHECKFREQ para todos os grupos, que respondem com a carga de trabalho observada, incluindo informações de frequência de destinos, construindo o parâmetro *WL*. Juntamente com as entradas N, O e lat, o processo RC pode computar a função de otimização descrita no Algoritmo 5. Os grupos servidores reiniciam suas informações locais de frequência assim que respondem a uma CHECKFREQ, para garantir que mudanças de localidade sejam detectadas mais rapidamente, sem interferência das frequências registradas em iterações anteriores.

Uma vez que um novo overlay o' é computado, o processo RC dispara o mecanismo de reconfiguração realizando multicast de uma requisição $[\text{RECONFIGURE}, o']$ com o novo overlay definindo uma nova ordem total o' para todos os grupos de servidores. O FlexCast garante que mensagens RECONFIGURE sejam entregues a todos os grupos na mesma ordem. Isso garante que qualquer requisição enviada sob a configuração antiga seja processada antes da mensagem RECONFIGURE, assegurando que todas as dependências dentro da configuração antiga sejam resolvidas antes da transição para a nova configuração. Devido à assincronia, um grupo pode receber uma requisição rotulada com uma nova configuração que ele ainda não reconheceu ou para a qual ainda não foi reconfigurado. Nesses casos, o processo armazena temporariamente essas requisições em um buffer. Assim que o grupo recebe a notificação de mudança de configuração e se reconfigura adequadamente, as requisições armazenadas são processadas.

Quando um grupo está pronto para entregar a mensagem RECONFIGURE, ele inicia

o processo de reconfiguração. Esse processo envolve várias etapas principais: suspender o tratamento de requisições recebidas, aguardar que os grupos abaixo no novo overlay o' estabeleçam conexões, iniciar conexões com os grupos acima em o', processar quaisquer requisições que estavam previamente armazenadas para a nova configuração e, por fim, retomar a operação normal do FlexCast, agora tratando requisições de acordo com a configuração atualizada.

Como nosso coordenador de reconfiguração apenas notifica os servidores sobre a nova configuração, um cliente pode enviar uma requisição r rotulada com uma configuração desatualizada e endereçá-la ao antigo r.`lca`(). Quando um servidor recebe uma requisição rotulada com uma configuração antiga, ele informa o cliente sobre o overlay atualizado o', instruindo o cliente a mudar para a nova configuração e retransmitir a requisição para o `lca` apropriado em o'.

3.4.2 Avaliação da reconfiguração

Esta seção apresenta a avaliação do mecanismo de reconfiguração do FlexCast. Descrevemos o ambiente e os benchmarks utilizados, apresentamos os resultados e resumimos as lições aprendidas.

Ambiente e implantação A configuração experimental foi montada em um cluster privado com 12 máquinas servidoras e 24 máquinas clientes conectadas por uma rede comutada de 1 Gbps. Cada máquina possui dois processadores AMD EPYC 7282 de 16 núcleos e 32 GB de RAM. O software instalado nas máquinas foi Linux Ubuntu 18.04 (64 bits) e a máquina virtual Java de 64 bits versão 11.0.3. As máquinas se comunicam via TCP.

Nossa avaliação do FlexCast busca compreender o comportamento dos protocolos considerados em implantações geograficamente distribuídas sujeitas a cargas de trabalho realistas e como nosso esquema de reconfiguração pode afetar o desempenho do sistema. Consideramos uma rede de longa distância emulada que modela a Amazon Web Services (AWS), a mesma ilustrada na Figura 3.5. Os primeiros experimentos de reconfiguração foram executados em uma implantação com 3 grupos de servidores, distribuídos em 3 regiões da AWS (Canadá, N. Virginia e São Paulo), conforme ilustrado na Figura 3.11. Também executamos experimentos com 6 grupos de servidores (Figura 3.15), em uma implantação com 6 regiões da AWS abrangendo 3 continentes (América, Europa e Ásia).

Para avaliar o sistema, utilizamos novamente o benchmark gTPC-C, um benchmark geograficamente distribuído e sensível à localidade, inspirado no benchmark bem estabelecido TPC-C [29], no qual os armazéns do TPC-C são traduzidos em grupos, implantados em uma ou mais regiões da AWS, e as transações do TPC-C em requisições

multicast para seus respectivos armazéns (ver Seção 3.3.5 para mais detalhes). Clientes operam em um ciclo fechado emitindo uma transação por vez e são implantados na mesma região de seu armazém base. Os experimentos duram aproximadamente de um a dois minutos, dependendo do cenário, período durante o qual os clientes coletam e armazenam dados de desempenho. Descartamos os primeiros e os últimos 10% dos dados coletados durante o experimento para evitar dados possivelmente ruidosos durante o aquecimento e o encerramento da execução. Conduzimos experimentos expondo o sistema a mudanças na localidade da carga de trabalho. Analisamos como nosso protocolo de reconfiguração responde a essas mudanças.

O impacto da reconfiguração Nesses experimentos, nosso processo *RC* opera periodicamente (aproximadamente a cada 10 segundos) calculando possíveis overlays otimizados para a carga de trabalho atual. A carga de trabalho é determinada da seguinte forma: $\{[0,1] = 32.6\%; [0,2] = 59.2\%; [1,2] = 4.6\%; [0,1,2] = 3.5\%;\}$, onde $[x,\ldots,y] = z\%$ significa que requisições são multicast para os grupos destino $x,\ldots,y$ com probabilidade z.

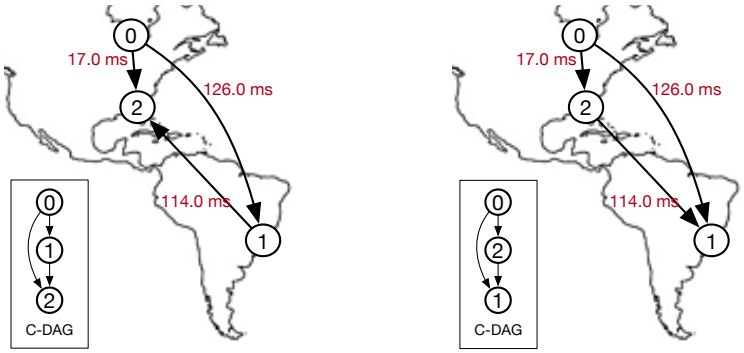


Figure 3.11. Configuração inicial C_1 (esquerda); e configuração C_2 (direita), computada pelo Algoritmo 5, para um overlay com 3 grupos.

Inicialmente, o sistema é configurado com a configuração C_1, desfavorável para a carga de trabalho (Figura 3.11, à esquerda). Por exemplo, um motivo que torna a configuração C_1 subótima para a carga de trabalho dada é que uma parcela significativa (32.6%) das requisições é enviada para os grupos 0 e 2. Isso leva a mensagens auxiliares potenciais sendo roteadas do grupo 0 para o grupo 1 e, então, do grupo 1 para o grupo 2 (um caminho com alta latência) antes que o grupo 2 possa entregar essas requisições. Após cerca de 10 segundos, o protocolo de reconfiguração é ativado, o algoritmo determina um novo overlay otimizado para a carga de trabalho atual e o sistema adota essa nova configuração C_2 (Figura 3.11, à direita).

Durante a reconfiguração (áreas cinzas na Figura 3.12), o sistema passa por um período de instabilidade de desempenho, evidenciado por interrupções tanto nos gráficos de vazão quanto de latência. Essa instabilidade ocorre porque os servidores estão executando as etapas de reconfiguração (isto é, alocando novas estruturas de dados, gerenciando novas conexões e processando requisições armazenadas em buffer). Além disso, em nosso protocolo, os clientes precisam identificar e se adaptar à nova configuração durante a comunicação com os servidores. Quando um cliente envia uma requisição baseada na configuração antiga, o servidor informa o cliente sobre a nova configuração. O cliente então deve retransmitir a requisição de acordo com a nova configuração.

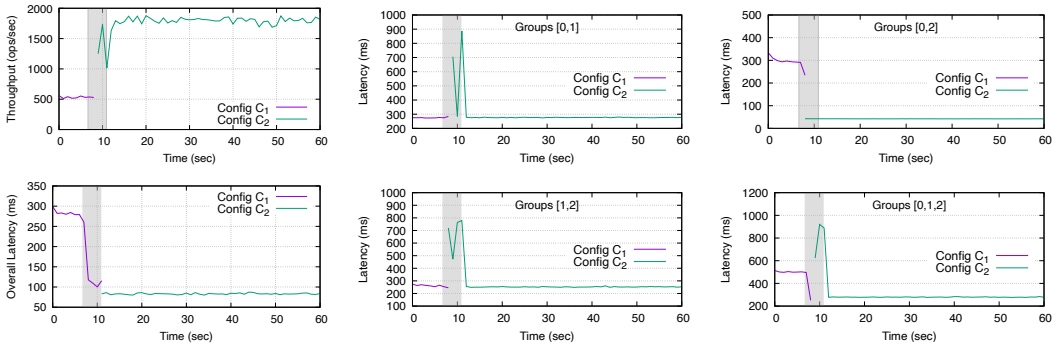


Figure 3.12. O impacto da reconfiguração no desempenho do sistema. Coluna da esquerda: vazão do sistema em requisições por segundo (topo) e latência média percebida pelos clientes em milissegundos (base); colunas do meio e da direita: latência de percentil 90 por conjunto de destinos. Todos os gráficos mostram dados para as configurações C_1 e C_2, enquanto o protocolo de reconfiguração é executado 10 segundos após o início da execução. A área cinza é o intervalo de reconfiguração.

Após esse período de instabilidade, a vazão do sistema melhora e a latência de resposta para os clientes diminui significativamente (Figura 3.12, topo à esquerda). Como os clientes executam em ciclo fechado, as menores latências da configuração C_2 levam a uma vazão aumentada.

Os resultados na Figura 3.12 mostram uma redução significativa nas latências de requisições para os destinos $[0, 2]$ e $[0, 1, 2]$. Com a configuração C_2, para destinos $[0, 2]$, o grupo `lca` 0 pode enviar requisições diretamente para o grupo 2 sem envolver o grupo 1. Isso elimina mensagens auxiliares atravessando o caminho de alta latência entre os grupos 0, 1 e 2. Como resultado, o grupo 2 pode entregar essas requisições imediatamente sem esperar por mensagens ACK do grupo 1, tornando o processo de comunicação mais eficiente.

Quanto aos destinos $[0, 1, 2]$, a melhoria de latência é atribuída a mudanças no pa-

pel dos grupos intermediários. Na configuração anterior, o grupo 1 era o intermediário e também era o grupo com maior latência, tornando o caminho de receber requisições do grupo 1 e enviar ACKs para o grupo 2 o mais custoso em termos de latência. Na nova configuração, entretanto, o grupo 2 atua como intermediário. Ele recebe a requisição do grupo `lca` quase imediatamente e envia sua mensagem ACK em paralelo com a requisição do grupo `lca` para o grupo 1. Esse processamento em paralelo permite que o grupo 1 receba tanto a requisição quanto a mensagem ACK quase simultaneamente, resultando em uma latência total de caminho significativamente menor do que na configuração anterior, melhorando o fluxo de comunicação e reduzindo atrasos, o que leva a melhor desempenho do sistema.

A Figura 3.13 destaca a capacidade do FlexCast de se adaptar a mudanças de carga de trabalho por meio de múltiplas reconfigurações. Inicialmente, quando o sistema detectou desempenho subótimo com o overlay inicial, ele se reconfigurou para um overlay mais adequado, resultando em maior vazão e menor latência. Posteriormente, uma mudança na localidade da carga de trabalho, conduzida pelos clientes, levou a uma queda de desempenho. Em resposta, o processo de monitoramento do sistema identificou mudanças nas frequências de destinos das requisições, recalculou um novo overlay e melhorou novamente o desempenho geral do sistema.

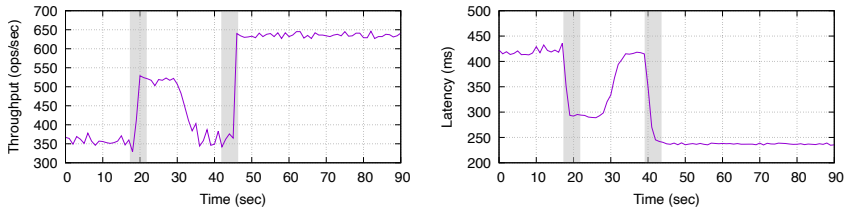


Figure 3.13. Execução do nosso protocolo de reconfiguração (intervalos em área cinza) adaptando dinamicamente o FlexCast a diferentes localidade de carga de trabalho.

Impacto em mensagens auxiliares Como demonstrado na Figura 3.14 (esquerda), apesar da vazão aumentada da configuração C_2, o número de mensagens auxiliares (ACKs e NOTIFs) diminui, o que também contribui para o desempenho geral do sistema. A Figura 3.14 (meio) também revela uma mudança no tráfego de dados por segundo de cada grupo. Especificamente, os grupos 1 e 2 apresentam uma inversão no volume de dados à medida que suas posições na hierarquia C-DAG mudam, sendo que o grupo ao final da hierarquia tipicamente recebe mais dados devido aos históricos de requisições e mensagens auxiliares. Todos os grupos veem um aumento de volume devido à vazão aumentada de C_2.

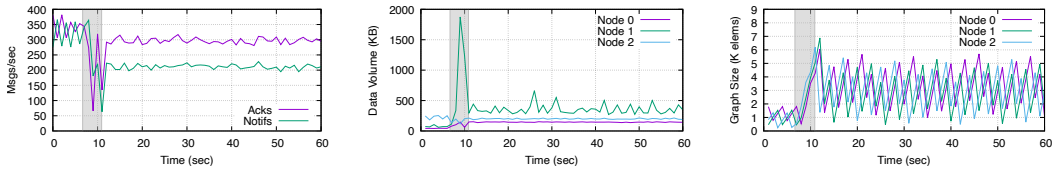


Figure 3.14. O impacto da reconfiguração em mensagens auxiliares (esquerda), volume de dados (meio) e tamanho da estrutura de dados interna (um grafo) usada para computar dependências em um grupo (direita). A área cinza é o intervalo de reconfiguração.

Além disso, o tamanho interno do grafo histórico de processamento de dependências aumenta para todos os grupos (Figura 3.14, direita) porque há maior vazão no sistema. Esse fenômeno beneficia grupos mais baixos na hierarquia. Esses grupos passam a ter acesso mais frequente a informações de dependência disponíveis localmente, em vez de esperar que essas informações cheguem por mensagens auxiliares. Isso simplifica o processamento e melhora a eficiência ao reduzir a necessidade de comunicação inter-grupos para obter dados de dependência.

Aumentando o tamanho do sistema Também conduzimos experimentos com diferentes números de grupos no sistema. A Figura 3.15 apresenta os resultados para uma execução com 6 grupos, em que mostramos apenas os grupos que experimentaram os impactos de latência mais significativos. Grupos não mostrados tiveram mudanças desprezíveis.

Observamos que, com mais grupos no sistema, a reconfiguração pode beneficiar desproporcionalmente certos grupos em relação a outros, dependendo das frequências de requisições na carga de trabalho. Isso é evidente porque, apesar de um aumento significativo na vazão geral do sistema (Figura 3.15, esquerda), as melhorias de latência não são distribuídas uniformemente entre todos os grupos. No entanto, destinos $[0, 2]$, que apresentam uma melhoria substancial de latência (Figura 3.15, meio), correspondem à maior parte das requisições da carga de trabalho, explicando o aumento expressivo de vazão. Além disso, o número de mensagens auxiliares também diminui significativamente (Figura 3.15, direita) após a reconfiguração.

O custo da otimização A Tabela 3.5 apresenta uma análise comparativa do desempenho da nossa função de otimização, detalhada no Algoritmo 5, em várias configurações de sistema com diferentes números de grupos. A tabela mostra o tempo médio, em segundos, necessário para computar um novo overlay com base nas frequências observadas de destinos da carga de trabalho e no tamanho do sistema. Implementamos

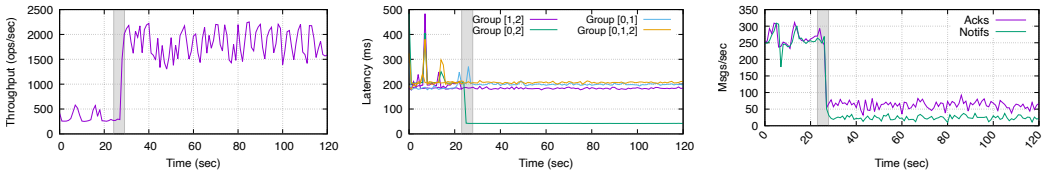


Figure 3.15. Desempenho da reconfiguração com 6 grupos e taxa de localidade de 98%. A área cinza é o intervalo de reconfiguração.

uma versão single-thread e uma multi-thread da nossa função de otimização. Nossas observações indicam que, embora o tempo necessário para computar um novo overlay para sistemas com 3 a 9 grupos permaneça relativamente curto, ele aumenta significativamente conforme o tamanho do sistema cresce, em ambas as versões. Esse problema ocorre porque o procedimento atual implementado para o Algoritmo 1 realiza uma busca exaustiva por todos os overlays possíveis. O algoritmo poderia ser projetado de forma mais eficiente se fosse abordado com métodos heurísticos. No entanto, não perseguimos essa otimização porque o cálculo é realizado em segundo plano por um processo separado, o que não impacta a responsividade do sistema. Além disso, em redes de grande escala como a AWS, ter mais de 12 regiões geralmente é desnecessário para representar áreas remotas de forma suficiente para cobrir o globo inteiro.

	3 grupos	6 grupos	9 grupos	12 grupos
Multi-thread	0.14	0.17	0.65	423.02
Single-thread	0.14	0.16	1.45	2452.48

Table 3.5. Tempo médio de execução (em segundos) da nossa função de otimização ao variar o tamanho do sistema, para as versões single-thread e multi-thread.

3.5 Trabalhos relacionados

Nesta seção, apresentamos uma visão abrangente de protocolos de atomic multicast e de reconfiguração propostos na literatura, destacando seus mecanismos centrais, pontos fortes e limitações.

3.5.1 Atomic multicast

Uma das abordagens mais antigas para atomic multicast, proposta por D. Skeen [13], envolve coordenação baseada em timestamps entre os destinatários da mensagem.

Nesse protocolo, uma mensagem m multicastada é primeiro enviada a todos os seus destinos. Ao receber m, cada destino atribui a ela um timestamp local e encaminha esse timestamp aos demais destinatários. Quando um destino reúne timestamps de todos os destinatários, ele calcula o timestamp final da mensagem como o máximo dos valores coletados. As mensagens são então entregues na ordem de seus timestamps finais. Embora esse protocolo garanta genuinidade, ele não possui tolerância a falhas.

Diversos protocolos de atomic multicast estendem a técnica de ordenação do Skeen para tolerar falhas [23], [39], [50], [69], [86]. Em todos esses protocolos, a ideia é implementar os destinos como grupos de processos. Assim, mensagens são endereçadas a um ou mais grupos de processos, em vez de a um conjunto de processos, como no protocolo original. Embora alguns processos em um grupo possam falhar, cada grupo atua como uma entidade confiável, cuja lógica é replicada dentro do grupo usando replicação por máquina de estados [90]. Protocolos recentes buscam reduzir o custo de replicação dentro dos grupos mantendo a ideia original do Skeen de atribuir timestamps às mensagens e entregar mensagens em ordem de timestamp. FastCast [23] melhora o desempenho ao executar, de forma otimista, partes da lógica de replicação dentro de um grupo em paralelo. WhiteBox atomic multicast [50] utiliza a abordagem líder-seguidor para replicar processos dentro dos grupos. RamCast [69] depende de memória compartilhada distribuída (RDMA) para reduzir a latência.

Delporte e Fauconnier [30] apresentam um protocolo distribuído genuíno de atomic multicast que não depende da troca de timestamps para ordenar mensagens. O protocolo atribui uma ordem total aos grupos e encaminha mensagens sequencialmente através de seus grupos de destino seguindo essa ordem. Uma mensagem m multicastada é inicialmente enviada ao grupo mais baixo em $m.\texttt{dst}$ de acordo com a ordem total. Quando o grupo recebe m, ele usa consenso para ordenar e entregar m dentro do grupo, então m é encaminhada ao próximo grupo em $m.\texttt{dst}$, de acordo com a ordem total dos grupos. Um grupo que entrega m só pode ordenar a próxima mensagem quando sabe que m está ordenada em todos os grupos em $m.\texttt{dst}$, o que ocorre após receber uma mensagem END do último grupo em $m.\texttt{dst}$. Embora a disseminação da mensagem siga uma ordem, a mensagem END retorna a cada grupo envolvido e, portanto, o protocolo é um protocolo distribuído de atomic multicast. Além de precisar de $n + 1$ etapas para entregar uma mensagem, onde n é o número de destinos da mensagem, como os grupos permanecem bloqueados até a chegada da mensagem END, este protocolo é afetado pelo efeito de comboio [2].

Alguns protocolos restringem a comunicação entre processos por meio de um overlay em árvore que determina como grupos podem se comunicar (por exemplo, [22, 42]). Para ordenar uma mensagem m usando uma árvore, m é primeiro enviada ao grupo de menor ancestral comum entre aqueles em $m.\texttt{dst}$, no pior caso a raiz da árvore de overlay. Em seguida, m é sucessivamente ordenada pelos grupos inferiores na

árvore até alcançar todos os grupos em $m.\texttt{dst}$. Um invariante importante é que grupos inferiores na árvore preservam a ordem induzida por grupos superiores. Embora simples, esse protocolo não é genuíno, pois uma mensagem pode precisar ser ordenada por um grupo que não está no conjunto de destinos da mensagem. Enquanto o protocolo baseado em árvore proposto por Garcia-Molina e Spauster [42] não tolera falhas, o ByzCast [22] consegue suportar falhas Bizantinas.

O protocolo Arrow [63] é um protocolo baseado em árvore, não tolerante a falhas, voltado a grupos abertos. Ele surge da combinação de um protocolo de multicast confiável com um protocolo distribuído de swap. Arrow assume um grafo G e uma árvore geradora T sobre G. Inicialmente, cada nó v em T possui $link(v)$, que é seu vizinho em T ou ele próprio se v for um sumidouro (inicialmente apenas a raiz de T). Para multicastar m, um nó v envia uma mensagem através de $link(v)$, que é encaminhada à raiz da árvore. Por definição, a raiz enviou a última mensagem antes de m. À medida que a mensagem é encaminhada, arestas mudam de direção e v torna-se a nova raiz (que enviou a última mensagem, que agora é m). Embora genuíno, esse procedimento pode resultar em mensagens de swap percorrendo o diâmetro de T e, somente então, um multicast, usando um multicast confiável subjacente, é emitido.

Restringir a comunicação como em uma árvore pode levar a algoritmos de atomic multicast mais simples. Além disso, se a comunicação precisa ser autenticada, como em protocolos tolerantes a falhas Bizantinas, um overlay em árvore requer menos chaves para serem mantidas e trocadas entre processos do que um protocolo distribuído totalmente conectado. Por fim, um protocolo totalmente conectado é uma suposição razoável em sistemas que executam dentro do mesmo domínio administrativo (por exemplo, o Spanner do Google [26]). Em outros contextos (por exemplo, sistemas descentralizados), no entanto, múltiplas entidades de diferentes domínios administrativos colaboram, mas não desejam estabelecer conexões com todos os demais domínios.

3.5.2 Reconfiguração

Não temos conhecimento de trabalhos que abordem reconfiguração de overlay em protocolos de atomic multicast. No entanto, reconfiguração de overlay tem sido usada para melhorar segurança de rede [33], tolerância a falhas de rede [80] e confiabilidade de sistemas publish/subscribe [82]. Em outro contexto, a reconfiguração do conjunto de réplicas que implementa o sistema tem sido usada em diversos protocolos distribuídos, como replicação por máquina de estados [12, 90], sistemas de quórum [3] e blockchains [10].

3.6 Conclusão

Aqui resumimos nossas principais descobertas e contribuições para aprimorar o desempenho de abstrações de comunicação em sistemas SMR. Nossa primeira contribuição introduziu uma nova propriedade, quiescência, para refinar a propriedade de minimalidade, garantindo a integridade de protocolos de atomic multicast genuínos ao evitar comunicação desnecessária, e o FlexCast, o primeiro protocolo de atomic multicast genuíno baseado em overlay, projetado para reduzir latência sem incorrer na sobrecarga de comunicação típica de soluções não genuínas. Analisamos minuciosamente o comportamento do FlexCast em diversas condições experimentais e demonstramos suas vantagens sobre protocolos de atomic multicast de ponta em implantações geograficamente distribuídas.

Nossa segunda contribuição foi o projeto e a implementação de um mecanismo de reconfiguração dinâmica para o FlexCast. Essa extensão permite que o protocolo adapte seu overlay de comunicação em tempo real com base na localidade da carga de trabalho e nas condições do sistema. Por meio de uma avaliação cuidadosa, mostramos que esse mecanismo leva a melhorias substanciais de desempenho em ambientes dinâmicos, tornando o FlexCast não apenas eficiente em topologias estáticas, mas também robusto e responsivo a mudanças.

3.6.1 FlexCast

Como baseado em overlay, o FlexCast considera conectividade reduzida em diferentes cenários de implantação. Como genuíno, ele favorece localidade geográfica e evita sobrecarga de comunicação. Para combinar ambos os aspectos, o FlexCast assume um overlay em DAG completo. Como mensagens podem entrar no overlay por diferentes grupos (nós) do DAG, cada grupo toma decisões locais de ordenação. Tiramos as seguintes conclusões de nossa avaliação experimental:

- O FlexCast é mais sensível ao overlay escolhido do que o protocolo hierárquico quando se trata de latência. A árvore escolhida, no entanto, impacta a sobrecarga de comunicação do protocolo hierárquico.

- O FlexCast supera consistentemente o protocolo distribuído (um algoritmo genuíno) em todas as configurações experimentadas. O FlexCast tem melhor desempenho do que o protocolo hierárquico no primeiro grupo de destino e pior na latência do segundo e terceiro destinos. No entanto, mensagens endereçadas a três (ou mais) grupos são raras em TPC-C e gTPC-C. Como um protocolo genuíno, o FlexCast não tem sobrecarga de comunicação (Seção 3.3.5), em contraste com um protocolo hierárquico.

- O protocolo hierárquico apresenta um compromisso entre latência e sobrecarga de comunicação. Embora sobrecarga de comunicação seja inerente a protocolos de atomic multicast não genuínos, no protocolo hierárquico, as árvores com melhor desempenho possuem a maior sobrecarga e vice-versa.

Um desafio interessante resolvido pelo FlexCast e ainda não abordado por outros protocolos de atomic multicast é como garantir ordem acíclica global a partir de informação local de ordenação proveniente de diferentes grupos. Isso é alcançado usando um protocolo sofisticado baseado em histórico. Apresentamos o projeto do FlexCast, sua implementação, e propusemos um novo benchmark para avaliá-lo: o gTPC-C integra distribuição geográfica e localidade ao conhecido benchmark TPC-C. O FlexCast apresenta redução importante de latência em cenários geograficamente distribuídos quando comparado a um algoritmo genuíno de atomic multicast ótimo em latência e a um protocolo hierárquico não genuíno.

3.6.2 Reconfiguração do FlexCast

Nesta segunda contribuição, estendemos o FlexCast com um mecanismo de reconfiguração. Propusemos, implementamos e avaliamos o esquema de reconfiguração do FlexCast, permitindo que ele se adapte dinamicamente a mudanças de carga. Mostramos que nosso protocolo de reconfiguração entrega um sistema capaz de se adaptar dinamicamente às demandas mutáveis da carga de trabalho, apresentando melhor desempenho à medida que as condições do sistema evoluem, tornando o sistema adequado para ambientes dinâmicos. Tiramos as seguintes conclusões de nossa avaliação experimental:

- Nosso protocolo de reconfiguração permite que o FlexCast se adapte à localidade da carga de trabalho atual e modifique seu overlay de comunicação em tempo real, resultando em desempenho significativamente melhor com as novas configurações propostas.

- Embora o desempenho geral do sistema não seja significativamente afetado, o algoritmo para calcular uma nova configuração torna-se ineficiente à medida que o número de grupos aumenta. No entanto, esse problema poderia ser tratado empregando uma abordagem alternativa que use heurísticas em vez de força bruta para determinar um novo overlay.

3.6.3 Observações finais

De forma geral, nossos resultados validam os objetivos de projeto do FlexCast: fornecer atomic multicast de baixa latência em sistemas geograficamente distribuídos sem in-

correr em sobrecarga de comunicação. Ao combinar uma estrutura de overlay C-DAG com um modelo de execução genuíno e quiescente e um mecanismo de reconfiguração dinâmica, o FlexCast mostra-se uma solução flexível e eficiente para sistemas distribuídos modernos. Sua capacidade de se adaptar à localidade da carga de trabalho e a condições de implantação em evolução o torna uma direção promissora para futuros sistemas SMR confiáveis.

Capítulo 4

Gerenciamento de estado em sistemas SMR

Apesar de seu sucesso (por exemplo, [16, 26, 44, 56]), SMR continua complexo de implementar, e seu desempenho geral frequentemente é limitado pela eficiência de seus componentes internos. Neste capítulo, estamos particularmente interessados em um de seus componentes específicos, responsável pelo gerenciamento e sincronização de estado em sistemas SMR.

Em uma arquitetura SMR, a *sincronização de estado* é um componente-chave que impacta o desempenho geral. Ela garante que todas as réplicas corretas mantenham um estado interno consistente. A sincronização de estado é crucial quando uma réplica entra no sistema, recupera-se de uma falha, ou precisa acompanhar réplicas mais rápidas. Como parte do procedimento de sincronização de estado, uma réplica recebe o estado da aplicação de uma ou mais réplicas operacionais. Muitas bibliotecas clássicas de replicação delegam à aplicação a responsabilidade de manter o estado (por exemplo, [12]). Como resultado, a própria biblioteca de replicação não é responsável por gerenciar o estado da aplicação, o que frequentemente leva a desempenho insatisfatório durante a sincronização de estado, especialmente na presença de falhas Bizantinas.

Neste capítulo, propomos uma mudança em como frameworks SMR lidam com gerenciamento e sincronização de estado, defendendo a integração de estruturas de dados avançadas aos sistemas SMR para aprimorar o gerenciamento de estado. Identificamos dois aspectos fundamentais que deveriam fazer parte dessas estruturas de dados: *clusterização* e *auto-validação*.

Estruturas de dados *clusterizadas*, nas quais os dados são agrupados ou particionados logicamente (isto é, fragmentados / sharded) para melhorar localidade e permitir verificação eficiente, são amplamente usadas em sistemas autenticados, concorrentes

e cache-aware [19, 46, 70, 96]. Essas estruturas são particularmente vantajosas em nosso cenário porque permitem que a integridade dos dados seja provada a qualquer momento, garantindo segurança mesmo sob condições adversariais. Além disso, diferentemente de sistemas tradicionais tolerantes a falhas Bizantinas (BFT), que exigem um quórum para validar leituras ou fornecer provas, estruturas clusterizadas podem operar com uma única réplica operacional, melhorando desempenho sem perder robustez.

Sob ameaças maliciosas, uma estrutura de dados clusterizada *auto-validável*, na qual os dados são particionados e cada cluster contém metadados suficientes para verificação independente de integridade, permite que réplicas verifiquem dados de forma concorrente. Réplicas honestas podem detectar rapidamente clusters inválidos, colocar uma réplica Bizantina em blacklist e buscar novamente clusters corrompidos de outra fonte. Em contraste, uma estrutura de dados comum só consegue detectar corrupção após baixar o estado inteiro, resultando em tempos de sincronização significativamente maiores e maior volume de transferência de dados.

Ilustramos nossa abordagem proposta introduzindo duas estruturas de dados novas: B+AVL tree e SVCSkipList. Uma B+AVL tree é uma estrutura de dados baseada em clusters inspirada por AVL* [41] e por B+Trees de Merkle, aproveitando suas vantagens enquanto evita suas limitações. Uma SVCSkipList é uma skiplist clusterizada auto-validável. Skiplists são estruturas de dados amplamente usadas para implementar armazenamentos chave-valor, com aplicações notáveis em sistemas como LevelDB (Google) [49], RocksDB (Facebook) [36] e Redis [87]. Uma de suas principais vantagens é a simplicidade de oferecer interfaces que suportam operações essenciais como busca, inserção e remoção, tornando-as ideais para uso no nível da aplicação. Implementamos ambas as estruturas de dados propostas e as avaliamos em diferentes configurações, mostrando que elas obtêm desempenho superior a abordagens baseline de estado da arte em diversos cenários, especialmente na presença de falhas Bizantinas.

4.1 Estruturas de dados auto-validáveis e clusterizadas

Uma estrutura de dados auto-validável é aquela que inclui inerentemente mecanismos ou metadados para verificar sua correção e integridade. Árvores de Merkle ou árvores balanceadas (AVL) similares (por exemplo, árvores Merkle-Patricia [94]) são estruturas auto-validáveis tipicamente usadas em blockchains. Além de prover acesso eficiente aos dados, essas estruturas capacitam réplicas a provar a integridade do estado para clientes. Em uma árvore de Merkle, cada folha armazena uma parte do estado e seu hash, e cada nó interno armazena o hash de seus filhos. Árvores de Merkle fornecem provas sucintas de integridade de dados para qualquer nó da árvore. Pode-se provar,

em tempo e espaço logarítmicos, que uma folha v pertence à árvore fornecendo o hash dos irmãos dos nós no caminho de v até a raiz da árvore.

Uma estrutura de dados clusterizada é um arranjo no qual elementos relacionados são armazenados juntos, tipicamente para otimizar padrões de acesso ou melhorar desempenho de cache (por exemplo, [19]). Isso é frequentemente obtido agrupando dados com base em atributos ou chaves específicas. Árvores de Merkle permitem que dados sejam clusterizados em pedaços menores (por exemplo, [41]), cada um dos quais pode ser validado independentemente. Isso significa que, durante a transferência de estado, a réplica receptora pode verificar a integridade dos dados recebidos checando hashes em diferentes níveis da árvore. Se qualquer parte dos dados tiver sido adulterada ou corrompida, ela pode ser identificada e isolada sem comprometer o estado como um todo. Esse mecanismo de auto-validação não apenas aumenta a confiabilidade da transferência de estado, como também melhora a eficiência do processo, pois apenas as porções inválidas do estado precisam ser retransferidas e revalidadas.

Além disso, estruturas clusterizadas podem melhorar significativamente a eficiência de serialização e desserialização. Em alguns sistemas de blockchain, por exemplo, entradas de uma estrutura em árvore são agrupadas em lotes, permitindo que o sistema serialize e desserialize esses lotes em vez de processar cada entrada individualmente. Essa abordagem de batching reduz overhead e acelera a sincronização de estado, levando a melhor desempenho geral. Peers em CometBFT, por exemplo, usam AVL+ trees (ver Seção 4.2.1) para gerenciamento de estado [27]. Um peer que serve um snapshot percorre a árvore, agrupa nós da árvore em clusters e serializa o cluster inteiro. Serializar um cluster contendo múltiplos nós da árvore é significativamente mais eficiente do que serializar cada nó individualmente. Essa eficiência decorre de agrupar todos os nós de um cluster em memória contígua, o que simplifica o processo de serialização.

Quando os nós são armazenados de forma contígua, o cluster inteiro pode ser serializado em uma única operação, eliminando a necessidade de iterar por nós individuais. Em contraste, se uma árvore binária é usada com nós espalhados na memória, cada nó precisa ser serializado separadamente, exigindo um loop para iterar por todos os nós. A Figura 4.1 compara o desempenho de serializar e desserializar uma árvore usando duas técnicas: dados clusterizados e itens individuais. Tanto a serialização quanto a desserialização se beneficiam significativamente de nós clusterizados, e a vantagem aumenta com o tamanho do cluster. A abordagem clusterizada supera a serialização e desserialização de nós individuais por um fator de 26× quando clusters contêm 4096 itens de dados, e por um fator de quase 6× quando clusters contêm 1024 itens de dados.

Embora a clusterização acelere a serialização e desserialização, ela introduz o overhead de construir clusters durante a transferência de estado. Uma alternativa mais

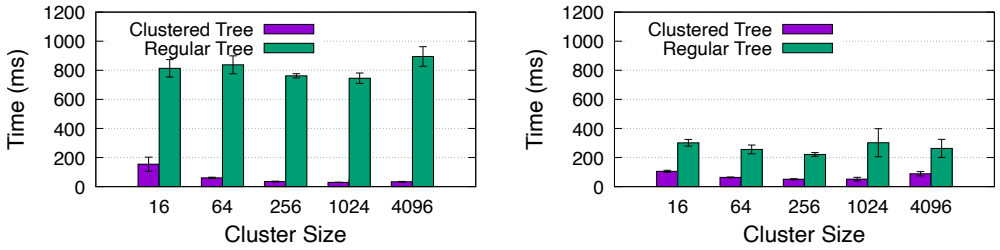


Figure 4.1. Tempo para serializar completamente (esquerda) e desserializar (direita) uma árvore clusterizada e uma árvore regular com 1 milhão de elementos, e vários tamanhos de cluster (cada elemento tem 512 bytes).

agressiva é usar estruturas de dados naturalmente organizadas em unidades de dados maiores, como AVL* trees [41] e B+Trees de Merkle [40], em que um peer que serve um snapshot pode evitar a travessia da árvore e a construção de clusters on-the-fly. Peers em recuperação também se beneficiam por não precisarem alocar nós individuais da árvore.

Além da eficiência de (de)serialização, estruturas clusterizadas oferecem outra vantagem crucial em cenários Bizantinos. Quando o estado é organizado em clusters verificáveis independentemente, clusters corrompidos podem ser detectados imediatamente ao serem recebidos, sem exigir o download da árvore inteira ou da estrutura completa. Isso permite detecção e rejeição precoce de dados inválidos, acelerando significativamente a recuperação. Em contraste, estruturas não clusterizadas frequentemente precisam baixar e processar a estrutura completa antes que a validação possa ocorrer, expondo sistemas a atrasos e ineficiências muito maiores diante de peers falhos ou maliciosos.

Começamos propondo uma estrutura de dados especializada no contexto de blockchains, um caso específico de sistemas de replicação de máquina de estados (SMR), usando uma estrutura do tipo árvore de Merkle. Em seguida, generalizamos essa abordagem estendendo-a para sistemas SMR mais amplos, onde buscamos integrar essas estruturas de dados avançadas para melhorar gerenciamento e sincronização de estado. Além disso, exploramos adaptar outros tipos de estruturas de dados além de árvores AVL para suportar clusterização e auto-validação.

4.2 B+AVL trees: sincronização eficiente de estado em blockchains

Nesta seção, abordamos a transferência de estado em sistemas de blockchain. De forma semelhante aos sistemas SMR genéricos, ela ocorre quando um peer se recupera de uma falha ou ingressa na rede blockchain. Para se atualizar em relação ao restante do sistema, o peer deve atualizar seu estado para corresponder ao dos peers operacionais. Sistemas modernos de blockchain (por exemplo, [27, 94]) utilizam snapshots periódicos do estado do sistema para melhorar o desempenho da transferência de estado. Um peer em recuperação primeiro baixa um snapshot recente e, em seguida, os dados subsequentes da blockchain (isto é, blocos completos ou cabeçalhos de blocos contendo resumos). O peer instala o snapshot e então reexecuta as transações que ocorreram após ele para reconstruir completamente o estado atual do sistema.

Em muitos sistemas de blockchain, os peers armazenam o estado em uma estrutura de dados especial, como uma árvore de Merkle [76] ou uma árvore Merkle-Patricia [94]. Nas AVL+ trees do Tendermint [1], cada nó folha armazena um hash criptográfico de seus dados, e cada nó interno armazena um hash calculado a partir do hash de seus filhos. Um peer em recuperação pode validar um snapshot computando o hash da raiz da árvore de Merkle e comparando-o com o hash da raiz armazenado no cabeçalho de bloco confiável.

Para reduzir o tempo necessário para que um novo peer ingresse na blockchain, um snapshot pode ser dividido em *clusters*, cada um contendo múltiplos nós da árvore, e um peer em recuperação pode baixar clusters de vários peers operacionais de forma concorrente. Variações dessa técnica foram implementadas tanto por sistemas de blockchain (por exemplo, [28]) quanto por sistemas de replicação de máquina de estados tolerantes a falhas Bizantinas (por exemplo, [11, 12, 18]). Algumas abordagens foram propostas para melhorar o desempenho e a robustez desse esquema [40, 41]. Construir clusters a partir da árvore de estado para servir um peer em recuperação e reconstruir nós da árvore a partir de clusters pode ser uma operação custosa, envolvendo travessia da árvore, serialização e desserialização dos nós. Esse custo pode ser substancialmente reduzido armazenando o estado do sistema em estruturas de dados "baseadas em clusters", como em AVL* trees [41] e B+Trees de Merkle [40]. Enquanto AVL* trees incorporam subárvores de estado em clusters, B+Trees de Merkle armazenam as folhas de uma célula em um cluster. Em ambos os casos, um peer que serve o estado a um peer em recuperação precisa apenas percorrer e serializar a lista de clusters existentes. O peer em recuperação recebe grandes unidades de dados, economizando recursos de forma semelhante ao batching [38], e evita alocar nós individuais da árvore ao depender de clusters.

Nesse contexto, introduzimos B+AVL trees, uma nova estrutura de dados baseada em clusters inspirada em AVL* e B+Trees de Merkle, aproveitando suas vantagens enquanto evita suas limitações. B+AVL trees são árvores binárias balanceadas que utilizam rotações, como árvores AVL, e integram hashes de Merkle para permitir validação individual de nós e subárvores, como em AVL* trees. Entretanto, em AVL* trees, manter o balanceamento por meio de rotações pode afetar as raízes dos clusters, frequentemente exigindo divisões de clusters antes ou depois das rotações, o que adiciona complexidade significativa. B+AVL trees, inspiradas em B+Trees, evitam essas complicações ao preencher clusters sequencialmente e dividi-los quando cheios, sem exigir rotações complexas entre clusters. Elas são mais eficientes em termos de espaço do que AVL* trees, permitindo que clusters armazenem mais dados, e fornecem provas de validação mais compactas do que B+Trees de Merkle.

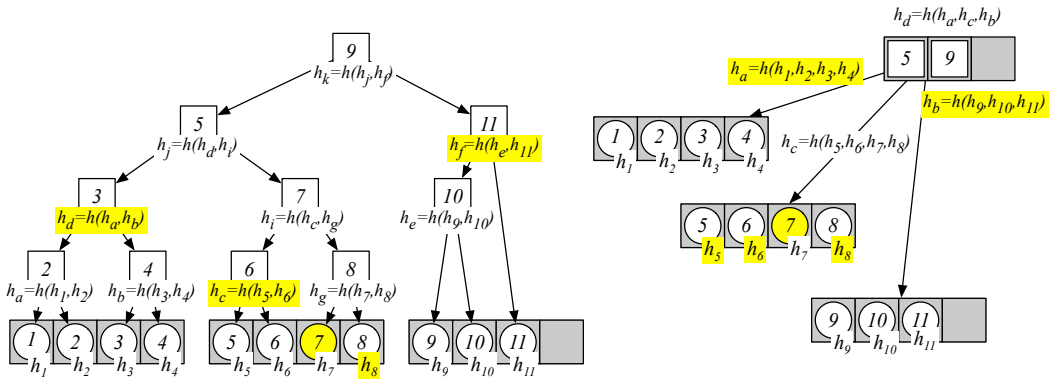
A Tabela 4.1 compara as principais propriedades das estruturas de dados para blockchain discutidas nesta seção. Ela destaca o tamanho das provas criptográficas (proof size) e se a estrutura suporta clusters auto-verificáveis. O tamanho da prova denota o número total de elementos (hashes) necessários para verificar a inclusão de uma entrada específica na árvore. A comparação considera AVL+ trees, AVL* trees, B+Trees de Merkle e B+AVL trees, onde n denota o número total de entradas na árvore, e b o número de entradas em uma célula de B-tree.

Estrutura de dados	**Tamanho da prova**	**Clusters auto-verificáveis**
AVL+ tree	$\mathcal{O}(\log_2 n)$	não
AVL* tree	$\mathcal{O}(\log_2 n)$	sim
Merkle B+Tree	$\mathcal{O}(\log_b n \cdot b)$	sim
B+AVL tree	$\mathcal{O}(\log_2 n)$	sim

Table 4.1. Estruturas de dados para blockchain (n: número de entradas, b: entradas em uma célula de B-tree).

4.2.1 AVL* trees e B+Trees de Merkle

AVL* trees estendem árvores AVL e AVL+ trees. Uma árvore AVL é uma árvore binária auto-balanceada em que as alturas dos filhos de um nó diferem em no máximo um. Quando esse balanceamento é violado (após inserções ou remoções), rotações o restauram, garantindo custo logarítmico para operações de busca, inserção e remoção. Inserções podem exigir até duas rotações; remoções podem exigir rotações proporcionais à altura da árvore. Uma AVL+ tree é uma variante em que apenas folhas armazenam valores; nós internos existem apenas em memória e podem ser recomputados a partir das folhas, reduzindo a necessidade de armazenamento persistente.



(a) AVL* tree (b) Merkle B+tree

Figure 4.2. Diferentes estruturas de dados usadas para armazenar estado em um peer de blockchain: (a) AVL* tree e (b) B+Tree de Merkle. (Círculos representam chaves e valores, quadrados brancos contêm chaves internas usadas para navegação, a área cinza representa clusters de dados.) A prova de que 7 é um elemento válido na AVL* tree inclui os hashes h_8, h_c, h_d e h_f, enquanto na B+Tree de Merkle a prova inclui os hashes h_5, h_6, h_8, h_a e h_b (destacados em amarelo).

AVL* trees [41] são AVL+ trees em que folhas são agrupadas em estruturas de dados maiores (isto é, clusters) para melhorar o gerenciamento e a sincronização de estado (ver Figura 4.2 (a)). Para fornecer uma prova de integridade de qualquer elemento de uma AVL* tree (isto é, folha), os nós da árvore são rotulados com hashes criptográficos, resultando no que é conhecido como uma árvore de hash ou árvore de Merkle. Em uma árvore AVL de Merkle, cada folha possui o hash de seu valor, e cada nó interno possui o hash de seus filhos. Na Figura 4.2 (a), os hashes destacados h_8, h_c, h_d, e h_f formam a prova necessária para verificar a inclusão da folha 7 na árvore. Esses hashes são mantidos atualizados conforme a árvore evolui e, para recuperá-los, um peer percorre o caminho da folha até a raiz, coletando os hashes dos irmãos ao longo do percurso. Durante a validação, um peer usa esses hashes para reconstruir o caminho até a raiz e compara o hash resultante com o hash de raiz confiável armazenado no cabeçalho de bloco correspondente. Se os hashes calculado e armazenado coincidirem, a folha 7 é confirmada como uma entrada válida. Uma prova em uma árvore AVL de Merkle tem tamanho $O(\log_2 n)$, onde n é o número de folhas da árvore.

B-trees generalizam árvores binárias permitindo que cada nó interno tenha entre $\lceil m/2 \rceil$ e m filhos, onde m é a ordem da árvore [61]. Todas as folhas estão no mesmo nível, e buscas, inserções e remoções têm complexidade logarítmica. B-trees são projetadas para eficiência de armazenamento e comunicação ao configurar o tamanho das células por meio da ordem da árvore. Em uma B+Tree, nós internos armazenam

cópias das chaves, enquanto folhas armazenam chaves e valores; folhas também podem incluir ponteiros para seus sucessores para acesso sequencial mais rápido.

Diferentemente das AVL* trees, que exigem algoritmos sofisticados para manter partes da árvore em clusters sem violar invariantes, B+Trees de Merkle são uma variação direta de B+Trees, estendida para fornecer provas de integridade [40]. Cada entrada em uma célula folha possui o hash de seu valor, e cada entrada em uma célula interna possui um hash calculado com base no hash de seus filhos. Na Figura 4.2 (b), a integridade da folha 7 pode ser verificada com os hashes h_5, h_6, h_8, h_a e h_b. Uma prova em uma B+Tree de Merkle tem tamanho $O(b \times \log_b n)$, onde n é o número de folhas da árvore e b o número de elementos em uma célula.

Discussão Tanto AVL* trees quanto B+Trees de Merkle melhoram o gerenciamento de dados e a sincronização de estado em blockchains ao incorporar clusterização diretamente na estrutura da árvore. Projetos baseados em clusters permitem serialização e desserialização eficientes dos dados, reduzindo overhead de comunicação. Outra vantagem é o uso de clusters *auto-verificáveis*: um peer em recuperação pode validar imediatamente um cluster ao recebê-lo verificando uma prova de raiz de subárvore embutida [41], sem reconstruir a árvore completa. Isso melhora significativamente o desempenho de recuperação na presença de peers maliciosos, que poderiam atrasar a sincronização ao fornecer dados inválidos (ver Seção 4.2.3).

4.2.2 A B+AVL tree

A B+AVL tree é uma estrutura de dados avançada projetada para armazenamento eficiente, busca e geração de provas sobre grandes coleções de pares chave-valor. Ela combina características de árvores de busca binária auto-balanceadas (como árvores AVL) e armazenamento de folhas em clusters (como em B+Trees), permitindo tanto atualizações eficientes quanto provas compactas. A B+AVL tree mantém balanceamento em dois níveis: a estrutura global da árvore e a organização interna dos clusters nas folhas.

Estrutura da B+AVL tree Uma B+AVL é uma árvore em dois níveis. Primeiramente, é uma árvore binária de busca auto-balanceada, em que cada nó interno contém os hashes de seus dois filhos e uma cópia da menor chave em sua subárvore direita (ver Figura 4.3). Diferentemente de uma AVL* tree, entretanto, os nós folha não hospedam espaço apenas para um único par chave-valor. Em vez disso, nós folha são implementados como nós folha de B+Tree, representando efetivamente um cluster de dados. Isso permite que clusters estejam em memória contígua para serialização eficiente. Quando

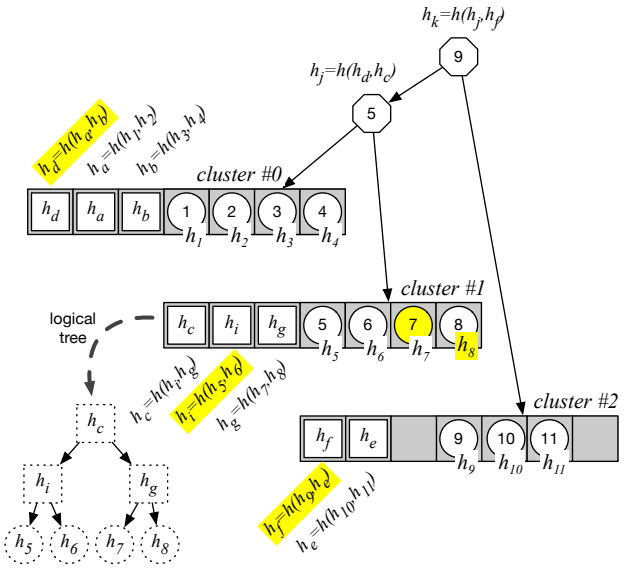


Figure 4.3. A B+AVL tree. A área cinza representa clusters de dados, círculos representam chaves, octógonos contêm chaves internas, quadrados brancos são hashes que constroem uma árvore lógica. A prova de que 7 é um elemento válido contém os hashes h_8, h_i, h_d e h_f.

um cluster fica cheio, ele é dividido e as duas metades são hospedadas em dois novos nós folha. Um novo nó interno é então adicionado para ligar as folhas.

Para evitar termos lineares no tamanho de uma prova para elementos individuais, que seriam introduzidos por clusters planos, cada cluster é aumentado com uma árvore de hash adicional *in-cluster*. Essa não é uma árvore de busca e seu único propósito é fornecer provas de tamanho logarítmico para elementos dentro de um cluster. Referimo-nos à raiz dessa árvore como a *raiz do cluster*. A raiz do cluster fornece os primeiros elementos da prova de hash para todo o cluster. Quando a prova do cluster e a prova de um elemento dentro do cluster são concatenadas, obtém-se uma prova de tamanho logarítmico para um elemento dentro da árvore como um todo.

Uma B+AVL tree realiza rotações em nós internos tratando as folhas que contêm clusters, independentemente de seu tamanho, como uma unidade. Assim como em uma árvore AVL, a altura de dois nós filhos pode diferir em no máximo 1. Entretanto, como um cluster pode variar entre meio cheio e cheio, a diferença de altura entre nós filhos na árvore interna do cluster também pode diferir em até 1. Consequentemente, após expandir clusters na visão lógica da árvore, a diferença de altura entre os filhos de qualquer nó em uma B+AVL tree pode ser de até 2, em comparação com a diferença máxima de 1 em uma árvore AVL. Isso, contudo, evita as rotações complexas entre clusters que uma AVL* tree precisa realizar para se manter balanceada.

Clusters da B+AVL tree Uma B+AVL tree representa um armazenamento chave-valor com chaves de tamanho fixo e valores de tamanho variável. Cada cluster contém três arrays: chaves, valores e hashes. O array de chaves possui tamanho conhecido e contém chaves ordenadas. Cada chave é aumentada com o tamanho e a posição inicial do valor correspondente no array de valores. Isso permite que novos valores sejam simplesmente anexados ao array de valores e limita a movimentação de dados durante inserções. Pré-alocamos um tamanho estimado para o array de valores e, se esse tamanho for excedido, o array é realocado. Para um cluster com M chaves, o array de hashes contém $2M-1$ entradas para representar os nós da árvore de hash interna do cluster em memória linear. A primeira metade desse array contém os hashes dos nós internos da árvore embutida no cluster, enquanto a segunda metade contém hashes diretos dos pares chave-valor. Graças ao mecanismo de divisão, é fácil ligar clusters consecutivos para simplificar consultas por intervalo.

Buscas e provas Para encontrar um valor, os nós internos são percorridos até que um nó folha seja alcançado. Durante essa travessia, uma prova do cluster pode ser obtida mantendo o registro dos valores de hash dos irmãos para cada nó encontrado. Dentro do cluster, o valor é recuperado por meio de uma busca binária no array de chaves. Como a árvore interna do cluster não é uma árvore de busca, a prova para um elemento específico não pode ser obtida durante a busca. Em vez disso, é necessário percorrer a árvore de hash interna do cluster de baixo para cima conhecendo a relação pai[1] de cada nó. Uma prova de um elemento individual é obtida concatenando a prova do cluster e a prova do elemento dentro do cluster. Por exemplo, na Figura 4.3, a integridade da folha 7 pode ser verificada com os hashes h_8, h_i, h_d, h_f e o hash de raiz no cabeçalho de bloco.

Reconstrução da B+AVL tree Para reconstruir a árvore, um peer deve requisitar todos os clusters, em qualquer ordem, e validá-los individualmente. Para isso, os arrays de chaves e valores são desserializados, e a árvore interna do cluster é recomputada pelo peer em recuperação. Em seguida, usando a raiz do cluster computada, a prova do cluster fornecida por um peer correto permitirá que o peer em recuperação assegure sua validade. Por exemplo, na Figura 4.3, o cluster #1 pode ser provado com os hashes h_d e h_f. Uma vez que todos os clusters sejam desserializados, eles precisam ser ordenados e a estrutura interna da árvore pode ser reconstruída. Como essa estrutura interna contém apenas hashes e cópias de chaves, sem quaisquer dados reais, e todos os dados já foram previamente verificados, a integridade da árvore reconstruída é garantida, eliminando a possibilidade de uma estrutura global inválida. Para assegurar que

[1] Para um array indexado a partir de 0, $\text{parent}(i) = \left\lfloor \frac{i-1}{2} \right\rfloor$.

a mesma estrutura de árvore seja obtida, observamos que todo nó interno da B+AVL tree contém uma cópia da menor chave no cluster mais à esquerda de sua subárvore direita. O nó interno encontra-se em uma determinada altura da estrutura da árvore e manter o registro dessa informação de altura da chave em todos os clusters é suficiente para codificar completamente e, portanto, reconstruir a estrutura da árvore interna. A mesma abordagem é usada nas AVL+ e AVL* trees.

4.2.3 Avaliação da B+AVL tree

Apresentamos agora detalhes técnicos sobre a implementação das estruturas de dados e o ambiente experimental. Em seguida, comparamos B+AVL trees com abordagens concorrentes considerando desempenho, tamanho de prova, eficiência de espaço e aspectos de sincronização de estado.

Implementação e ambiente Implementamos protótipos de todas as estruturas de dados em Go. Nossas implementações são baseadas na AVL+ tree do Tendermint [1], mantendo apenas funcionalidades essenciais. Utilizamos Amino (um subconjunto do Proto3) para serialização. A implementação da árvore B+AVL está publicamente disponível para apoiar reprodutibilidade e experimentação adicional.[2] Avaliamos quatro estruturas de dados: B+AVL, AVL*, B+Tree e uma implementação AVL+ baseline, doravante chamada Baseline, na qual clusters são preenchidos com folhas serializadas até o tamanho alvo, podendo atravessar subárvores e sem validação de provas.

Avaliamos as estruturas tanto em cenários standalone quanto distribuídos. Os testes standalone focam em desempenho de inserção/busca, tamanhos de prova e uso de espaço. No ambiente distribuído, avaliamos custos de sincronização de estado e resiliência a ataques Bizantinos. Um peer em recuperação utiliza um esquema semelhante ao usado em AVL* [41] para rastrear índices de clusters e detectar quando a árvore completa foi recuperada. Simulamos um ataque em que um peer malicioso corrompe clusters aleatórios. Nossos benchmarks comparam B+AVL com a árvore Baseline, que precisa refazer a transferência completa do estado após um ataque, destacando a eficiência da B+AVL nessas condições.

O ambiente experimental standalone é composto por uma máquina equipada com dois processadores AMD EPYC 7282 de 16 núcleos, 32 GB de RAM, executando Linux Ubuntu 18.04.6 e Go versão 1.20.2. Os experimentos distribuídos foram conduzidos no CloudLab [34], com uma configuração de 3 a 12 máquinas servidoras (peers ativos) e 1 máquina cliente (peer novo/em recuperação), todas interconectadas por uma rede comutada de 1 Gbps. Emulamos um ambiente WAN incorporando latências observadas

[2] https://github.com/MicheleCattaneo/B_plus_AVL

em regiões da AWS na Europa. As máquinas possuem oito processadores ARMv8 (Atlas/A57) de 64 bits, 64 GB de RAM, executando Linux Ubuntu 20.04 e Go versão 1.13.8.

Desempenho Nossos experimentos iniciais avaliam o desempenho da construção de diferentes árvores por meio de inserções incrementais e buscas em AVL* trees, B+Trees de Merkle, B+AVL trees e na árvore Baseline. Comparamos essas técnicas usando clusters de 256 elementos, variando o tamanho total da árvore. Em todos os experimentos, cada elemento tem 512 bytes, com chaves de 4 bytes.

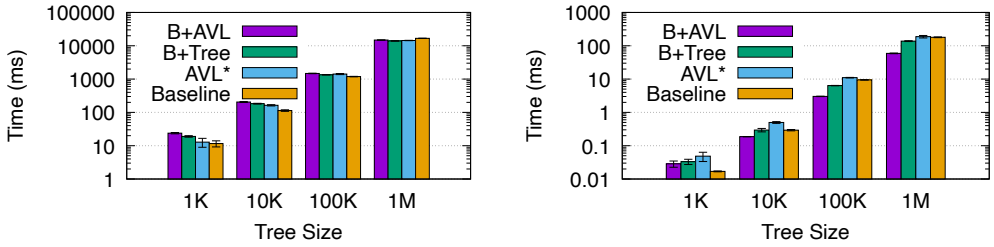


Figure 4.4. Desempenho das operações de inserção (esquerda) e busca (direita) para as quatro estruturas de dados, usando árvores com elementos de 512 bytes e clusters de 256 elementos, variando o tamanho da árvore. O eixo y (escala logarítmica) mostra o tempo médio para inserir todos os elementos ou buscar 10% deles. As barras indicam o desvio padrão.

A Figura 4.4 (esquerda) mostra o tempo requerido pelas quatro técnicas para construir árvores de cinco tamanhos diferentes, usando a mesma sequência de elementos de um conjunto de dados compartilhado. Para árvores menores (1K e 10K elementos), a B+AVL tree apresenta desempenho ligeiramente inferior às demais, seguida pela B+Tree, devido à necessidade da B+AVL de recomputar todos os hashes de um cluster após cada inserção. Embora otimizações no cálculo de hashes possam melhorar isso, elas aumentariam a complexidade. À medida que o tamanho da árvore cresce (100K e 1M elementos), as diferenças de desempenho entre as técnicas diminuem.

A Figura 4.4 (direita) mostra o desempenho de busca das quatro técnicas ao consultar 10% dos elementos em árvores de diferentes tamanhos. Todas as técnicas utilizam a mesma sequência de chaves. Para árvores pequenas, a árvore Baseline apresenta o melhor desempenho, enquanto para árvores maiores, a B+AVL tree supera as demais. Isso se deve ao layout de memória contígua dentro dos clusters, que melhora a utilização de cache ao minimizar faltas de cache durante operações de busca. Especificamente, usando a ferramenta `perf` do Linux para coletar contadores de desempenho de hardware, observamos que a B+AVL tree alcançou a menor taxa de cache miss, de

26.7%, seguida pela B+Tree com 29.0%. A AVL* tree apresentou uma taxa de 30.6%, e a árvore Baseline teve a maior taxa, de 33.1%.

Tamanho das provas A Figura 4.5 apresenta a distribuição dos tamanhos das provas para árvores com 1 milhão de elementos, variando o tamanho dos clusters. O tamanho da prova para cada elemento foi medido após a inserção de todos os elementos. A B+Tree de Merkle (inferior esquerdo) exibe um crescimento linear no tamanho das provas (ver também Figura 4.2 (b)). Esse crescimento ocorre porque provar um valor exige os hashes de todos os irmãos para cada nó. Consequentemente, à medida que o tamanho do cluster cresce, o número de hashes requeridos também aumenta, uma desvantagem significativa das B+Trees.

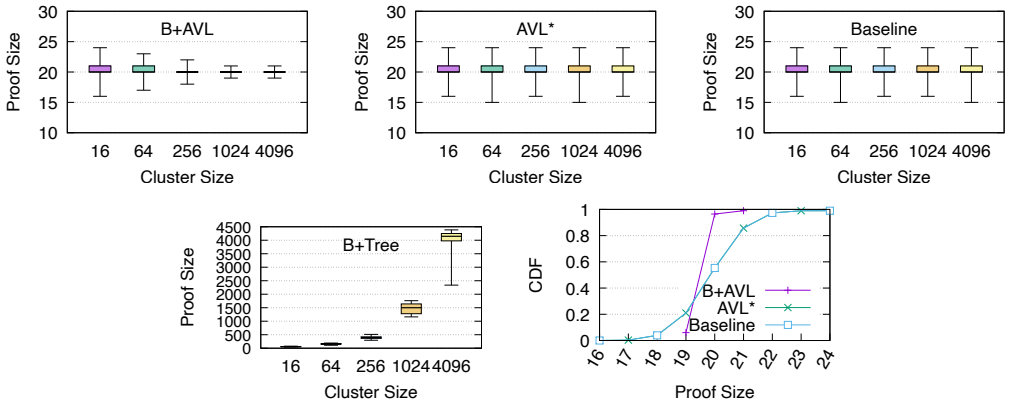


Figure 4.5. Proof sizes (boxplots) measured on a tree with 1 million elements after all insertions, varying the cluster sizes. Whiskers represent the minimum and maximum values. The CDF illustrates the distribution of proof sizes for the B+AVL, AVL*, and Baseline trees with a cluster size of 1024, revealing that B+AVL proofs have lower variance.

Em contraste, B+AVL, AVL* e a árvore Baseline exibem tamanhos de prova muito menores, que permanecem inalterados pelo tamanho do cluster. Além disso, a B+AVL tree demonstra uma característica desejável de menor variância no tamanho das provas à medida que o tamanho do cluster aumenta. Árvores AVL mantêm balanceamento garantindo que a diferença de altura entre os filhos de um nó seja no máximo um, enquanto o comprimento de uma prova é diretamente proporcional à profundidade das folhas, o que introduz alguma variância. Em contraste, B+AVL trees constroem subárvores dentro dos clusters com diferença máxima de profundidade de apenas um. Esse projeto resulta em menor variância no comprimento das provas ao considerar clusters individuais. À medida que o tamanho do cluster aumenta, essa estrutura balanceada

exerce maior influência no comprimento total da prova, conferindo à B+AVL tree uma vantagem distinta. Essa vantagem também é observada nas funções de distribuição acumulada mostradas na figura inferior direita, onde a B+AVL tree demonstra comprimentos de prova mais consistentes e menores em comparação com as AVL* e Baseline.

Eficiência de espaço No próximo conjunto de experimentos, avaliamos a eficiência de espaço de todas as técnicas. Definimos eficiência de espaço como a razão entre o número real de clusters da árvore e o número mínimo requerido em um cenário ideal. Por exemplo, uma eficiência de espaço igual a 2 indica que há o dobro de clusters do que seria necessário no caso ótimo. O pior caso para a B+Tree e a B+AVL tree ocorre quando todos os clusters estão meio cheios [24], levando a eficiência de espaço igual a 2. A árvore Baseline (não exibida nos gráficos) oferece a melhor eficiência de espaço, igual a 1. Portanto, focaremos nossa discussão nas três árvores restantes.

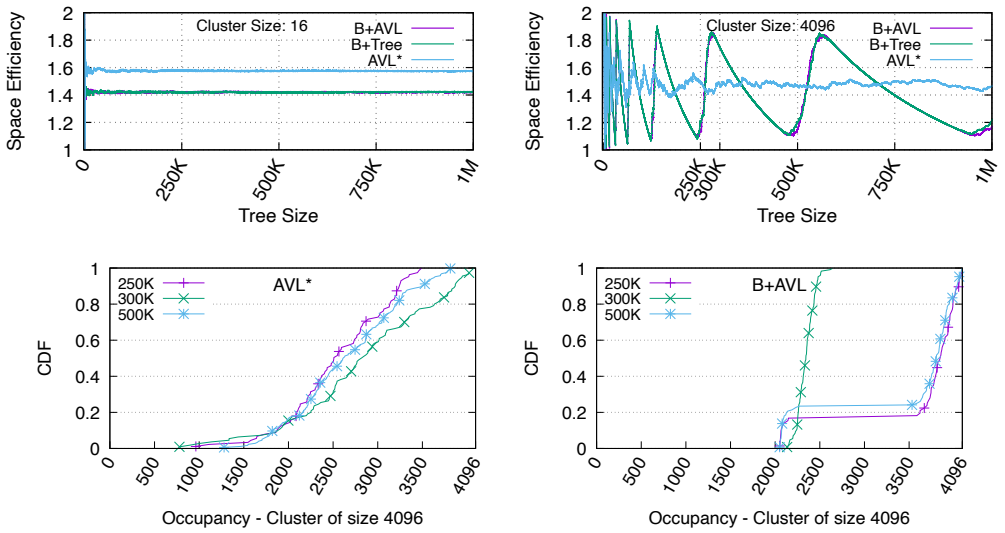


Figure 4.6. Eficiência de espaço após inserir cada elemento, com clusters de tamanho 16 (superior esquerdo) e 4096 (superior direito). O eixo x mostra o número de elementos na árvore, e o eixo y representa a eficiência de espaço. Funções de distribuição acumulada para a distribuição de ocupação dos clusters em três pontos distintos durante a inserção de elementos com clusters de tamanho 4096, para AVL* (inferior esquerdo) e B+AVL (inferior direito).

Na Figura 4.6, os dois gráficos na parte superior ilustram a eficiência de espaço após a inserção de cada elemento. O eixo x representa o número de elementos na árvore, enquanto o eixo y representa a eficiência de espaço. À esquerda, o tamanho do cluster é definido como 16, e à direita, como 4096. Observamos que, para tamanhos de cluster

	Tamanho do cluster				
Estrutura de dados	16	64	256	1024	4096
AVL*	1.58 (±0.01)	1.56 (±0.01)	1.50 (±0.02)	1.51 (±0.02)	1.48 (±0.04)
B+Tree	1.42 (±0.00)	1.43 (±0.00)	1.43 (±0.06)	1.41 (±0.17)	1.40 (±0.23)
B+AVL	1.42 (±0.00)	1.44 (±0.01)	1.44 (±0.03)	1.43 (±0.13)	1.41 (±0.24)

Table 4.2. Eficiência de espaço média (e desvio padrão) das três estruturas de dados com tamanhos de cluster variados.

pequenos, as três árvores exibem eficiência de espaço estável à medida que a árvore cresce, com a B+AVL e a B+Tree apresentando os melhores resultados. Entretanto, com tamanhos de cluster maiores, todas as árvores exibem eficiência de espaço mais instável conforme a árvore cresce, especialmente a B+AVL e a B+Tree.

Embora a eficiência de espaço média seja semelhante entre todas as árvores com tamanho de cluster 4096 (ver Tabela 4.2), a AVL* mantém uma eficiência de espaço mais consistente em comparação com a B+AVL e a B+Tree, que exibem maiores flutuações conforme os elementos são inseridos. Esse comportamento ocorre porque B+Trees tendem a preencher todos os clusters antes de aumentar sua altura, dado que os elementos são distribuídos uniformemente para inserção. Quando os clusters estão quase cheios (isto é, eficiência de espaço próxima de 1), inserções subsequentes tendem a causar divisões, reduzindo a eficiência de espaço. À medida que a maior parte dos clusters divide (isto é, eficiência de espaço se aproximando de 2), eles ficam meio cheios e voltam a ser preenchidos gradualmente. Esse problema não ocorre na AVL*, pois a altura da árvore não está tão fortemente ligada ao número de clusters.

Esse fenômeno também é observável nas duas funções de distribuição acumulada na parte inferior da Figura 4.6, onde medimos a distribuição de ocupação dos clusters em três pontos distintos durante a inserção de elementos com clusters de tamanho 4096. A AVL* (inferior esquerdo) exibe uma distribuição de ocupação mais uniforme, variando de meio cheio a cheio nos três pontos. Em contraste, a B+AVL (inferior direito) apresenta distribuições variáveis: com 300K elementos, conforme a eficiência de espaço se aproxima de 2 (ver o gráfico superior), a distribuição indica que a maioria dos clusters está quase meio cheia. Entretanto, com 250K e 500K elementos, quando a eficiência de espaço se aproxima do ótimo, a maioria dos clusters está quase cheia.

Sincronização de estado Nos experimentos distribuídos de sincronização de estado, comparamos B+AVL apenas com a árvore Baseline, pois ela reflete a abordagem amplamente utilizada na prática (por exemplo, a AVL+ do Tendermint), apesar de não possuir clusters auto-verificáveis e exigir retransmissão completa após ataques. Excluímos

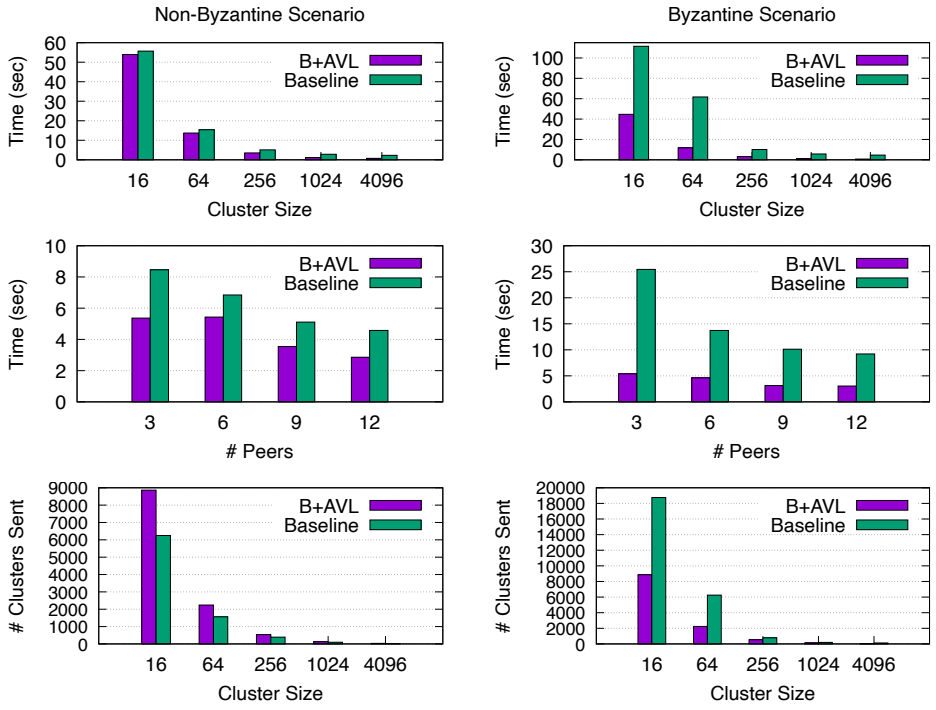


Figure 4.7. Tempo de sincronização de estado das árvores B+AVL e Baseline sem ataques (coluna esquerda) e com 1 peer Bizantino (coluna direita) para uma árvore de tamanho 100K: desempenho em função do tamanho do cluster com 9 peers (topo); desempenho em função do número de peers com clusters de 256 (meio); e o número de clusters enviados durante a sincronização conforme variamos o tamanho do cluster com 9 peers (base).

AVL* e B+Tree porque elas são estruturas standalone não usadas para sincronização de estado (B+Tree) ou empregam verificação de clusters semelhante à B+AVL (AVL*), oferecendo, assim, resiliência comparável, mas sem a eficiência de espaço e de provas da B+AVL.

A Figura 4.7 compara a B+AVL com a árvore Baseline, que constrói clusters simplesmente preenchendo-os enquanto percorre a árvore, resultando em clusters não auto-verificáveis. Os gráficos ilustram resultados de experimentos conduzidos em uma rede WAN emulada com uma árvore contendo 100K elementos. Os gráficos à esquerda mostram sincronização de estado apenas com peers honestos, enquanto os à direita incluem um peer Bizantino. Os gráficos superiores exibem o tempo (em milissegundos) necessário para transferir a árvore inteira, variando o tamanho dos clusters e com 9 peers servindo clusters. Os gráficos do meio ilustram o impacto de variar o número de peers servindo clusters com tamanho de cluster 256. Os gráficos inferiores mostram

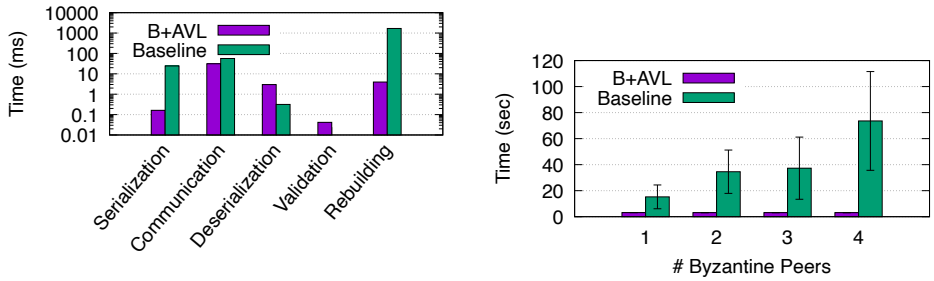


Figure 4.8. Esquerda: detalhamento de desempenho mostrando o tempo (ms, escala logarítmica) de cada fase da sincronização de estado com uma árvore de 100K elementos e clusters de 256 elementos. Direita: desempenho das árvores B+AVL e Baseline sob ataque Bizantino, com 100K elementos e clusters de 256 elementos, variando o número de peers Bizantinos.

o número de clusters enviados por cada técnica conforme o tamanho do cluster varia com 9 peers servindo clusters.

Observamos que, na ausência de ataques, a B+AVL apresenta desempenho ligeiramente melhor que a Baseline, apesar de enviar mais clusters. Essa vantagem é atribuída ao fato de que os tempos de serialização, comunicação e reconstrução (rebuild) da Baseline são maiores do que na B+AVL (Figura 4.8, esquerda). A serialização na árvore Baseline requer percorrer a árvore inteira para preencher os clusters. Na B+AVL, os clusters são diretamente acessíveis, permitindo serialização mais eficiente, pois a memória em cada cluster B+AVL é contígua. Consequentemente, a serialização pode ser realizada com uma única chamada por cluster, em comparação com a Baseline, que requer uma chamada para cada folha. Além disso, a memória contígua nos clusters da B+AVL melhora o desempenho de cache, particularmente para chaves, já que múltiplas chaves podem caber na mesma linha de cache. Esse uso eficiente de memória acelera ainda mais o processo de serialização (ver também a Figura 4.1). A reconstrução também é mais rápida na B+AVL devido à sua estrutura simplificada. Na B+AVL, há apenas um nó interno por cluster, enquanto na árvore Baseline há um nó interno para cada nó folha. Isso significa que a Baseline possui significativamente mais nós internos para reconstruir, tornando o processo mais lento. A desserialização, por sua vez, leva mais tempo na B+AVL, pois os valores de hash da árvore lógica dentro do cluster precisam ser computados para validar clusters individualmente. Em contraste, a Baseline requer apenas a desserialização de folhas individuais, tornando esse processo mais direto e menos intensivo em recursos.

Sob ataques Bizantinos, B+AVL trees permitem que peers verifiquem a integridade dos dados com overhead computacional mínimo. Suas provas pequenas possibilitam

verificação rápida e a vantagem dos clusters auto-verificáveis torna-se evidente. Peers honestos usando B+AVL trees podem detectar rapidamente dados inválidos, colocar o peer Bizantino em blacklist e refazer o download dos clusters corrompidos a partir de outra fonte. Em contraste, a árvore Baseline só consegue detectar corrupção após baixar a árvore inteira, levando a tempos de sincronização muito maiores e maior transferência de dados. A Figura 4.7 (direita) mostra que a árvore Baseline pode levar até 4× mais tempo e exigir o dobro de clusters em relação à B+AVL.

A lacuna de desempenho piora para a árvore Baseline conforme o número de possíveis peers Bizantinos aumenta, como mostrado na Figura 4.8 (direita), que ilustra o impacto de variar o número de peers Bizantinos até quatro. Com mais peers Bizantinos, a Baseline torna-se progressivamente mais ineficiente, levando a tempos de sincronização mais longos. Em contraste, a B+AVL mantém desempenho forte, demonstrando resiliência e eficiência superiores no gerenciamento de ataques Bizantinos. Curiosamente, em alguns casos, a B+AVL completa a sincronização de estado mais rapidamente em cenários Bizantinos do que em cenários não Bizantinos (por exemplo, Figura 4.7 no topo com tamanho de cluster 16). Isso pode ocorrer se o peer Bizantino estiver em uma região distante na rede. Uma vez colocado em blacklist, o peer passa a buscar clusters apenas de peers mais próximos, acelerando a sincronização. Isso evidencia a capacidade da B+AVL de gerenciar peers Bizantinos de forma eficaz e otimizar a seleção de peers, melhorando ainda mais os tempos de sincronização em certas condições.

4.3 SVCSkipList: sincronização eficiente de estado em SMR

Nesta seção, discutimos estruturas de dados clusterizadas auto-validáveis para aprimorar o gerenciamento e a sincronização de estado em sistemas SMR de propósito geral. Na seção anterior, introduzimos a B+AVL tree, uma estrutura de dados para permitir validação eficiente de estado em sistemas blockchain. Blockchains são uma instância específica de sistemas SMR, em que o estado é representado como uma sequência de blocos, e cada bloco contém um conjunto de transações. Agora estendemos esse conceito para um contexto mais geral, e propomos o uso de estruturas de dados clusterizadas auto-validáveis em sistemas SMR genéricos. Propomos uma nova estrutura de dados, a SVCSkipList, uma skiplist clusterizada auto-validável, e a integramos a um framework SMR bem conhecido, o BFT-SMaRt [12], para aprimorar suas capacidades de gerenciamento de estado. Nesta seção, discutiremos a adoção de tal estrutura, destacando seus potenciais benefícios e aplicabilidade para melhorar a resiliência e o desempenho de SMR.

4.3.1 Skiplists

Uma skiplist mantém um conjunto S de elementos ordenados organizados em uma sequência de listas encadeadas $S_0, S_1, S_2, \ldots, S_t$. O nível base S_0 contém todos os elementos de S em ordem crescente, juntamente com os elementos sentinela $-\infty$ e $+\infty$. Cada nível superior S_i, para $i \geq 1$, armazena um subconjunto dos elementos do nível imediatamente abaixo, S_{i-1}.

O método usado para selecionar elementos de um nível para o próximo define o tipo de skiplist. Na variante randomizada, cada elemento de S_{i-1} é promovido independentemente para S_i com uma probabilidade fixa p (tipicamente $p = 0.5$). Alternativamente, skiplists determinísticas [78] usam regras fixas para garantir que entre quaisquer dois elementos em S_i haja pelo menos um e no máximo três elementos em S_{i-1}.

Em ambas as variantes, os elementos sentinela $-\infty$ e $+\infty$ são sempre promovidos para níveis superiores, e o número de níveis t é mantido como $O(\log n)$. O nível superior S_t contém apenas os elementos sentinela; o nó que armazena $-\infty$ nesse nível é chamado de nó inicial s.

Um nó em S_{i-1} cujo elemento não aparece em S_i é chamado de *nó de platô* (*plateau node*), enquanto um nó cujo elemento também está presente em S_i é chamado de *nó de torre* (*tower node*). Consequentemente, entre quaisquer dois nós de torre no mesmo nível existe uma sequência de nós de platô. Em skiplists determinísticas, o número de nós de platô entre dois nós de torre é limitado entre um e três, enquanto em skiplists randomizadas o número esperado é um.

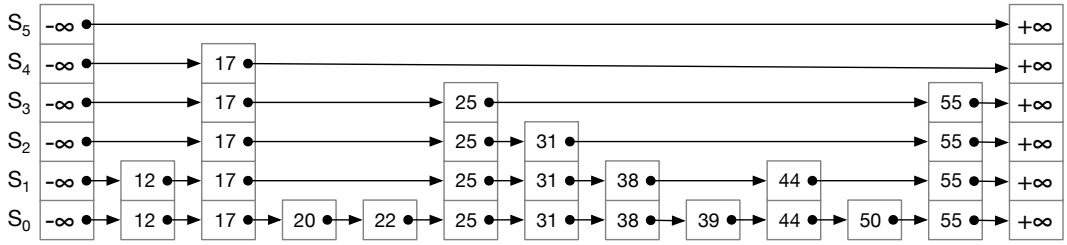


Figure 4.9. Exemplo de uma skiplist.

Cada nó em uma skiplist está associado a um elemento do conjunto e participa de dois tipos de relações estruturais (ver Figura 4.9). Verticalmente, nós correspondentes ao mesmo elemento são ligados entre níveis consecutivos, formando uma torre. Horizontalmente, nós dentro do mesmo nível são ligados em ordem crescente, permitindo a travessia ao longo daquele nível. Casos de fronteira são tratados com elementos sentinela, que encerram as travessias vertical e horizontal nos níveis mais baixo e mais alto, respectivamente.

As operações básicas de uma skiplist são as seguintes:

- **Busca**: Para buscar um elemento x, comece na lista do nível mais alto no nó mais à esquerda, mova-se para a direita até alcançar um nó com chave maior ou igual a x. Se for igual, retorne-o; caso contrário, desça um nível e continue até alcançar S_0.
- **Inserção**: Primeiro, realize uma busca para localizar a posição onde x deve ser inserido em S_0. Em seguida, insira x nessa posição em S_0 e promova-o para níveis superiores com probabilidade p, criando um novo nó e inserindo-o em cada nível percorrido durante a promoção.
- **Remoção**: Realize uma busca para localizar x. Em seguida, remova seus nós correspondentes de todos os níveis em que ele aparece, atualizando os ponteiros dos nós anteriores conforme necessário.

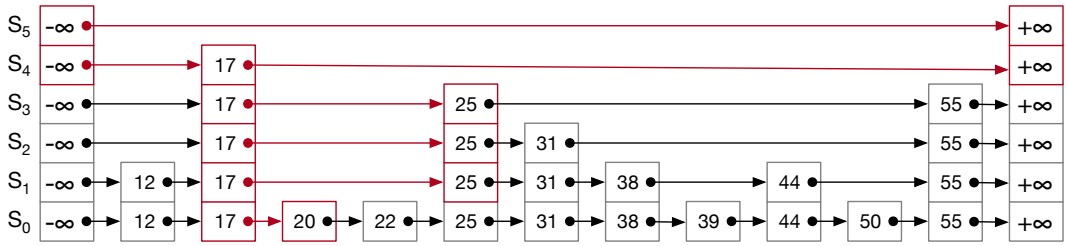


Figure 4.10. Busca pelo elemento 17 na skiplist da Figura 4.9. Os nós visitados e os links percorridos estão destacados em vermelho.

Em skiplists determinísticas, o procedimento de busca é garantido em tempo $O(\log n)$. Para skiplists randomizadas, é bem conhecido [47] que o mesmo procedimento executa em tempo esperado $O(\log n)$. Com alta probabilidade, a altura t da estrutura é $O(\log n)$, e o número esperado de nós visitados por nível é constante. Além disso, estudos experimentais [84] mostraram que skiplists randomizadas superam, na prática, estruturas de árvores balanceadas como árvores 2–3 e árvores rubro-negras. Na nossa implementação da SVCSkipList, detalhada mais adiante, adotamos um projeto de skiplist randomizada, pois ele oferece melhor desempenho e simplicidade em comparação com skiplists determinísticas, ao mesmo tempo em que mantém complexidade logarítmica para operações de busca, inserção e remoção.

Skiplists clusterizadas auto-validáveis Uma skiplist pode incorporar um esquema de hashing criptográfico resistente a colisões para permitir auto-validação (por exemplo,

[46]). Uma função hash resistente a colisões h garante que seja computacionalmente inviável encontrar duas entradas distintas com a mesma saída, assegurando assim a integridade dos dados. Em nossa abordagem, o hash de cada nó é computado recursivamente usando a função h (ver Figura 4.11(a)): se um nó n está no nível base, seu hash é computado diretamente como o digest de seus dados (chave e valor). Caso contrário, n é um nó interno, e seu hash é computado como o digest da concatenação dos hashes de todos os seus filhos imediatos no nível abaixo, que são os nós do nível inferior entre a chave de n e a chave de seu vizinho à direita. Portanto, o valor $h(S_0)$ armazenado no nó inicial S_0 é chamado de *hash raiz* (*root hash*), e serve como um digest que representa a skiplist inteira.

Embora esquemas de hashing sejam um mecanismo robusto para auto-validação de estruturas de dados, eles não atendem bem ao requisito de suportar sincronização de estado eficiente. No projeto apresentado em [46], por exemplo, a skiplist inteira é tratada como uma única entidade para validação, o que significa que qualquer sincronização de estado envolvendo essa estrutura exigiria transferir e validar todo o estado da skiplist. Essa abordagem, embora segura, é ineficiente ao lidar com estados grandes ou ao operar em ambientes propensos a falhas Bizantinas.

Defendemos que, no contexto de replicação de máquina de estados (SMR), uma estrutura de dados clusterizada auto-validável deve atender às seguintes restrições:

1. **Operações determinísticas**: Todos os procedimentos para manipular a estrutura de dados e manter clusters devem ser determinísticos, garantindo que as operações sejam executadas de forma consistente em todas as réplicas.

2. **Clusterização**: A estrutura de dados deve suportar um esquema de clusterização que agrupe dados em subconjuntos com base em critérios específicos e determinísticos, assegurando serialização/desserialização e transferência eficientes.

3. **Integridade estrutural**: Cada cluster deve preservar as propriedades essenciais da versão não clusterizada da estrutura de dados, garantindo que características operacionais fundamentais, como complexidade de tempo e espaço, sejam mantidas. Isso permite que operações dentro de um cluster espelhem aquelas do layout original, preservando correção e eficiência.

4. **Auto-validação**: A estrutura de dados como um todo, e cada um de seus clusters, deve ser auto-verificável, permitindo transmissão e validação independentes de clusters.

5. **Versionamento**: Em um sistema SMR, réplicas podem estar em estados diferentes devido a atrasos de rede ou diferentes velocidades de processamento. A

estrutura de dados deve incorporar versionamento para garantir consistência, com checkpoints periódicos permitindo que réplicas respondam com o mesmo estado, independentemente do progresso local.

Tais estruturas permitem que clusters sejam transferidos em paralelo e a partir de múltiplas réplicas durante uma sincronização de estado. Além disso, se uma falha Bizantina for detectada em um cluster, apenas aquele cluster específico precisa ser retransmitido, e não a estrutura inteira. Ao receber e validar todos os clusters necessários, a réplica receptora deve ser capaz de reconstruir a estrutura completa de forma determinística. Garantimos que o projeto da nossa SVCSkipList atende a essas restrições, conforme detalhamos nas seções a seguir.

Estratégia de clusterização Discutimos dois mecanismos para clusterizar uma skiplist: *particionamento estático* e *particionamento dinâmico*. Nas seções seguintes, examinamos as principais propriedades de cada abordagem, destacando suas respectivas vantagens e desvantagens, e discutindo como elas impactam desempenho, escalabilidade e a quantidade de espaço necessária para armazenar e gerenciar a estrutura (eficiência de espaço).

Particionamento estático: Uma estratégia comum de clusterização é o particionamento *estático* ou *por intervalos* (*range partitioning*). Nessa abordagem, uma skiplist pode ser dividida em segmentos fixos, gerenciados de forma independente. Cada cluster é uma skiplist independente auto-validável, com seu próprio hash raiz. Para representar a estrutura inteira, os hashes raiz dos clusters podem ser agregados em um único hash raiz usando uma árvore binária de hashes: hashes de clusters formam as folhas, e nós pais fazem hash recursivamente de pares de filhos até a raiz. Entretanto, uma desvantagem notável dessa abordagem é sua sensibilidade à distribuição do workload [70]. Ela funciona bem sob chaves uniformemente distribuídas, resultando em clusters balanceados e uso eficiente de espaço. Em contraste, workloads enviesados podem causar tamanhos de clusters e distribuição de dados desiguais, reduzindo a eficiência de espaço e potencialmente degradando o desempenho geral.

Particionamento dinâmico: Para lidar com as limitações associadas a técnicas de particionamento estático, exploramos o conceito de *particionamento dinâmico*. Nessa abordagem, os clusters são definidos selecionando um nível $l \geq 1$ de modo que os elementos no nível l definam as raízes dos clusters. Essa técnica particiona a skiplist em múltiplos clusters, cada um delimitado por elementos no nível l, como ilustrado na Figura 4.11. Consequentemente, o conjunto de nós no nível l serve como a lista de raízes de clusters, definindo efetivamente o conjunto de clusters. O número de nós nesse nível corresponde diretamente ao número total de clusters na skiplist. Essa abordagem também permite que a aplicação especifique aproximadamente o tamanho

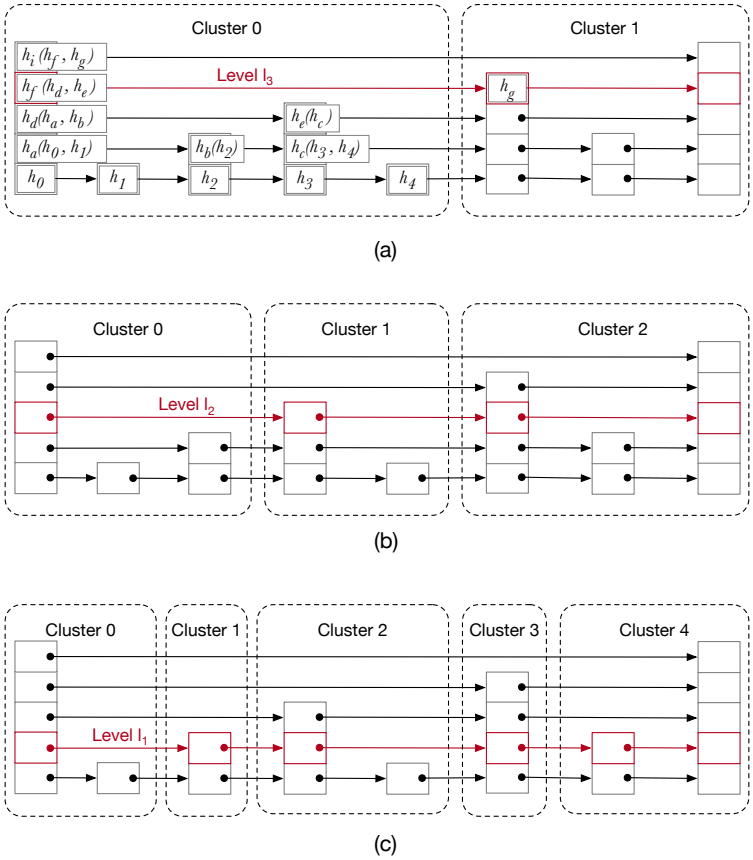


Figure 4.11. Skiplists clusterizadas com particionamento dinâmico. Em (a), usamos o cluster 0 para ilustrar como o hash do nó raiz (h_i) é computado como o hash da concatenação dos hashes de seus filhos. Esse processo se aplica a cada cluster e é realizado recursivamente em toda a skiplist.

desejado de cluster. Selecionar níveis mais altos resulta em clusters maiores (por exemplo, Figura 4.11(a)), enquanto escolher níveis mais baixos produz clusters menores (por exemplo, Figura 4.11(c)). Ela também suporta auto-validação ao computar um hash para cada raiz de cluster, permitindo validação independente de clusters.

Além disso, a clusterização dinâmica permite que clusters reflitam a distribuição real dos dados em vez de tamanhos fixos, melhorando o desempenho, especialmente quando algumas chaves são acessadas com maior frequência. Essa flexibilidade melhora recuperação, validação e adaptabilidade a workloads variados e conjuntos de dados em evolução.

Estratégia de versionamento Para implementar versionamento, copy-on-write pode ser usado ao augmentar cada nó da skiplist com metadados indicando sua versão de criação ou última modificação. A skiplist evolui com seu número de versão incrementado periodicamente, conforme detalhado mais adiante com o checkpointing das réplicas. Quando uma réplica na versão v modifica a skiplist e os nós afetados são de uma versão mais antiga $v' < v$, ela cria novas cópias desses nós (ou referências) rotuladas com v, preservando versões anteriores. Isso garante sincronização de estado determinística, já que réplicas podem progredir em ritmos distintos e manter visões diferentes do estado. Ao especificar a versão desejada durante a sincronização, uma réplica obtém um estado consistente, enquanto as réplicas de origem usam os metadados de versão para identificar e transmitir apenas elementos pertencentes à versão solicitada, garantindo correção e consistência.

4.3.2 SVCSkipList em detalhe

Apresentamos agora a SVCSkipList, nossa Self-Validating Clustered SkipList com um esquema de hashing e um mecanismo de clusterização para transferência e validação independentes de segmentos da skiplist. Essas melhorias endereçam as ineficiências da transferência de estado completa ao permitir que porções menores e verificáveis do estado sejam processadas separadamente. Apresentamos os principais aspectos, a implementação e alguns dos algoritmos por trás da SVCSkipList. Embora o particionamento estático ofereça simplicidade, suas limitações sob workloads enviesados nos levaram a adotar um esquema de particionamento dinâmico, que se adapta melhor às distribuições de chaves e garante clusterização balanceada e eficiente.

Definições preliminares O Algoritmo 6 define os componentes centrais da SVCSkipList. As variáveis globais acompanham parâmetros do sistema como um todo, tais como a versão atual v, o nível de cluster l (isto é, o nível que define as fronteiras dos clusters), a probabilidade de geração de nível dos nós p e a semente s. Nosso sistema requer que a probabilidade p e a semente s sejam compartilhadas entre todas as réplicas para preservar o determinismo do SMR. Ao usar a mesma probabilidade e semente para gerar níveis de nós, todas as réplicas constroem estruturas de skiplist idênticas para a mesma sequência de operações. Isso garante que a estrutura de dados permaneça consistente entre réplicas, mantendo a correção do estado replicado. A estrutura evolui ao longo de versões i, com arrays usados para rastrear diferentes versões dos sentinelas de cabeça e cauda (isto é, $head[i]$, $tail[i]$), bem como a altura $h[i]$ e o tamanho $n[i]$ (isto é, número de elementos) de cada versão. O nó $head[i]$ é a cabeça global da SVCSkipList na versão i, armazenando o *hash raiz* da estrutura para aquela versão, o que é essencial para auto-validar tanto a skiplist inteira quanto seus clusters individuais.

Algorithm 6 SVCSkipList Definições Gerais

```
1:  Variáveis globais:
2:      v : ℕ                                              {Versão atual}
3:      p : ℝ                                              {Probabilidade para geração de níveis}
4:      s : ℤ                                              {Semente para geração determinística de níveis}
5:      l : ℕ                                              {Nível de cluster para definir fronteiras dos clusters}
6:      ∀i : head[i] : Array⟨SVNode⟩                       {Nó cabeça na versão i}
7:      ∀i : tail[i] : Array⟨SVNode⟩                       {Nó cauda na versão i}
8:      ∀i : h[i] : Array⟨ℕ⟩                               {Altura da skiplist na versão i}
9:      ∀i : n[i] : Array⟨ℕ⟩                               {Número de elementos na versão i}
10: Cada SVNode possui:
11:     k : ℤ                                              {Chave}
12:     val : Bytes                                        {Valor}
13:     v : ℕ                                              {Versão do nó}
14:     hash : Bytes                                       {Hash deste nó}
15:     ∀i : left[i], right[i] : Array⟨SVNode⟩             {Links direcionais em}
16:     ∀i : up[i], down[i] : Array⟨SVNode⟩                {Versão i}
17:     ℓ : ℕ                                              {Nível do nó}
18:     del : Bool                                         {Marcador de remoção (true se logicamente removido)}
19: Cada Cluster possui:
20:     id : ℕ                                             {Identificador único do cluster}
21:     n_b : ℕ                                            {Número de nós do nível base}
22:     k_r : ℤ                                            {Delimitador de chave à direita}
23:     f : SVNode                                         {Primeiro nó no cluster}
24:     π : List⟨Hash⟩                                     {Caminho de prova para autenticação}
```

Os elementos na SVCSkipList são encapsulados em nós versionados auto-verificáveis (SVNode), que mantêm integridade entre versões por meio de links cientes de versão. Cada SVNode mantém uma chave k, um valor val e uma versão v, além de um hash usado para verificação. Os nós usam arrays para rastrear links em diferentes versões, com ponteiros direcionais (isto é, $left[i]$, $right[i]$, $up[i]$, $down[i]$) para cada versão i, e recebem um nível ℓ. Um flag de remoção del é usado para indicar a remoção lógica de nós. Remoção física não é um recurso crítico na avaliação da nossa estrutura de dados e não foi implementada em nosso protótipo. Em vez disso, remoções são implementadas marcando nós como logicamente removidos, com remoção física e coleta de lixo deixadas como melhorias futuras.

Cada Cluster encapsula um segmento contíguo da skiplist no nível base e contém metadados estruturais e de verificação. Entretanto, um objeto físico Cluster é instanciado apenas mediante uma requisição de sincronização de estado, seguindo o procedimento detalhado posteriormente no Algoritmo 8. Quando um objeto cluster é instanciado, o campo id identifica unicamente o cluster e corresponde ao seu índice

dentro da lista de clusters determinada pelo nível de cluster l. O campo n_b registra o número de nós do nível base contidos no cluster. O delimitador k_r marca a chave limite à direita, que define o intervalo de chaves do cluster. O ponteiro f referencia o primeiro nó do nível base do cluster. Por fim, π armazena a prova de autenticação, usada durante a verificação para assegurar a integridade do cluster independentemente do restante da estrutura.

As operações básicas para *buscar*, *inserir* e (logicamente) *remover* elementos na skiplist são implementadas com base nos algoritmos clássicos de skiplist [84], estendidos para suportar versionamento. A estrutura de dados inclui uma função para incrementar a versão atual, permitindo rastrear mudanças de estado ao longo do tempo. Réplicas produzem novas versões de forma síncrona em pontos de estado correspondentes aplicando conjuntos idênticos de comandos, o que garante consistência em todo o sistema. A operação de *inserção* usa os parâmetros previamente definidos, como a probabilidade p e a semente s, para atribuir níveis a novos nós de forma determinística. Se a chave dada já existe, seu valor associado é atualizado. Durante a inserção, um hash local também é computado serializando a chave e o valor, assegurando integridade no nível de elemento. Além disso, a operação de *inserção* adota a estratégia de copy-on-write descrita anteriormente para versionamento: quando um nó é criado ou modificado em uma nova versão, como ao atualizar seu valor após um incremento de versão, um novo objeto nó é criado para a nova versão contendo o novo valor, enquanto o objeto nó original com o valor anterior é preservado em sua versão anterior. Essa abordagem garante que cada versão mantenha um snapshot consistente e imutável do estado da skiplist no momento de sua criação. Tal imutabilidade será essencial mais adiante durante a sincronização de estado.

Além disso, a SVCSkipList fornece uma função recursiva para computar os hashes dos nós, permitindo verificação estrutural. Essa função pode ser invocada para computar o hash da skiplist inteira ou seletivamente sobre clusters individuais, como parte do processo de validação descrito nas seções seguintes.

Auto-validação Como mostrado no Algoritmo 7, definimos a função `computeHashes` para assegurar verificabilidade entre versões por meio do cálculo de hashes criptográficos sobre a skiplist versionada. A função inicia o cálculo a partir do nó cabeça da versão i e invoca `compHashRec`, que percorre recursivamente a skiplist de cima para baixo. Para nós internos, `compHashRec` coleta os hashes dos nós filhos dentro de um intervalo de chaves delimitado, concatena-os e computa um hash do valor combinado (como demonstrado na Figura 4.11(a)). Para nós folha, o hash é derivado diretamente da serialização da chave e do valor. Esse procedimento recursivo pode ser usado tanto para computar hashes de toda a skiplist a partir da cabeça global quanto para validar clusters individuais iniciando a recursão a partir do nó cabeça de um cluster. Essa

Algorithm 7 Cálculo Recursivo de Hashes

```
1:  Func computeHashes(i : ℕ):
2:      n ← head[i]                                   {Obtém o nó cabeça na versão i}
3:      compHashRec(n, i)                             {Cálculo recursivo}
4:  Func compHashRec(n : SVNode, i : ℕ) → Bytes:
5:      if n.down[i] ≠ ∅ then                         {Se n na versão i é um nó interno}
6:          if n.right[i] = ∅ then                    {Chave limite para os filhos}
7:              r ← +∞
8:          else
9:              r ← n.right[i].k
10:         aux ← n.down[i]                           {Começa do filho mais à esquerda}
11:         L ← []                                    {Lista para armazenar hashes dos filhos}
12:         while aux ≠ ∅ ∧ aux.k < r do               {Itera filhos no intervalo}
13:             L.append.(compHashRec(aux, i))        {Chamada recursiva}
14:             aux ← aux.right[i]                    {Move para o próximo irmão}
15:         n.hash ← H(⨁ L)                           {Hash concat. hashes dos filhos}
16:     else
17:         n.hash ← H(n.k, n.val)                    {Hash de nó folha}
18:     return n.hash
```

abordagem unificada permite verificação eficiente e consistente tanto no escopo global quanto no escopo de clusters.

Clusterização Para permitir sincronização de estado escalável e verificação modular, a SVCSkipList é particionada logicamente em *clusters*, com segmentos contíguos de nós do nível base agrupados com base em fronteiras estruturais e de versionamento.

O Algoritmo 8 detalha um procedimento para recuperar clusters da skiplist em uma versão específica v dentro de um intervalo de índices $[r_s, r_e]$. Esse intervalo corresponde aos índices dos clusters dentro da lista de clusters definida no nível l. A necessidade desse intervalo é explicada na próxima seção. A função principal, `getClusters`, itera sobre o intervalo, invocando `getCluster` para cada índice de cluster. A função `getCluster` instancia um cluster para a versão especificada v localizando primeiro o nó inicial do cluster por meio de `findClusterStart`, que recupera o nó raiz do cluster a partir da lista de clusters definida pelo nível l. Ela atualiza o delimitador de chave à direita k_r, que marca a fronteira do cluster na versão v, e gera uma prova criptográfica π para autenticar o conteúdo do cluster relativamente ao nó cabeça da skiplist naquela versão. Em seguida, a função identifica o primeiro nó do nível base dentro do cluster por meio de `findBaseNode`. Depois, conta o total de nós do nível base n_b até o delimitador à direita, todos na versão v. Por fim, o cluster totalmente inicializado é retornado.

Cada objeto `Cluster` possui uma função dedicada `genProof` para computar uma

Algorithm 8 Recuperar Clusters na Versão v

```
1:  Func getClusters(v : ℕ, r_s : ℕ, r_e : ℕ) → List⟨Cluster⟩:
2:      C ← []                                          {Lista de clusters}
3:      for i ← r_s to r_e do
4:          c ← getCluster(i, v)                        {Obtém cluster no índice i e versão v}
5:          C.append(c)
6:      return C
7:  Func getCluster(idx : ℕ, v : ℕ) → Cluster:
8:      c ← Cluster(idx)                                {Cria cluster com o id dado}
9:      n ← findClusterStart(head[v], l, idx)           {Encontra a cabeça do cluster}
10:     c.k_r ← n.right[v].k                            {Delimitador de chave à direita do cluster}
11:     c.π ← genProof(n, head[v], v)                   {Gera prova de autenticação}
12:     c.f ← findBaseNode(n, v)                        {Obtém o primeiro nó do nível base no cluster}
13:     c.n_b ← countNodes(n, v, c.k_r)                 {Conta nós do nível base}
14:     return c
```

prova de autenticação compacta π, que é uma lista de hashes, incluindo o conjunto mínimo de hashes dos níveis superiores necessário para autenticar a integridade do cluster e o *hash raiz* da SVCSkipList (a partir de $head[v]$) para a versão dada v. Partindo do nó delimitador do cluster, ela sobe a hierarquia da skiplist, identificando em cada nível os nós irmãos necessários que contribuem para o caminho de verificação do hash.

Cada `Cluster` também suporta serialização e remontagem, permitindo conversão de e para uma representação compacta adequada para transmissão. Além disso, cada cluster pode validar seu conteúdo usando as mesmas funções definidas no Algoritmo 7 para recomputar hashes locais e verificá-los em relação ao *hash raiz* esperado ao longo do caminho de prova.

Sincronização de estado com SVCSkipList Nossa SVCSkipList aprimora o protocolo de sincronização de estado de bibliotecas SMR ao permitir transferências baseadas em clusters eficientes, versionadas e autenticadas. O processo (detalhado no Algoritmo 9) começa quando uma réplica solicita uma sincronização de estado para uma versão específica v. A réplica de origem calcula seu intervalo de clusters, com base no seu id e no número total de clusters na versão v, recupera-os usando o mecanismo de extração de clusters (Algoritmo 8) e serializa a estrutura, os metadados e os dados de validação π de cada cluster em um formato compacto. Esses clusters serializados são transmitidos por canais especializados de sincronização de estado no framework SMR, estendidos para suportar nossa abordagem.

Ao receber, a réplica solicitante desserializa os dados recebidos para reconstruir cada cluster e o valida usando o procedimento descrito no Algoritmo 7. Especifica-

Algorithm 9 Procedimento de Sincronização de Estado

```
1:  Variáveis globais para a réplica r:
2:      r.receivedClusters : Set⟨Cluster⟩        {Conjunto de clusters válidos recebidos}
3:      r.totalClusters : ℕ                      {Total esperado de clusters}
4:      r.id : ℕ                                 {Id da réplica}
5:  Func stateSyncRequest(v : ℕ):
6:      sendSyncRequest(v)                       {Réplica solicita sync de estado para a versão v}
7:  Func processSyncRequest(v : ℕ):              {Ré plica de origem recebe req.}
8:      (r_s, r_e) ← clusterRange(v, r.id)       {Calcula intervalo de índices de clusters}
9:      C ← getClusters(v, r_s, r_e)             {Recupera clusters (Algoritmo 8)}
10:     σ ← serialize(C)                         {Serializa clusters e info de validação π}
11:     sendClusters(σ, v)                       {Envia clusters à réplica solicitante}
12: Func receiveClusters(σ : Bytes, v : ℕ):
13:     C ← deserialize(σ)                       {Desserializa clusters e info de validação π}
14:     for all c ∈ C do
15:         if validateCluster(c, v) then        {valida cada cluster separadamente}
16:             receivedClusters.add(c)          {Adiciona cluster válido ao conjunto}
17:         else
18:             refetchCluster(c.id, v)          {Refaz download de cluster inválido}
19:     if |receivedClusters| = totalClusters then   {Reconstrói a skiplist quando}
20:         rebuildSkiplist(receivedClusters, v)     {todos os clusters recebidos/válidos}
```

mente, ela computa os hashes do cluster usando a estrutura de prova e os valida contra o *hash raiz* da SVCSkipList fornecido em π, assegurando correção e prevenindo inconsistências. Clusters validados são acumulados em um conjunto de clusters recebidos. A réplica continua a refazer o download de quaisquer clusters inválidos até que todos os clusters esperados sejam recebidos e verificados com sucesso. Uma vez que o conjunto completo de clusters esteja disponível e validado, a réplica reconstrói a SVCSkipList integrando todos os clusters e restaurando suas conexões estruturais.

4.3.3 Integração ao BFT-SMaRt

Integramos a nossa SVCSkipList ao framework BFT-SMaRt [12]. Essa escolha é justificada pela abordagem modular do BFT-SMaRt, que apresenta um módulo de gerenciamento de estado bem definido, uma implementação baseada em Java e uma arquitetura multithread ciente de multicore que facilita replicação tolerante a falhas Bizantinas de alto desempenho. Para realizar essa integração, fizemos diversas modificações na base de código existente. Nesta seção, descrevemos as principais alterações.

Implementação da estrutura de dados Integramos nossa implementação da SVCSkipList à biblioteca BFT-SMaRt, expondo operações de manipulação de estado para a camada de aplicação. Nossa abordagem também realiza checkpoints; entretanto, enquanto a abordagem base do BFT-SMaRt cria um snapshot serializando e copiando todo o estado, o checkpointing em nossa abordagem consiste em criar uma nova versão simplesmente incrementando o número da versão e recomputando os hashes da versão anterior. Esse projeto não apenas reduz o overhead de checkpointing, mas também transfere o gerenciamento do estado da aplicação para a própria biblioteca de replicação, liberando os desenvolvedores de rastrear ou serializar manualmente o estado.

Protocolo de transferência de estado Modificamos o protocolo de transferência de estado para explorar a natureza clusterizada da SVCSkipList. Isso permite que as réplicas colaborem e transfiram concorrentemente diferentes intervalos de clusters, em vez de todo o estado, acelerando a sincronização de estado. Nossa implementação utiliza o procedimento descrito no Algoritmo 9. Além disso, estendemos o framework BFT-SMaRt para suportar a transmissão de estado por meio de um socket dedicado, desacoplando a comunicação de gerenciamento de estado dos canais de consenso e coordenação de réplicas. Essa melhoria beneficia tanto o módulo padrão de gerenciamento de estado do BFT-SMaRt quanto o nosso novo módulo baseado na SVCSkipList. Também tratamos uma limitação da implementação base do BFT-SMaRt que restringia o tamanho do estado a aproximadamente 2 GB devido ao tamanho máximo de arrays de bytes em Java. Ao particionar checkpoints em múltiplos arrays de bytes em vez de um único, o sistema passa a suportar estados de tamanho arbitrariamente grande.

Mecanismo de validação Implementamos um mecanismo que permite que réplicas verifiquem a integridade de clusters recebidos de forma independente, aumentando a tolerância a falhas e reduzindo a dependência do consenso para validação. Um framework SMR pode ser estendido de modo que as réplicas concordem durante o consenso sobre o *hash raiz* de cada versão da SVCSkipList. Esse hash raiz pode então ser compartilhado de forma segura entre as réplicas como parte do protocolo de sincronização de estado. Alternativamente, uma réplica pode iniciar a sincronização de estado solicitando o hash raiz a outras réplicas e aguardando o recebimento de $f + 1$ valores idênticos, assegurando consistência sem modificar o protocolo de consenso. Durante a sincronização de estado, os clusters são validados incrementalmente à medida que chegam, conforme detalhado no Algoritmo 9. Essa abordagem contrasta com o mecanismo base do BFT-SMaRt, que realiza a validação apenas após todo o estado ter sido completamente transferido, exigindo uma nova transferência completa caso qualquer corrupção seja detectada.

Opções de configuração Adicionamos opções de configuração para permitir que usuários especifiquem parâmetros como o nível de cluster da SVCSkipList, a probabilidade de promoção e a semente. Adotamos essa abordagem de configuração estática por simplicidade; entretanto, projetar um protocolo leve para que réplicas concordem sobre uma semente comum — ou até mesmo a rotacionem periodicamente — é direto. De forma semelhante, a capacidade de suportar reconfiguração dinâmica do nível de cluster, adaptando o tamanho dos clusters dinamicamente de acordo com as demandas da aplicação, é uma funcionalidade poderosa habilitada por nossa estrutura de dados.

Essas modificações foram projetadas para garantir que a nossa SVCSkipList pudesse ser integrada de forma transparente ao framework BFT-SMaRt existente, ao mesmo tempo em que oferece desempenho aprimorado e maior tolerância a falhas por meio de suas capacidades de gerenciamento de estado clusterizado e auto-validável.

4.3.4 Avaliação da SVCSkipList

Justificativa Comparamos o desempenho da nossa SVCSkipList com uma técnica de base (isto é, BFT-SMaRt) em implantações SMR típicas: réplicas com modos de falha independentes (por exemplo, diferentes zonas de disponibilidade [6]) dentro da mesma região geográfica. A avaliação experimental busca responder às seguintes questões:

1. Como as técnicas se comparam em condições normais (isto é, execução de requisições sem sincronização de estado)?

2. Qual é o custo de checkpointing em cada abordagem?

3. Quanto tempo leva para sincronizar o estado sem ataques?

4. Como o tamanho dos clusters impacta a sincronização de estado?

5. Como a sincronização de estado impacta a execução?

6. Quanto tempo leva para sincronizar o estado sob ataques?

Nosso objetivo é entender como as técnicas se comportam sob diferentes workloads, tamanhos de estado, números de réplicas e tamanhos de clusters. A Tabela **??** resume o espaço de parâmetros considerado em nossa avaliação.

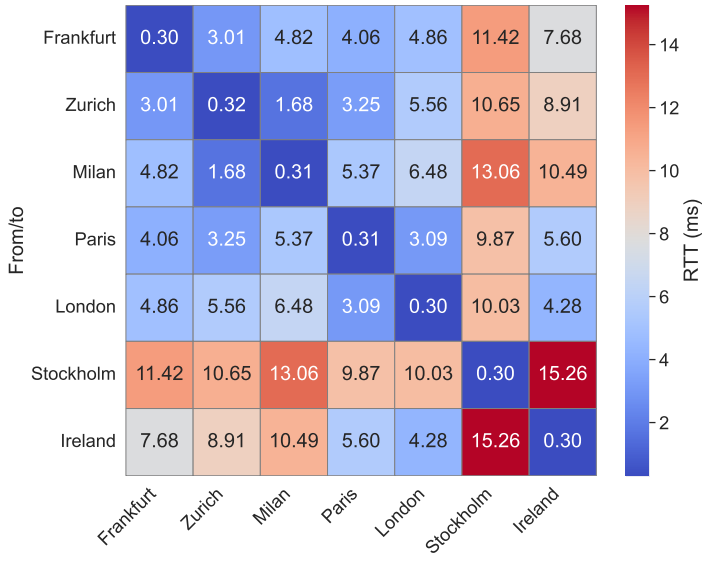


Figure 4.12. Average round-trip time (RTT in milliseconds) between pairs of availability zones used in our emulated public cloud environment.

Preliminares Antes de apresentar os resultados da avaliação, descrevemos o ambiente experimental utilizado para avaliar o desempenho e a resiliência da nossa abordagem. Detalhamos o ambiente de hardware, a configuração de nuvem pública emulada, a aplicação e a configuração de clientes e réplicas. Também descrevemos parâmetros-chave de execução e configurações específicas do framework BFT-SMaRt para garantir reprodutibilidade e relevância para cenários realistas de implantação.

Hardware Todos os experimentos foram conduzidos em um cluster privado com máquinas executando Ubuntu 18.04.6 LTS, com 32 GB de RAM e um processador AMD EPYC 7282 de 16 núcleos, totalizando 32 threads de hardware (16 núcleos com 2 threads por núcleo). A CPU opera em modo 64 bits e suporta frequência máxima de 2.8 GHz.

Nuvem pública emulada Nesse ambiente, emulamos a largura de banda e a latência de instâncias Amazon EC2 M5 (`m5.large`), uma opção padrão e de baixo custo para implantações práticas, distribuídas em sete zonas de disponibilidade na Europa [6]. A largura de banda máxima de rede entre duas máquinas foi configurada para 1 Gbps. Os padrões médios de latência entre máquinas são mostrados no mapa de calor da Figura 4.12[3] e configurados com 5% de jitter.

Aplicação Implementamos um armazenamento simples de chave–valor inspirado

[3] https://www.cloudping.co/

no YCSB [25], utilizando a interface de aplicação do lado do servidor do BFT-SMaRt. As chaves são inteiros e os valores são arrays de bytes aleatórios de tamanho 1 KB. Para controlar o tamanho inicial do estado, a skiplist é pré-populada com um número suficiente de pares chave–valor para atingir o tamanho de estado desejado, variando de 1 GB a 4 GB dependendo do cenário experimental. As chaves são inseridas sequencialmente em ordem crescente até que o tamanho alvo do estado seja alcançado. Durante os experimentos, os clientes acessam as chaves seguindo uma distribuição uniforme.

A mesma aplicação foi utilizada tanto para a SVCSkipList quanto para o baseline, garantindo uma comparação justa. A principal diferença reside no gerenciamento de estado: o baseline utiliza o mecanismo padrão do BFT-SMaRt, o que significa que a skiplist é gerenciada inteiramente pela aplicação e totalmente serializada/desserializada durante o checkpointing e a instalação do estado. Em contraste, nossa abordagem integra a SVCSkipList diretamente a um novo módulo de gerenciamento de estado (Seção 4.3.3), explorando suas características de clusterização e auto-validação.

Clientes Implementamos processos clientes usando a interface de proxy de clientes do BFT-SMaRt para gerar um workload misto de leituras e escritas. Os clientes operam em modo de laço fechado, emitindo cada nova requisição apenas após receber a resposta da requisição anterior. Todos os clientes e réplicas foram distribuídos uniformemente pelas zonas emuladas descritas anteriormente.

Réplicas Cada réplica pré-inicializa sua SVCSkipList de forma determinística, garantindo que o sistema inicie com um estado inicial consistente. Como já mencionado, consideramos tamanhos de estado de 1 GB, 2 GB e 4 GB, nos quais um número proporcional de elementos de 1 KB foi inserido na skiplist durante a inicialização. Os experimentos foram conduzidos usando configurações de servidor com quatro réplicas (capazes de tolerar até uma falha Bizantina) ou sete réplicas (capazes de tolerar até duas falhas Bizantinas). Os experimentos com quatro réplicas foram implantados em Frankfurt, Zurique, Milão e Paris (ver Figura 4.12), enquanto os experimentos com sete réplicas abrangeram todas as zonas consideradas. Para simular falhas Bizantinas, uma ou duas réplicas foram deliberadamente configuradas para corromper dados durante a sincronização de estado, emulando comportamento malicioso.

Execuções As execuções experimentais variaram de 1 a 3 minutos, dependendo do cenário específico. Dados iniciais de aquecimento e períodos finais de término foram descartados para garantir que apenas o desempenho em regime estacionário fosse analisado.

Parâmetros do BFT-SMaRt O framework BFT-SMaRt foi configurado para operar em modo tolerante a falhas Bizantinas, com os mecanismos de log e checkpoint utilizando armazenamento em memória em vez de persistência em disco, para reduzir o overhead de E/S durante os experimentos. A maioria dos parâmetros de configuração foi mantida em seus valores padrão para preservar o comportamento usual; entre-

tanto, alguns foram ajustados para habilitar suporte a grandes estados de aplicação. Em particular, a frequência de checkpoint foi ajustada com base no tamanho do estado testado, garantindo operação eficiente e evitando consumo excessivo de memória ou atrasos excessivos de checkpoint durante os experimentos.

Desempenho em condições normais Para avaliar o desempenho do sistema em condições normais, variamos o número de clientes concorrentes emitindo operações ao serviço replicado, com o objetivo de identificar pontos de saturação e avaliar como nossa estrutura lida com alta contenção. Medimos throughput e latência com um estado de aplicação de 1 GB, comparando a SVCSkipList com o baseline original do BFT-SMaRt sob workloads *read-most* (RM) e *update-heavy* (UH).

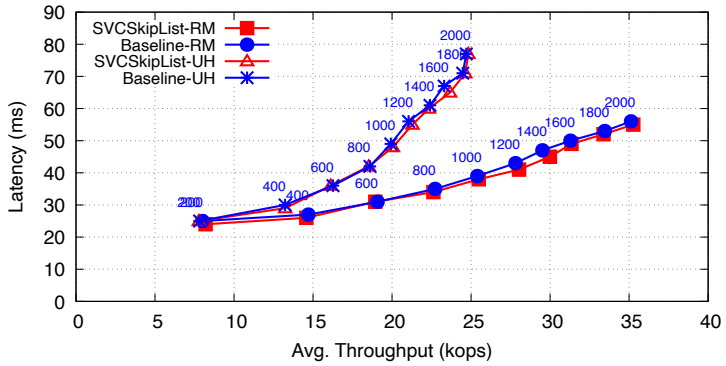


Figure 4.13. Vazão versus latência em diferentes workloads e quantidades de clientes, tanto para a SVCSkipList quanto para a abordagem Baseline.

A Figura 4.13 apresenta a vazão (em kops, milhares de operações por segundo) e a latência média (em milissegundos). À medida que o número de clientes cresce de 200 para 2000, ambas as técnicas escalam de forma efetiva, mostrando um aumento consistente de vazão. Por exemplo, no workload RM a SVCSkipList atinge desempenho comparável ao Baseline, com diferenças tipicamente abaixo de 2% mesmo em cargas mais altas. Esses resultados indicam que nossa abordagem mantém desempenho comparável ao Baseline entre os workloads, com pequenas vantagens devido ao menor overhead de checkpointing, o que ajuda a reduzir o custo médio de processamento por requisição.

Com base nesses dados, calculamos os pontos de maior potência do sistema, isto é, onde a razão entre vazão e latência é máxima. Isso garante que nossa avaliação reflita o ponto operacional de melhor eficiência em termos de custo de processamento por requisição. Embora isso não represente a vazão máxima absoluta do sistema, indica o ponto de inflexão em que o sistema atinge seu pico de vazão efetiva antes que as

				Vazão (kops)	**Latências** (ms)				
WL	**R#**	**Algoritmo**	**Estado**	Méd.	Méd.	50^{th}	99^{th}	99.9^{th}	99.99^{th}
RM	4	SVCSkipList	1GB	26.0	41.9	39.0	81.0	646.0	1016.0
	4	Baseline	1GB	25.0	43.0	40.0	67.0	1481.0	1820.0
	4	SVCSkipList	2GB	25.5	43.6	40.0	67.0	1335.0	1788.0
	4	Baseline	2GB	24.2	47.4	41.0	70.0	2872.0	3450.0
	4	SVCSkipList	4GB	23.1	44.7	39.0	68.0	2440.0	3710.0
	4	Baseline	4GB	16.9	60.4	41.0	309.0	6403.0	6419.0
UH	4	SVCSkipList	1GB	16.1	36.2	35.0	54.0	701.0	1129.0
	4	Baseline	1GB	16.2	36.6	35.0	53.0	1489.0	1828.0
RM	7	SVCSkipList	1GB	9.0	132.5	129.6	198.3	849.3	929.3
	7	Baseline	1GB	8.8	135.0	130.6	197.3	1538.3	1901.3
	7	SVCSkipList	2GB	8.8	134.0	129.6	204.0	1604.3	1698.3
	7	Baseline	2GB	8.4	144.2	129.6	229.0	3486.3	3562.0

Table 4.3. Vazão e latência durante a execução normal em várias configurações no ponto de maior potência (isto é, razão máxima entre vazão e latência). Melhores valores de vazão e latência para cada configuração em azul. A SVCSkipList supera o Baseline em quase todos os casos.

latências comecem a aumentar significativamente devido a efeitos de enfileiramento. Assim, os pontos usados em todos os experimentos subsequentes foram escolhidos da seguinte forma: para o workload RM, 1200 clientes (onde a vazão era em torno de 28 kops com latência média de 41–43 ms), e para o workload UH, 600 clientes (onde a vazão era em torno de 16 kops com latência média de 36 ms).

Impacto do workload A Tabela 4.3 apresenta a vazão e a latência para ambos os workloads usando um estado de 1GB, durante a execução normal. Os resultados mostram que, para o workload RM, a SVCSkipList obtém uma latência média ligeiramente menor (41.9 ms) em comparação ao Baseline (43.0 ms), com latências medianas semelhantes; entretanto, ela apresenta uma melhoria notável em percentis mais altos. Em particular, embora ambos os sistemas tenham latências de percentil 99 comparáveis (81.0 ms vs. 67.0 ms), a SVCSkipList apresenta latências de cauda substancialmente menores nos percentis 99.9 e 99.99 (646.0 ms e 1016.0 ms) em comparação ao Baseline (1481.0 ms e 1820.0 ms), devido ao seu menor overhead de checkpointing (ver Figura 4.14). No workload UH, as latências média e mediana são praticamente idênticas entre as duas abordagens; contudo, a SVCSkipList novamente reduz significativamente as latências de cauda. No percentil 99.9, a SVCSkipList reporta 701.0 ms versus 1489.0 ms no Baseline, e no percentil 99.99, 1129.0 ms em comparação a 1828.0 ms.

Impacto do tamanho do estado A Tabela 4.3 também mostra o impacto do au-

mento do tamanho do estado no desempenho do sistema durante a execução normal. À medida que o estado da aplicação cresce de 1 GB para 4 GB, com quatro réplicas e o workload RM, a vazão tanto da SVCSkipList quanto do Baseline diminui gradualmente. Entretanto, a queda é notavelmente mais acentuada no Baseline: enquanto a SVCSkipList sustenta níveis de vazão próximos de 26 kops até 2 GB e cai apenas para cerca de 23 kops em 4 GB, a vazão do Baseline cai de 25 kops em 1 GB para apenas 16.9 kops em 4 GB. Isso sugere que nossa abordagem lida com estados maiores de forma mais eficiente, novamente devido ao menor overhead de checkpointing.

De forma semelhante, os resultados de latência mostram que a SVCSkipList mantém latências média e mediana mais estáveis à medida que o tamanho do estado aumenta. Por exemplo, a latência média do Baseline aumenta de 43.0 ms para 60.4 ms quando o estado cresce de 1 GB para 4 GB, enquanto a SVCSkipList cresce apenas modestamente, de 41.9 ms para 44.7 ms. Além disso, as latências de cauda do Baseline se degradam conforme o estado cresce, enquanto a SVCSkipList as mantém significativamente menores. No geral, esses resultados indicam que nossa estrutura escala melhor com o tamanho do estado, oferecendo maior estabilidade tanto em vazão quanto em latência no pior caso à medida que o estado do sistema cresce.

Impacto do checkpointing A Figura 4.14 apresenta os tempos de checkpointing para ambas as técnicas em diferentes tamanhos de estado, com quatro réplicas e o workload RM, reportados em segundos. Os resultados demonstram que o esquema de multiversionamento da SVCSkipList supera consistentemente os checkpoints do Baseline. Por exemplo, com um estado de 1 GB, a SVCSkipList leva aproximadamente 0.87 s versus 1.64 s do Baseline, reduzindo quase pela metade. À medida que o tamanho do estado aumenta, os benefícios se tornam mais evidentes: para 2 GB, a SVCSkipList leva 1.66 s em comparação a 3.44 s no Baseline, enquanto para 4 GB, o tempo de checkpointing é 3.24 s contra 6.74 s. Essa redução decorre da abordagem de multiversionamento copy-on-write mais eficiente utilizada na SVCSkipList, que evita o overhead de cópia de dados durante a criação de checkpoints. Ela apenas incrementa o número da versão e recomputa hashes, em vez de serializar todo o estado de uma vez, como na abordagem Baseline.

Sincronização de estado sem ataque Agora avaliamos os tempos de sincronização de estado para as técnicas SVCSkipList e Baseline, medidos como o tempo necessário para sincronizar todo o estado da aplicação entre as réplicas em um cenário benigno (sem ataques). Como a réplica que executa a sincronização de estado neste caso é aquela que sofreu uma falha e está retornando ao sistema, referimo-nos a ela como a réplica *em recuperação*.

Para cada configuração, executamos três experimentos independentes para coletar dados e calcular os valores médio, mínimo e máximo. A Tabela 4.4 apresenta dois

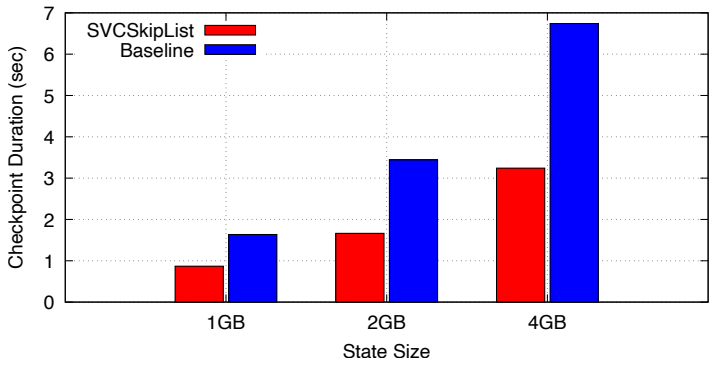


Figure 4.14. Tempo médio para criar um checkpoint para ambas as técnicas sob o workload RM.

		Sem ataque			Sob ataque		
R#	**Algoritmo**	**Méd.**	**Min**	**Máx**	**Méd.**	**Min**	**Máx**
4	SVCSkipList	36.3	35.8	37.3	36.1	35.5	36.6
4	Baseline	39.3	37.9	41.8	51.5	48.9	56.3
7	SVCSkipList	35.7	31.3	39.5	37.3	32.7	40.2
7	Baseline	44.2	39.2	51.7	103.2	90.0	121.0

Table 4.4. Tempo de sincronização de estado (em segundos) com e sem ataques. Melhores resultados para cada configuração em azul.

tamanhos de grupo: 4 réplicas e 7 réplicas. Para a configuração com 4 réplicas, a SVCSkipList alcançou um tempo médio de sincronização de aproximadamente 36.3 segundos, enquanto o Baseline exigiu cerca de 39.3 segundos, com tempos variando entre 37.9 e 41.8 segundos, ao passo que a SVCSkipList apresentou desempenho mais consistente, entre 35.8 e 37.3 segundos. Para a configuração com 7 réplicas, a vantagem da SVCSkipList tornou-se ainda mais evidente, com um tempo médio de 35.7 segundos em comparação a 44.2 segundos do Baseline, que também exibiu maior variabilidade. Esses resultados indicam que nosso mecanismo otimizado de transferência de estado reduz o tempo de sincronização e melhora a consistência entre execuções, especialmente à medida que a escala do sistema aumenta.

Também avaliamos o impacto da variação do número de clusters no tempo de sincronização de estado da SVCSkipList, no mesmo cenário. Os resultados da Figura 4.15 indicam uma relação não linear entre o nível de cluster e o tempo de sincronização. Por exemplo, com nível de cluster 5 (32 749 clusters), o tempo médio de sincronização foi de aproximadamente 39.8 segundos, enquanto o nível 9 (2 053 clusters) alcançou

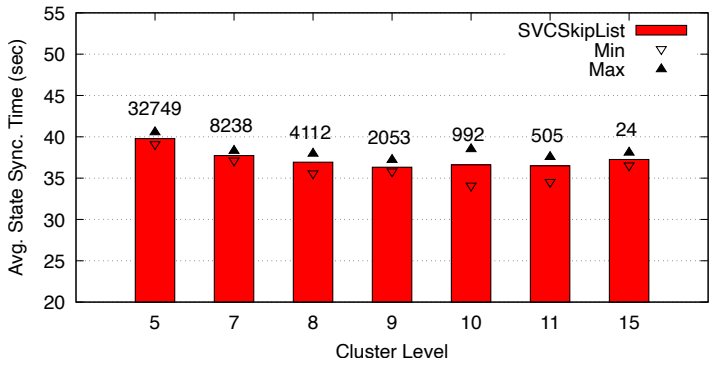


Figure 4.15. Tempo de sincronização de estado da SVCSkipList com 4 réplicas sob o workload RM, mostrado para diferentes níveis de cluster (x-axis) e tamanhos de cluster (valores acima das barras).

o menor tempo médio, em torno de 36.3 segundos. Aumentar o nível de cluster além desse ponto não resultou em melhorias adicionais: o nível 15 (apenas 24 clusters) aumentou ligeiramente para 37.3 segundos. Esses resultados sugerem que uma granularidade moderada de clusters (como o nível 9) oferece um equilíbrio ideal entre overhead de metadados e eficiência de transferência por cluster, resultando em menores tempos de sincronização. Assim, os próximos experimentos adotam o nível 9 como configuração para nossa estratégia de clustering.

As Figuras 4.16a e 4.16b ilustram o impacto da sincronização de estado na vazão do sistema para as técnicas SVCSkipList e Baseline em uma configuração com 4 réplicas sem comportamento malicioso. Cada gráfico apresenta a vazão de uma réplica operacional e da réplica em recuperação ao longo do tempo, com a área cinza indicando o intervalo de sincronização. Durante a sincronização, a vazão da réplica operacional sofre uma leve queda. Como esperado, a vazão da réplica em recuperação cai a zero durante todo o período de sincronização. Após a conclusão da sincronização, a réplica em recuperação apresenta um pico transitório de vazão devido ao acúmulo de requisições enfileiradas durante o processo de sincronização. Em seguida, tanto a réplica operacional quanto a réplica em recuperação convergem para níveis de vazão comparáveis aos observados antes da transferência de estado. A duração da sincronização é notavelmente menor para a SVCSkipList. Além disso, a Tabela 4.5 fornece informações adicionais. Embora a vazão de ambas as técnicas seja semelhante durante a sincronização de estado, a SVCSkipList apresenta latência menor que o Baseline na maioria dos percentis.

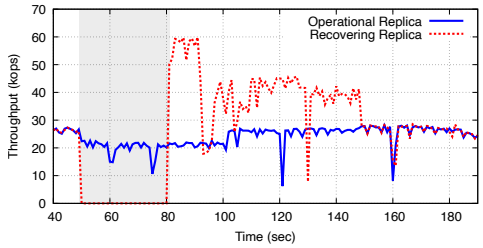
				Vazão (kops)	**Latências** (ms)				
WL	**R#**	**Algoritmo**	**Estado**	Méd.	Méd.	50^{th}	99^{th}	99.9^{th}	99.99^{th}
RM	4	SVCSkipList	1GB	20.3	44.0	43.0	81.0	315.8	856.5
	4	Baseline	1GB	20.1	46.2	44.0	85.3	380.3	1507.1
	4	SVCSkipList	2GB	21.2	50.2	46.3	152.0	981.3	1985.4
	4	Baseline	2GB	21.2	54.5	51.0	143.0	766.2	2677.4
	7	SVCSkipList	1GB	8.1	145.2	144.3	163.6	733.3	915.7
	7	Baseline	1GB	8.0	155.5	147.0	231.3	1301.3	1531.4
	7	SVCSkipList	2GB	8.0	145.5	144.6	174.3	934.1	1415.8
	7	Baseline	2GB	8.0	155.9	146.0	162.0	1018.8	1104.4

Table 4.5. Vazão e latência durante a sincronização de estado em várias configurações no ponto de maior potência (isto é, máxima razão entre vazão e latência). Melhores valores de vazão e latência para cada configuração em azul. A SVCSkipList supera o Baseline em quase todos os casos.

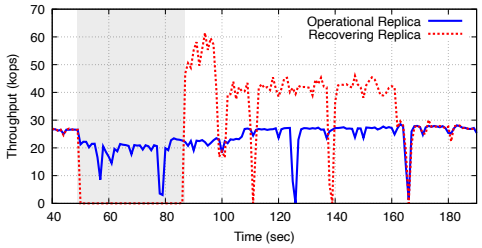
Sincronização de estado sob ataque Também investigamos o comportamento da sincronização de estado em condições adversárias. Emulamos ataques Bizantinos nos quais uma réplica defeituosa corrompe o estado antes de enviá-lo durante a sincronização. Mais precisamente, o ataque consiste em alterar o valor de uma chave selecionada aleatoriamente no armazenamento chave–valor antes da transmissão. No caso da SVCSkipList, a corrupção ocorre dentro de um único cluster, que é a unidade de transmissão de dados em nosso projeto.

A Tabela 4.4 mostra os tempos médio, mínimo e máximo de sincronização de estado para ambas as abordagens sob ataque. Com quatro réplicas (isto é, uma réplica Bizantina), a SVCSkipList alcançou um tempo médio de sincronização de aproximadamente 36.1 segundos, enquanto o Baseline exigiu cerca de 51.5 segundos, além de apresentar maior variância. Notavelmente, o tempo de sincronização da SVCSkipList sob ataque permanece equivalente ao observado sem ataques, devido à sua capacidade de validar clusters de forma independente, descartando apenas os inválidos para que sejam rapidamente requisitados novamente, enquanto mantém a transferência e validação paralelas dos demais clusters.

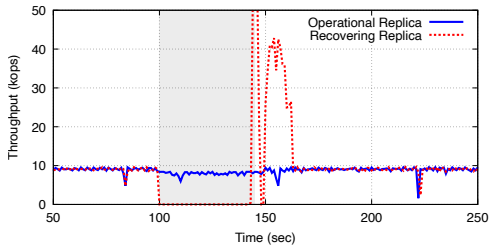
Com sete réplicas (isto é, duas réplicas Bizantinas), a SVCSkipList completou a sincronização em um tempo médio de 37.3 segundos, enquanto a abordagem Baseline levou significativamente mais tempo, com média de 103.2 segundos, novamente apresentando maior variabilidade. Esses resultados demonstram que a SVCSkipList supera consistentemente o Baseline em cenários sob ataque, alcançando até 64% de redução no tempo de sincronização na configuração com sete réplicas, além de apresentar desempenho mais estável entre execuções.



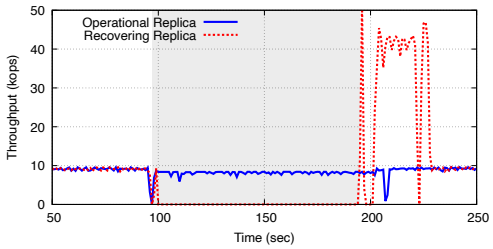
(a) SVCSkipList, 4 réplicas, sem ataque



(b) Baseline, 4 réplicas, sem ataque



(c) SVCSkipList, 7 réplicas, sob ataque



(d) Baseline, 7 réplicas, sob ataque

Figure 4.16. Vazão ao longo do tempo para SVCSkipList e Baseline, mostrando tanto uma réplica operacional quanto a réplica em recuperação, em um sistema com quatro réplicas sem ataques (topo) e sete réplicas sob ataque de duas réplicas Bizantinas (base). A área cinza indica o período de sincronização de estado.

A SVCSkipList demonstra tempos de sincronização reduzidos mesmo com réplicas Bizantinas adicionais. Isso se deve ao seu projeto, que permite transferências paralelas de clusters e auto-verificação por cluster, possibilitando que réplicas honestas sejam utilizadas simultaneamente e que clusters corrompidos sejam descartados de forma eficiente e rapidamente requisitados novamente. Em contraste, a abordagem Baseline sofre uma degradação significativa sob ataques, com o tempo de sincronização quase dobrando na presença de duas réplicas Bizantinas. Isso decorre de sua abordagem monolítica de transferência de estado, que exige a reinicialização completa da sincronização sempre que uma corrupção é detectada, limitando a escalabilidade e a resiliência sob falhas Bizantinas.

As Figuras 4.16c e 4.16d comparam a interferência na vazão do sistema para as abordagens SVCSkipList e Baseline durante a sincronização de estado sob ataque de duas réplicas Bizantinas em uma configuração com sete réplicas. Os resultados demonstram que a SVCSkipList reduz significativamente o tempo de sincronização sob ataque com impacto mínimo na vazão geral do sistema, enquanto o Baseline incorre em tempos de sincronização substancialmente mais longos, afetando negativamente o desempenho do sistema.

4.4 Trabalhos relacionados

Para contextualizar nossas contribuições, revisamos esforços relacionados em três áreas principais: sincronização de estado em sistemas replicados, técnicas de sincronização baseadas em blockchains e o uso de skiplists como estruturas de dados. Pesquisas anteriores em replicação de máquina de estados—especialmente em sistemas tolerantes a falhas Bizantinas (BFT)—exploraram diversos mecanismos para checkpointing, recuperação e transferência de estado. Em paralelo, plataformas de blockchain desenvolveram estruturas baseadas em Merkle para suportar sincronização escalável e verificável, impulsionadas pelas demandas de descentralização e por estados de grande escala. Por fim, skiplists têm sido amplamente adotadas na literatura de sistemas por sua simplicidade e eficiência, mas seu potencial permanece pouco explorado no contexto de gerenciamento de estado em BFT. Discutimos trabalhos representativos em cada domínio e posicionamos nossas contribuições em relação a eles.

4.4.1 Sincronização de estado

Melhorar a velocidade de sincronização de estado sem a necessidade de processar todo o log de transações tem sido um foco central para muitos sistemas de replicação de máquina de estados tolerantes a falhas Bizantinas (BFT SMR). Embora trabalhos anteriores tenham abordado estratégias para checkpointing e recuperação em BFT SMR [20, 32], o uso de estruturas de dados especializadas para facilitar transferência e validação permanece inexplorado. Em [12], os autores avançam as técnicas de checkpointing e sincronização do BFT-SMaRt. A plataforma SMR trata o estado como uma entidade opaca por meio de funções fornecidas pela aplicação. Checkpointing sequencial é proposto para mitigar o impacto no desempenho durante a execução normal, escalonando snapshots de estado entre réplicas. Enquanto um snapshot completo é obtido de uma única réplica, o log de comandos faltante é particionado e recuperado em paralelo a partir de múltiplas réplicas. Réplicas adicionais fornecem hashes do estado e de segmentos do log, permitindo validação.

Em [18], os autores propõem recuperação proativa no PBFT, uma abordagem seminal de organização de estado para otimizar transferência em sistemas BFT. O estado é estruturado como uma sequência de páginas de tamanho fixo, cada uma correspondendo a uma folha de uma árvore de Merkle. O conteúdo das páginas é opaco para a plataforma SMR, pois é gerenciado pela aplicação. Checkpoints são criados periodicamente em intervalos de consenso, durante os quais as réplicas produzem uma cópia da árvore de Merkle usando uma estratégia de copy-on-write. Durante a recuperação, uma réplica primeiro obtém uma raiz de Merkle certificada (de $f + 1$ respostas coincidentes) para um checkpoint dado. Em seguida, ela recupera recursivamente nós

internos (metadados) e páginas folha de diferentes réplicas. As respostas incluem os digests das páginas e o número do último checkpoint. Uma vez que todas as páginas sob um mesmo nó pai são coletadas, o pai pode ser verificado. Esse processo recursivo sobe até a raiz, confirmando consistência para o checkpoint.

Enquanto em [18] estado e metadados usam estruturas diferentes, [41] propõe uma única estrutura que combina estado da aplicação e facilita particionamento e verificação: a AVL* tree, uma árvore AVL "Merkleizada". Projetada para blockchains, a AVL* tree fornece funcionalidades de armazenamento chave-valor no nível da aplicação enquanto organiza chunks da árvore para transferência de estado eficiente, reconstrução e validação rápida, e também reduz o impacto de checkpointing na vazão. Durante a execução normal, atualizações propagam-se incrementalmente: quando uma transação afeta um chunk, apenas o hash do chunk e seu caminho até a raiz são recomputados. Contudo, para manter a AVL* balanceada, nós podem mover-se de um chunk para outro, adicionando complexidade aos algoritmos. Um peer em recuperação ou um novo peer obtém uma raiz de Merkle confiável e o número de chunks a partir do cabeçalho de um bloco subsequente. O peer então baixa chunks em paralelo de diferentes peers, validando cada chunk de forma independente ao verificar sua prova de inclusão contra a raiz de Merkle e recomputar hashes internos de suas folhas até a raiz do chunk. Um processo de reconstrução determinístico garante estruturas consistentes da árvore entre peers corretos.

Nosso trabalho tem alguma semelhança com [41], no contexto de blockchains; entretanto, nós ou otimizamos estruturas em árvore em termos de eficiência de cache, desempenho, tamanhos de prova e eficiência de espaço (B+AVL tree), ou miramos sistemas SMR genéricos e usamos diferentes estruturas de dados, como skiplists, em vez de estruturas em árvore (SVCSkipList), aproveitando as vantagens das skiplists mencionadas anteriormente e abrindo caminho para possíveis melhorias algorítmicas futuras baseadas em otimizações e variações de skiplists [95].

Preocupações sobre o impacto de checkpointing e transferência rápida de estado também aparecem na literatura de SMR tolerante a falhas por crash. Em [58], os autores propõem usar diferentes réplicas para armazenar partes do estado (partições), permitindo que uma recuperação busque diferentes partições em paralelo de diferentes réplicas. A abordagem garante que snapshots de partições permaneçam consistentes com respeito a comandos entre partições. Os logs são aplicados a cada partição para assegurar um estado correto, atualizado e completo. Em linha similar, transferência de estado sob demanda é proposta em [75]. Nessa abordagem, partes do estado são transferidas conforme necessário pela réplica em recuperação, que começa a processar o log e novos comandos sem necessariamente completar a instalação de todo o estado.

4.4.2 Blockchains

Diferentemente de sistemas SMR tradicionais, que frequentemente consideram árvores de Merkle muito caras ou as utilizam apenas periodicamente para sincronização, muitos sistemas de blockchain já alavancam estruturas de dados baseadas em Merkle para armazenamento de estado. Hoje, à medida que os tamanhos de estado em sistemas de blockchain continuam a crescer, sincronizar torna-se mais custoso. Consequentemente, árvores de Merkle têm recebido atenção considerável. Essas árvores desempenham um papel vital em permitir light clients, capacitando-os a consultar folhas específicas eficientemente e verificar sua integridade sem precisar baixar todo o estado ou o histórico de transações.

No Geth [43], a implementação em Go do Ethereum, a sincronização de estado envolve requisitar nós individuais da árvore. Peers não conseguem antecipar a duração desse processo de sincronização por não conhecerem o número total de nós [45]. Como a árvore de Merkle faz parte das regras de consenso, com raízes de Merkle armazenadas em cabeçalhos de blocos, peers podem autenticar a correção dos nós recebidos. Contudo, o desempenho de peers que requisitam e que servem é adversamente afetado pelo pequeno tamanho dos nós (menos de 1 KB), sua distribuição aleatória no banco de dados e o grande tamanho do estado (dezenas de GBs), quando nós são requisitados individualmente.

Agrupar (batching) nós de uma árvore poderia ser uma solução potencial, mas impõe desafios para garantir verificabilidade enquanto protege peers honestos contra ataques maliciosos. Por exemplo, no OpenEthereum [81], snapshots são criados periodicamente serializando todo o estado e segmentando-o em chunks grandes, cada um associado a hashes publicados em um arquivo manifesto. No entanto, como o manifesto não é integrado ao mecanismo de consenso, não é possível verificar a correção de um chunk antes de baixar todos os chunks. Assim, a conclusão bem-sucedida da sincronização de estado depende de obter um manifesto correto, exigindo suposições fortes como confiar em um peer específico ou presumir que a maioria dos peers conectados seja honesta. Em snapshots Tendermint IAVL+, nós da árvore são serializados e agrupados em chunks de tamanho fixo [1]. Entretanto, como o cabeçalho de bloco do Tendermint não inclui hashes do snapshot, peers não conseguem verificar chunks incrementalmente, ressaltando a necessidade de uma estrutura em árvore com suporte nativo a chunking.

Melhorias de escalabilidade em blockchains também podem ser classificadas em duas categorias principais: on-chain e off-chain. Soluções on-chain melhoram a própria infraestrutura da blockchain, como algoritmos de consenso eficientes [60][85] e sharding [71][98]. Soluções off-chain, ou técnicas de “Layer-2” (L2), descarregam computação e armazenamento para reduzir a carga na blockchain principal (L1). Exemp-

los incluem State Channels [79] e rollups [17]. zkSync Lite [64], um rollup de zero-knowledge, alcança desempenho quase 6 vezes mais rápido do que o Ethereum.

No contexto de otimização de mecanismos de rollup, árvores Sparse Merkle têm sido adotadas. Em [72] os autores estudam os algoritmos de árvore Sparse Merkle apresentados no zkSync Lite e propõem um algoritmo eficiente de atualização em lote para calcular uma nova raiz de hash dada uma lista de operações de conta (folha). Usando a construção do zkSync Lite como benchmark, o algoritmo melhora o tempo de atualização de conta de $O(log_n)$ para $O(1)$ e reduz o custo de atualização em lote pela metade usando uma travessia em um único passo. Avanços em criptografia também introduziram generalizações de árvores de Merkle chamadas acumuladores, permitindo provas $O(1)$ de pertencimento a conjunto e clientes de blockchain stateless [14].

4.4.3 Skiplists

Em um survey abrangente [95], argumenta-se que skiplists são simples, fáceis de implementar e possuem a mesma complexidade assintótica que contrapartes baseadas em árvores. Experimentos comparando árvores AVL e árvores auto-ajustáveis mostram experimentalmente que skiplists têm melhor desempenho em operações de busca, inserção e remoção. Vistas como uma alternativa a árvores balanceadas, skiplists tornaram-se amplamente adotadas em diversos sistemas devido à sua simplicidade, pois não exigem re-balanceamento. Desde que a skiplist original foi introduzida [84], um corpo considerável de otimizações emergiu, incluindo operações concorrentes lock-free, multi-versionamento e particionamento. Particionar a skiplist pode reduzir sua altura, diminuindo o número de acessos e atualizações de ponteiros, como demonstrado em [70]. Essa abordagem é geralmente considerada mais simples de implementar do que em estruturas baseadas em árvores. Tanto [48] quanto [31] abordam o problema de verificar informação recuperada de uma entidade não confiável. O primeiro emprega skiplists com hashing, enquanto o segundo usa skiplists para armazenar hashes de entradas de tabelas mantidas em um Sistema de Gerenciamento de Banco de Dados (DBMS). Os dados armazenados e os hashes são fornecidos por uma fonte confiável. Usuários que recuperam dados do armazenamento não confiável podem verificá-los com os hashes armazenados e informações públicas da fonte. Além disso, diversas implementações de armazenamento chave-valor baseadas em skiplists estão disponíveis.

Embora skiplists tenham sido extensivamente estudadas, a combinação de versionamento, particionamento e auto-validação em uma única skiplist não foi encontrada na literatura. Além disso, a aplicação de skiplists para otimizar gerenciamento de estado em sistemas BFT SMR permanece inexplorada.

4.5 Conclusão

Neste capítulo, apresentamos duas estruturas de dados complementares que abordam aspectos-chave de replicação de máquina de estados sob tolerância a falhas Bizantinas. Nosso objetivo foi melhorar a eficiência e robustez do gerenciamento de estado, tanto em termos do que é armazenado (layout da estrutura de dados) quanto de como é sincronizado (protocolos de transferência de estado). A seguir, resumimos os benefícios e resultados da avaliação de cada estrutura.

4.5.1 B+AVL tree

Abordamos o desafio de transferência eficiente de estado em sistemas de blockchain, particularmente para a recuperação de peers defasados. Embora soluções existentes dependam de snapshots e validação baseada em Merkle para evitar replay completo de transações, elas enfrentam limitações em eficiência de clusterização e tamanho de provas. Propusemos B+AVL trees, uma nova estrutura de dados que combina as forças de balanceamento e validação de árvores AVL com a eficiência de espaço de B+Trees. Diferentemente de AVL* trees, B+AVL trees simplificam a manutenção de clusters ao impedir o movimento de folhas entre clusters e fornecem provas de validação mais compactas do que B+Trees de Merkle. Resultados experimentais demonstram as vantagens práticas dessa abordagem.

A Tabela 4.6 apresenta uma análise quantitativa de nossos resultados. Ela destaca as principais vantagens de B+AVL trees e fornece um resumo conciso dos resultados em várias métricas, comparando o desempenho de todas as técnicas normalizado pela árvore Baseline, com valores menores representando melhor desempenho.

Em resumo:

- A B+AVL tree demonstra consistência de desempenho em operações de inserção e busca, comparável a outras estruturas em árvore. Notavelmente, ela supera todas as demais técnicas em operações de busca com árvores grandes, pois armazena chaves de forma contígua dentro de um cluster, otimizando o uso de cache.

- A B+AVL tree apresenta tamanhos de prova de Merkle significativamente menores em comparação com B+Trees tradicionais e exibe tamanhos de prova mais estáveis do que a AVL* tree à medida que o tamanho do cluster aumenta, como refletido pelos valores de desvio padrão dos tamanhos de prova na Figura 4.5.

- A árvore Baseline alcança a maior eficiência de espaço ao preencher completamente todos os clusters, exceto o último. A B+AVL, contudo, apresenta eficiência de espaço superior às demais técnicas, especialmente evidente com clusters

		Baseline tree	AVL* tree	B+Tree	B+AVL tree
Search	Small tree	**1**	2.86	1.95	1.68
	Large tree	1	1.04	0.76	**0.32**
Insert	Small tree	**1**	1.10	1.64	2.05
	Large tree	1	0.86	**0.83**	0.89
Proof size	Small cluster	1	1	2.85	**0.99**
	Large cluster	1	0.99	192.20	**0.98**
Space efficiency	Small cluster	**1**	1.58	1.42	1.42
	Large cluster	**1**	1.48	1.40	1.41
No Byz. State Sync.	Small cluster	1	—	—	**0.97**
	Large cluster	1	—	—	**0.32**
	Few peers	1	—	—	**0.63**
	Many peers	1	—	—	**0.62**
One Byz. State Sync.	Small cluster	1	—	—	**0.40**
	Large cluster	1	—	—	**0.15**
	Few peers	1	—	—	**0.21**
	Many peers	1	—	—	**0.33**
Many Byz. State Sync.	1 byz. peer	1	—	—	**0.20**
	4 byz. peers	1	—	—	**0.04**

Table 4.6. Resumo dos resultados da nossa B+AVL tree para várias métricas avaliadas, comparando o desempenho de todas as técnicas normalizado pela árvore Baseline. Os valores representam o desempenho médio observado em múltiplos experimentos. Valores menores indicam melhor desempenho. Melhores resultados estão em negrito.

menores. Embora todas as técnicas exibam eficiência média de espaço similar, a B+AVL e a B+Tree demonstram variância mais instável à medida que a árvore cresce (ver valores de desvio padrão para clusters grandes na Tabela 4.2).

- Em sincronização de estado, a B+AVL supera a árvore Baseline tanto em cenários não Bizantinos quanto Bizantinos. Sua sincronização mais rápida decorre de serialização eficiente e reconstruções facilitadas por um layout de memória compacto. Sob ataque, clusters auto-verificáveis permitem detecção e recuperação rápidas, enquanto a Baseline sofre com tentativas custosas. A diferença de desempenho cresce com mais nós Bizantinos e, em alguns casos, B+AVL tem desempenho ainda melhor sob ataque devido a caminhos de recuperação otimizados.

4.5.2 SVCSkipList

Também introduzimos o conceito de estruturas de dados clusterizadas auto-validáveis como uma abordagem fundamentada para melhorar o gerenciamento de estado em

replicação de máquina de estados tolerante a falhas Bizantinas (BFT SMR). Instanciamos essa ideia com a SVCSkipList, uma nova estrutura de dados que possibilita transferência de estado eficiente, paralela e verificável de forma independente. Ao integrar estreitamente representação e validação de estado ao framework de replicação, nossa solução aborda limitações críticas de bibliotecas SMR existentes, especialmente na presença de falhas Bizantinas.

Fase de execução	**Principal benefício da SVCSkipList**
Execução normal	Vazão comparável ou maior; latência de cauda significativamente menor em todas as cargas
Checkpointing	Até 2× mais rápido para estados grandes devido a multi-versionamento
Sincronização (sem ataque)	Tempos de sincronização mais rápidos e consistentes, especialmente com muitos réplicas
Sincronização (sob ataque)	Até 64% menos tempo graças à verificação paralela baseada em clusters

Table 4.7. Resumo das melhorias de desempenho alcançadas pela SVCSkipList em comparação ao baseline.

Integramos nossa estrutura de dados SVCSkipList ao conhecido framework BFT-SMaRt e a avaliamos em um ambiente de rede de longa distância (WAN). Nossa avaliação demonstra que SVCSkipList oferece desempenho robusto em uma variedade de cargas de trabalho e tamanhos de sistema, trazendo melhorias tanto na execução normal quanto em cenários de sincronização de estado. A Tabela 4.7 apresenta uma comparação concisa dos ganhos de desempenho ao longo das fases de execução.

Em resumo:

- **Execução normal:** Nossa abordagem, empregando a SVCSkipList, alcança vazão comparável ao baseline em todas as configurações e consistentemente maior em estados maiores. Ela mantém latências média e de cauda estáveis, mesmo quando o estado cresce de 1GB para 4GB. Notavelmente, ela também supera o baseline em latências de alto percentil (por exemplo, 99.9% e 99.99%)—reduzindo latências de cauda em até 60%—e exibe maior resiliência a degradação de desempenho em implantações maiores (7 réplicas).

- **Checkpointing:** Nossa abordagem reduz significativamente os tempos de checkpointing graças a multi-versionamento eficiente e à estrutura de skiplist, e reduz pela metade o tempo de checkpointing para estados grandes (até 4GB) em comparação ao baseline.

- **Sincronização de estado sem ataque:** Nossa abordagem apresenta tempos de sincronização levemente mais rápidos e mais consistentes (por exemplo, 36.3s vs. 39.3s para 4 réplicas) e escala melhor em grupos maiores de réplicas (por exemplo, 35.7s vs. 44.2s para 7 réplicas).
- **Sincronização de estado sob ataque Bizantino:** Ela também alcança até 64% menos tempo de sincronização (por exemplo, 37.3s vs. 103.2s com 7 réplicas) e evita reinícios completos de transferência de estado ao validar clusters independentemente e em paralelo, enquanto o baseline sofre com tentativas em cascata sempre que alguma corrupção é detectada.

4.5.3 Considerações finais

Em conjunto, a B+AVL tree e a SVCSkipList oferecem uma abordagem fundamentada e prática para aprimorar SMR resiliente a Bizantinos. A B+AVL tree aborda limitações de armazenamento e tamanho de provas. Tanto a B+AVL tree quanto a SVCSkipList melhoram recuperação e sincronização ao habilitar paralelismo e verificação baseada em clusters. Ao integrar esses desenhos a frameworks de replicação, sistemas podem reduzir significativamente overheads de desempenho e aumentar resiliência em ambientes adversariais e de grande escala.

Capítulo 5

Conclusão

Esta tese abordou desafios fundamentais no projeto e na implementação de sistemas de Replicação por Máquina de Estados (State Machine Replication – SMR), com foco em comunicação eficiente e gerenciamento de estado. Embora a SMR forneça uma abstração poderosa para construir sistemas distribuídos tolerantes a falhas, seu desempenho e sua escalabilidade permanecem limitados pelos custos de seus principais componentes estruturais, como a coordenação entre réplicas e a sincronização de estado. O trabalho apresentado aqui propôs novas técnicas para mitigar algumas dessas limitações, demonstrando que é possível alcançar maior vazão e menor latência sem comprometer a correção ou a confiabilidade.

5.1 Resumo das contribuições

As principais contribuições desta tese abrangem duas dimensões complementares: comunicação e gerenciamento de estado.

- **Quiescência:** A introdução de uma nova propriedade de atomic multicast para refinar a minimalidade, garantindo correção e desempenho durante a comunicação. Enquanto a minimalidade especifica quando processos podem trocar mensagens, a quiescência indica quando eles devem cessar a comunicação.

- **FlexCast:** Um protocolo de atomic multicast genuíno baseado em overlay que reduz a sobrecarga de comunicação enquanto preserva garantias de ordenação. Ao explorar um overlay em grafo acíclico direcionado (DAG) e decisões locais de ordenação, o FlexCast alcança menor latência e melhor escalabilidade em ambientes geograficamente distribuídos do que protocolos de ponta.

- **FlexCast reconfigurável:** Um mecanismo de reconfiguração dinâmica que permite ao overlay de multicast se adaptar a mudanças na carga de trabalho e na rede.

- **B+AVL trees:** Uma nova estrutura de dados que combina as propriedades de balanceamento e verificação das árvores AVL com a compacidade das B+Trees. As B+AVL trees permitem transferência de estado eficiente e verificável ao fornecer provas menores, desempenho estável em diferentes cargas de trabalho e layout de memória otimizado.

- **SVCSkipList:** A primeira realização de uma *estrutura de dados clusterizada autovalidável* para SMR tolerante a falhas Bizantinas de propósito geral. O SVCSkipList suporta transferência de estado paralela e independentemente verificável, alcançando ganhos substanciais de desempenho em vazão, checkpointing e sincronização, mesmo sob condições Bizantinas.

Em conjunto, essas contribuições avançam a compreensão e a realização prática de sistemas replicados escaláveis e confiáveis. Elas demonstram como a combinação de otimizações de comunicação e de estruturas de dados pode oferecer melhorias mensuráveis em latência, vazão e eficiência de recuperação mantendo fortes garantias de consistência.

5.2 Direções futuras

Embora esta tese represente um avanço significativo, ela também abre diversas possibilidades para exploração futura. Essas possibilidades podem ser agrupadas em duas categorias principais: (i) extensões incrementais do trabalho atual e (ii) direções de pesquisa mais amplas inspiradas por seus resultados.

5.2.1 Extensões incrementais

- **Otimização de overlay baseada em aprendizado:** Embora a reconfiguração do FlexCast adapte overlays à localidade da carga de trabalho, trabalhos futuros podem explorar o uso de IA ou aprendizado de máquina para detectar padrões em tráfego, latência ou confiabilidade de nós, e otimizar proativamente o overlay. Por exemplo, um modelo leve poderia prever hotspots, antecipar falhas ou identificar grupos de réplicas que se beneficiariam de ajustes locais de ordenação, reduzindo latência e sobrecarga de comunicação em implantações dinâmicas ou geograficamente distribuídas. Abordagens híbridas que combinem multicast

genuíno e hierárquico também poderiam ser guiadas por tais modelos para melhorar ainda mais a eficiência.

- **Integração adaptativa de estruturas de dados:** As estruturas de dados propostas, B+AVL trees e SVCSkipList, poderiam ser integradas a plataformas blockchain ou bibliotecas SMR existentes de modo a permitir que o sistema selecione adaptativamente qual estrutura utilizar com base em características da carga de trabalho, tamanho do estado ou requisitos de desempenho. Por exemplo, as B+AVL trees poderiam ser usadas para atualizações de estado compactas e eficientes em cache, enquanto o SVCSkipList poderia lidar com transferências de estado em larga escala ou resilientes a falhas Bizantinas. Essa seleção adaptativa, combinada com otimizações para gerenciamento de memória, serialização e caching, permitiria uma implantação mais eficiente e flexível em ambientes reais.

- **Estender outras estruturas de dados com clusterização e autovalidação:** Além das B+AVL trees e do SVCSkipList, trabalhos futuros podem investigar como outros tipos de estruturas de dados, como grafos, tries ou índices baseados em hash, podem ser aprimorados com clusterização, multiversionamento e mecanismos de autovalidação. O estudo dessas estruturas permitiria avaliar compromissos de desempenho, aplicabilidade a diferentes cargas de trabalho e resiliência sob condições Bizantinas ou de alta latência. Essa exploração pode ampliar o conjunto de técnicas verificáveis de gerenciamento de estado para sistemas SMR e blockchain, além de fornecer orientações para otimizações específicas por carga de trabalho.

5.2.2 Direções de pesquisa em alto nível

Além das extensões imediatas discutidas acima, este trabalho também abre direções de pesquisa mais amplas que podem ser perseguidas com base nos mecanismos e nas percepções desenvolvidas nesta tese.

- **Integração com motores de consenso modernos:** As estruturas de dados propostas para sincronização de estado também podem ser incorporadas a outros frameworks amplamente utilizados de SMR ou blockchain, como Tendermint [15] ou HotStuff [97], para avaliar seu impacto em cenários realistas de implantação. Isso proporcionaria uma compreensão mais aprofundada de como multicast otimizado e gerenciamento de estado verificável interagem com algoritmos de consenso existentes.

- **Desacoplar ordenação e disseminação do payload:** Trabalhos futuros podem explorar técnicas de atomic multicast que desacoplem a disseminação da ordenação de requisições da entrega do payload propriamente dita. Por exemplo, um overlay baseado em árvore poderia ser usado para estabelecer uma ordem global consistente de forma eficiente, enquanto um overlay em DAG completo (C-DAG) poderia disseminar simultaneamente o payload às réplicas usando conectividade direta. Essa separação permitiria que cada overlay explorasse seus pontos fortes: ordenação rápida com mínima comunicação na árvore e distribuição eficiente e paralela do payload no C-DAG, potencialmente melhorando latência, vazão e escalabilidade em implantações SMR grandes ou geograficamente distribuídas.

De forma geral, essas direções se apoiam diretamente nas bases estabelecidas por esta tese, com foco em extensões práticas capazes de reduzir a lacuna entre pesquisa experimental e implantações confiáveis em larga escala.

5.3 Considerações finais

Em conclusão, esta tese contribui com avanços tanto teóricos quanto práticos em direção à replicação escalável e tolerante a falhas. Ao repensar como réplicas se comunicam e gerenciam estado, mostra que os compromissos tradicionais entre desempenho e confiabilidade podem ser mitigados por meio de mecanismos cuidadosamente projetados. As técnicas propostas aqui, de multicast baseado em overlay a estruturas de dados autovalidáveis, lançam as bases para a próxima geração de sistemas distribuídos confiáveis capazes de atender às exigências de desempenho e confiabilidade de aplicações modernas.

Bibliography

[1] *ADR 053: State Sync Prototype* [n.d.]. https://github.com/tendermint/-tendermint/blob/master/docs/architecture/adr-053-state-sync-prototype.md.

[2] Ahmed-Nacer, T., Sutra, P. and Conan, D. [2016]. The convoy effect in atomic multicast, *2016 IEEE 35th Symposium on Reliable Distributed Systems Workshops (SRDSW)*, IEEE Computer Society, Los Alamitos, CA, USA, pp. 67–72.
URL: *https://doi.ieeecomputersociety.org/10.1109/SRDSW.2016.22*

[3] Alchieri, E., Bessani, A., Greve, F. and Fraga, J. d. S. [2017]. Efficient and Modular Consensus-Free Reconfiguration for Fault-Tolerant Storage, *International Conference on Principles of Distributed Systems*.

[4] Alchieri, E., Dotti, F., Marandi, P., Mendizabal, O. and Pedone, F. [2018]. Boosting state machine replication with concurrent execution, *2018 Eighth Latin-American Symposium on Dependable Computing (LADC)*, pp. 77–86.

[5] Alchieri, E., Dotti, F. and Pedone, F. [2018]. Early scheduling in parallel state machine replication, *ACM SoCC*.

[6] Amazon Web Services [2025]. Amazon EC2 Instances, https://aws.amazon.com/ec2/. Accessed: 2025-06-25.

[7] Batista, E., Alchieri, E., Dotti, F. and Pedone, F. [2019]. Resource utilization analysis of early scheduling in parallel state machine replication, *9th Latin-American Symposium on Dependable Computing (LADC)*.

[8] Batista, E., Alchieri, E., Dotti, F. and Pedone, F. [2022]. Early scheduling on steroids: Boosting parallel state machine replication, *Journal of Parallel and Distributed Computing* .
URL: *https://www.sciencedirect.com/science/article/pii/S0743731522000375*

[9] Batista, E., Coelho, P., Alchieri, E., Dotti, F. and Pedone, F. [2023]. Flexcast: Genuine overlay-based atomic multicast, *Proceedings of the 24th International Middleware Conference*, Middleware '23, Association for Computing Machinery, New York, NY, USA, p. 288–300.

[10] Bessani, A., Alchieri, E., Sousa, J., Oliveira, A. and Pedone, F. [2020]. From byzantine replication to blockchain: Consensus is only the beginning, *International Conference on Dependable Systems and Networks*.

[11] Bessani, A., Santos, M., Felix, J. a., Neves, N. and Correia, M. [2013]. On the efficiency of durable state machine replication, *Proceedings of the 2013 USENIX Conference on Annual Technical Conference*, USENIX ATC'13, USENIX Association, USA, p. 169–180.

[12] Bessani, A., Sousa, J. and Alchieri, E. E. [2014]. State machine replication for the masses with bft-smart, *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pp. 355–362.

[13] Birman, K. P. and Joseph, T. A. [1987]. Reliable communication in the presence of failures, *ACM Trans. Comput. Syst.* **5**(1): 47–76.
URL: *https://doi.org/10.1145/7351.7478*

[14] Boneh, D., Bünz, B. and Fisch, B. [2019]. Batching techniques for accumulators with applications to iops and stateless blockchains, *Annual International Cryptology Conference*, Springer, pp. 561–586.

[15] Buchman, E., Kwon, J. and Milosevic, Z. [2018]. The latest gossip on BFT consensus, *CoRR* **abs/1807.04938**.
URL: *http://arxiv.org/abs/1807.04938*

[16] Burrows, M. [2006]. The chubby lock service for loosely-coupled distributed systems, *OSDI*.

[17] Buterin, V. [n.d.]. An incomplete guide to rollups, https://vitalik.eth.limo/general/2021/01/05/rollup.html.

[18] Castro, M. and Liskov, B. [2002]. Practical byzantine fault tolerance and proactive recovery, *ACM Trans. Comput. Syst.* **20**(4): 398–461.
URL: *https://doi.org/10.1145/571637.571640*

[19] Chilimbi, T. M., Hill, M. D. and Larus, J. R. [1999]. Cache-conscious data structures: Design and implementation, *Proceedings of the ACM SIGPLAN Conference*

on Programming Language Design and Implementation (PLDI).
URL: *https://www.microsoft.com/en-us/research/wp-content/uploads/2016/12/ccds.pdf*

[20] Clement, A., Kapritsos, M., Lee, S., Wang, Y., Alvisi, L., Dahlin, M. and Riche, T. [2009]. Upright cluster services, *SOSP*.

[21] Cloudping [2022]. AWS Latency Monitoring Website.
URL: *https://www.cloudping.co/grid*

[22] Coelho, P., Junior, T. C., Bessani, A., Dotti, F. and Pedone, F. [2018]. Byzantine fault-tolerant atomic multicast, *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pp. 39–50.

[23] Coelho, P., Schiper, N. and Pedone, F. [2017]. Fast atomic multicast, *DSN*.

[24] Comer, D. [1979]. Ubiquitous b-tree, *ACM Comput. Surv.* **11**(2): 121–137.
URL: *https://doi.org/10.1145/356770.356776*

[25] Cooper, B. F., Silberstein, A., Tam, E., Ramakrishnan, R. and Sears, R. [2010]. Benchmarking cloud serving systems with YCSB, *Proceedings of the 1st ACM Symposium on Cloud Computing (SoCC)*, ACM, pp. 143–154.

[26] Corbett, J. C., Dean, J., Epstein, M., Fikes, A., Frost, C., Furman, J., Ghemawat, S., Gubarev, A., Heiser, C., Hochschild, P. et al. [2012]. Spanner: Google's globally-distributed database, *OSDI*.

[27] *Cosmos network* [n.d.]. https://cosmos.network/.

[28] *Cosmos SDK* [n.d.]. https://github.com/cosmos/cosmos-sdk.

[29] Council, T. P. P. [1996]. Tpc benchmark c standard specification, *http://www.tpc. org/tpcc/spec/tpcc_current. pdf* .

[30] Delporte-Gallet, C. and Fauconnier, H. [2000]. Fault-tolerant genuine atomic multicast to multiple groups, *Proceedings of the 12th International Conference on Principles of Distributed Systems (OPODIS)*, pp. 107–122.

[31] Di Battista, G. and Palazzi, B. [2007]. Authenticated relational tables and authenticated skip lists, *in* S. Barker and G.-J. Ahn (eds), *Data and Applications Security XXI*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 31–46.

[32] Distler, T. [2021]. Byzantine fault-tolerant state-machine replication from a systems perspective, *ACM Comput. Surv.* **54**(1).
URL: *https://doi.org/10.1145/3436728*

[33] Drees, M., Gmyr, R. and Scheideler, C. [2016]. Churn- and dos-resistant overlay networks based on network reconfiguration, *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures*.

[34] Duplyakin, D., Ricci, R., Maricq, A., Wong, G., Duerig, J., Eide, E., Stoller, L., Hibler, M., Johnson, D., Webb, K., Akella, A., Wang, K., Ricart, G., Landweber, L., Elliott, C., Zink, M., Cecchet, E., Kar, S. and Mishra, P. [2019]. The design and operation of CloudLab, *Proceedings of the USENIX Annual Technical Conference (ATC)*, pp. 1–14.
URL: *https://www.flux.utah.edu/paper/duplyakin-atc19*

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

[36] Facebook [2013]. Rocksdb: A persistent key-value store for flash and ram storage. https://github.com/facebook/rocksdb.

[37] Fischer, M. J., Lynch, N. A. and Paterson, M. S. [1985]. Impossibility of distributed consensus with one faulty processor, *Journal of the ACM* **32**(2): 374–382.

[38] Friedman, R. and van Renesse, R. [1997]. Packing messages as a tool for boosting the performance of total ordering protocols, *Proceedings of the 6th International Symposium on High Performance Distributed Computing, HPDC '97, Portland, OR, USA, August 5-8, 1997*, IEEE Computer Society, pp. 233–242.

[39] Fritzke, U., J., Ingels, P., Mostefaoui, A. and Raynal, M. [1998]. Fault-tolerant total order multicast to asynchronous groups, *Proceedings of the The 17th IEEE Symposium on Reliable Distributed Systems*, pp. 228–234.

[40] Fynn, E. [2021]. *Scaling Blockchains*, PhD thesis, Università della Svizzera Italiana (USI).

[41] Fynn, E., Buchman, E., Milosevic, Z., Soulé, R. and Pedone, F. [2022]. Robust and fast blockchain state synchronization, *in* E. Hillel, R. Palmieri and E. Rivière (eds), *26th International Conference on Principles of Distributed Systems, OPODIS 2022, December 13-15, 2022, Brussels, Belgium*, Vol. 253 of *LIPIcs*, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, pp. 8:1–8:22.

[42] Garcia-Molina, H. and Spauster, A. [1989]. Message ordering in a multicast environment, *[1989] Proceedings. The 9th International Conference on Distributed Computing Systems*, pp. 354–361.

[43] *Geth v1.9.0: Six months distilled* [n.d.]. https://blog.ethereum.org/2019/07/10/geth-v1-9-0/.

[44] Glendenning, L., Beschastnikh, I., Krishnamurthy, A. and Anderson, T. [2011]. Scalable consistency in scatter, *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*.

[45] *Go Ethereum FAQ* [n.d.]. https://geth.ethereum.org/docs/faq.

[46] Goodrich, M. T. and Tamassia, R. [2001]. Efficient authenticated dictionaries with skip lists and commutative hashing, *Tech. Rep.* .
URL: *https://cs.brown.edu/cgc/stms/papers/hashskip.pdf*

[47] Goodrich, M. T., Tamassia, R. and Goldwasser, M. H. [2014]. *Data Structures and Algorithms in Java*, 6 edn, John Wiley & Sons, Hoboken, NJ.

[48] Goodrich, M., Tamassia, R. and Schwerin, A. [2001]. Implementation of an authenticated dictionary with skip lists and commutative hashing, *Proceedings DARPA Information Survivability Conference and Exposition II. DISCEX'01*, Vol. 2, pp. 68–82 vol.2.

[49] Google [2011]. Leveldb. https://github.com/google/leveldb.

[50] Gotsman, A., Lefort, A. and Chockler, G. [2019]. White-box atomic multicast, *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, IEEE, pp. 176–187.

[51] Guerraoui, R. and Schiper, A. [2001]. Genuine atomic multicast in asynchronous distributed systems, *Theor. Comput. Sci.* **254**(1-2): 297–316.

[52] Hadzilacos, V. and Toueg, S. [1994a]. A modular approach to fault-tolerant broadcasts and related problems, *Technical report*, USA.

[53] Hadzilacos, V. and Toueg, S. [1994b]. A modular approach to the specification and implementation of fault-tolerant broadcasts, *Technical report*, Department of Computer Science, Cornell.

[54] Herlihy, M. and Wing, J. M. [1990]. Linearizability: A correctness condition for concurrent objects, *ACM Transactions on Programing Languages and Systems* **12**(3): 463–492.

[55] Hoang Le, L., Fynn, E., Eslahi-Kelorazi, M., Soulé, R. and Pedone, F. [2019]. Dynastar: Optimized dynamic partitioning for scalable state machine replication, *Int. Conference on Distributed Computing Systems*.

[56] Hunt, P., Konar, M., Junqueira, F. P. and Reed, B. [2010]. Zookeeper: wait-free coordination for internet-scale systems, *ATC*, Vol. 8.

[57] *IAVL+ implementation* [n.d.]. https://github.com/tendermint/iavl.

[58] Junior, E. G., Alchieri, E., Dotti, F. L. and Mendizabal, O. M. [2024]. Reducing persistence overhead in parallel state machine replication through time-phased partitioned checkpoint, *J. Internet Serv. Appl.* **15**(1): 194–211.
URL: *https://doi.org/10.5753/jisa.2024.3891*

[59] Kapritsos, M., Wang, Y., Quema, V., Clement, A., Alvisi, L. and Dahlin, M. [2012]. All about eve: execute-verify replication for multi-core servers, *OSDI*.

[60] Kiayias, A., Russell, A., David, B. and Oliynykov, R. [2017]. Ouroboros: A provably secure proof-of-stake blockchain protocol, *in* J. Katz and H. Shacham (eds), *Advances in Cryptology – CRYPTO 2017*, Springer International Publishing, Cham, pp. 357–388.

[61] Knuth, D. [1973]. *The Art Of Computer Programming, vol. 3: Sorting And Searching*, Addison-Wesley.

[62] Kotla, R. and Dahlin, M. [2003]. High throughput byzantine fault tolerance, *Technical Report UTCS-TR-03-58*, University of Texas.

[63] Kuhn, F. and Wattenhofer, R. [2004]. Dynamic analysis of the arrow distributed protocol, *Proceedings of the Sixteenth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '04, Association for Computing Machinery, New York, NY, USA, p. 294–301.
URL: *https://doi.org/10.1145/1007912.1007962*

[64] Labs, M. [n.d.]. Zksync: scaling and privacy engine for ethereum, https://github.com/matter-labs/zksync.

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

[66] Lamport, L. [1989]. The part-time parliament., *Technical Report 49*, Digital Equipment Corporation, Systems Research Centre.

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

[68] Lamport, L. [2005]. Generalized Consensus and Paxos, *Technical report*, Microsoft Research Technical Report MSR-TR-2005-33.

[69] Le, L. H., Eslahi-Kelorazi, M., Coelho, P. R. and Pedone, F. [2021]. Ramcast: Rdma-based atomic multicast, *Proceedings of the 22nd International Middleware Conference* .

[70] Li, Z., Jiao, B., He, S. and Yu, W. [2022]. Phast: Hierarchical concurrent log-free skip list for persistent memory, *IEEE Transactions on Parallel and Distributed Systems* **33**(12): 3929–3941.

[71] Luu, L., Narayanan, V., Zheng, C., Baweja, K., Gilbert, S. and Saxena, P. [2016]. A secure sharding protocol for open blockchains, *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, Association for Computing Machinery, New York, NY, USA, p. 17–30.
URL: *https://doi.org/10.1145/2976749.2978389*

[72] Ma, B., Pathak, V. N., Liu, L. and Ruj, S. [2023]. One-phase batch update on sparse merkle trees for rollups.
URL: *https://arxiv.org/abs/2310.13328*

[73] Mao, Y., Junqueira, F. P. and Marzullo, K. [2008]. Mencius: building efficient replicated state machines for wans, *Proceedings of the 8th USENIX conference on Operating systems design and implementation (OSDI)*, pp. 369–384.

[74] Marandi, P. J., Bezerra, C. E. B. and Pedone, F. [2014]. Rethinking state-machine replication for parallelism, *ICDCS*.

[75] Mendizabal, O. M., Dotti, F. L. and Pedone, F. [2017]. High performance recovery for parallel state machine replication, *in* K. Lee and L. Liu (eds), *37th IEEE International Conference on Distributed Computing Systems, ICDCS 2017, Atlanta, GA, USA, June 5-8, 2017*, IEEE Computer Society, pp. 34–44.
URL: *https://doi.org/10.1109/ICDCS.2017.193*

[76] Merkle, R. C. [1988]. A digital signature based on a conventional encryption function, *in* C. Pomerance (ed.), *Advances in Cryptology — CRYPTO '87*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 369–378.

[77] Moraru, I., Andersen, D. G. and Kaminsky, M. [2013]. There is more consensus in egalitarian parliaments, *SOSP*.

[78] Munro, J. I., Papadakis, T. and Sedgewick, R. [1992]. Deterministic skip lists, *Proceedings of the Third Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '92, Society for Industrial and Applied Mathematics, USA, p. 367–375.

[79] Negka, L. D. and Spathoulas, G. P. [2021]. Blockchain state channels: A state of the art, *IEEE Access* **9**: 160277–160298.

[80] Obelheiro, R. R. and Fraga, J. d. S. [2007]. Overlay network topology reconfiguration in byzantine settings, *13th Pacific Rim International Symposium on Dependable Computing*, pp. 155–162.

[81] *OpenEthereum WarpSync* [n.d.]. https://openethereum.github.io/wiki/Warp-Sync.

[82] Parzyjegla, H., Muhl, G. and Jaeger, M. [2006]. Reconfiguring publish/subscribe overlay topologies, *26th IEEE International Conference on Distributed Computing Systems Workshops (ICDCSW'06)*, pp. 29–29.

[83] Pedone, F. and Schiper, A. [1999]. Generic broadcast, *Distributed Computing*, Springer, pp. 94–106.

[84] Pugh, W. [1990]. Skip lists: a probabilistic alternative to balanced trees, *Commun. ACM* **33**(6): 668–676.
URL: *https://doi.org/10.1145/78973.78977*

[85] Rocket, T., Yin, M., Sekniqi, K., van Renesse, R. and Sirer, E. G. [2020]. Scalable and probabilistic leaderless bft consensus through metastability.
URL: *https://arxiv.org/abs/1906.08936*

[86] Rodrigues, L., Guerraoui, R. and Schiper, A. [1998]. Scalable atomic multicast, *International Conference on Computer Communications and Networks*, pp. 840–847.

[87] Sanfilippo, S. et al. [2009]. Redis. https://redis.io.

[88] Schiper, N. and Pedone, F. [2008a]. On the inherent cost of atomic broadcast and multicast in wide area networks, *International conference on Distributed computing and networking (ICDCN)*, pp. 147–157.

[89] Schiper, N. and Pedone, F. [2008b]. Solving atomic multicast when groups crash, *International Conference On Principles Of Distributed Systems (OPODIS)*, Springer, pp. 481–495.

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

[91] Shvachko, K., Kuang, H., Radia, S. and Chansler, R. [2010]. The hadoop distributed file system, *MSST*.

[92] Thomson, A., Diamond, T., Weng, S.-C., Ren, K., Shao, P. and Abadi, D. J. [2012]. Calvin: fast distributed transactions for partitioned database systems, *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pp. 1–12.

[93] Verma, A., Pedrosa, L., Korupolu, M., Oppenheimer, D., Tune, E. and Wilkes, J. [2015]. Large-scale cluster management at google with borg, *EuroSys*.

[94] Wood, G. et al. [2014]. Ethereum: A secure decentralised generalised transaction ledger, *Ethereum project yellow paper* **151**(2014): 1–32.

[95] Xing, L., Vadrevu, V. S. P. K. and Aref, W. G. [2025]. The ubiquitous skiplist: A survey of what cannot be skipped about the skiplist and its applications in data systems, *ACM Comput. Surv.* **57**(11).
URL: *https://doi.org/10.1145/3736754*

[96] Xing, L., Vadrevu, V. S. P. K. and Ghanem, W. [2023]. The ubiquitous skiplist: A survey of what cannot be skipped about the skiplist and its applications in data systems, *ACM Computing Surveys* .
URL: *https://dl.acm.org/doi/10.1145/3736754*

[97] Yin, M., Malkhi, D., Reiter, M. K., Gueta, G. G. and Abraham, I. [2019]. Hotstuff: Bft consensus with linearity and responsiveness, *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, PODC '19, Association for Computing Machinery, New York, NY, USA, p. 347–356.
URL: *https://doi.org/10.1145/3293611.3331591*

[98] Zamani, M., Movahedi, M. and Raykova, M. [2018]. Rapidchain: Scaling blockchain via full sharding, *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, Association for Computing Machinery, New York, NY, USA, p. 931–948.
URL: *https://doi.org/10.1145/3243734.3243853*