UNIVERSIDADE FEDERAL DE UBERLÂNDIA

Davi Augusto Silva

Otimização da estrutura hierárquica em um algoritmo de multicast atômico sujeito a falhas bizantinas

Uberlândia, Brasil

2024

UNIVERSIDADE FEDERAL DE UBERLÂNDIA

Davi Augusto Silva

Otimização da estrutura hierárquica em um algoritmo de multicast atômico sujeito a falhas bizantinas

Trabalho de conclusão de curso apresentado à Faculdade de Computação da Universidade Federal de Uberlândia, como parte dos requisitos exigidos para a obtenção título de Bacharel em Ciência da Computação.

Orientador: Paulo Rodolfo da Silva Leite Coelho

Universidade Federal de Uberlândia – UFU

Faculdade de Computação

Bacharelado em Ciência da Computação

Uberlândia, Brasil

2024

Davi Augusto Silva

Otimização da estrutura hierárquica em um algoritmo de multicast atômico sujeito a falhas bizantinas

Trabalho de conclusão de curso apresentado à Faculdade de Computação da Universidade Federal de Uberlândia, como parte dos requisitos exigidos para a obtenção título de Bacharel em Ciência da Computação.

Trabalho aprovado. Uberlândia, Brasil, 01 de novembro de 2016:

Paulo Rodolfo da Silva Leite Coelho
Orientador

Professor

Professor

Uberlândia, Brasil
2024

Resumo

Este trabalho apresenta uma otimização do protocolo *ByzCast*, um algoritmo de *multicast* atômico voltado para sistemas distribuídos sujeitos a falhas bizantinas. O *ByzCast* tradicional depende de grupos auxiliares para organizar e rotear mensagens, o que aumenta a complexidade e demanda um número elevado de nós. A proposta deste estudo visa eliminar esses grupos auxiliares, resultando em uma estrutura mais leve e eficiente, com menor consumo de recursos e comunicação.

A implementação otimizada foi avaliada em um ambiente de experimentação com a plataforma *CloudLab*, onde foi comparada à versão original do protocolo. A análise de desempenho demonstrou que a nova versão apresenta uma performance variável: similar ou superior em alguns cenários, e inferior em outros. Esses resultados indicam que ajustes de implementação podem aprimorar a consistência dos ganhos de desempenho observados.

Os resultados confirmam que o objetivo principal de simplificar a estrutura do *ByzCast* foi atingido, mantendo-se a integridade do protocolo e a robustez contra falhas bizantinas. Como trabalhos futuros, sugere-se a adaptação dinâmica da topologia conforme a carga medida e investigações adicionais para otimizar o desempenho do protocolo.

Palavras-chave: ByzCast, Sistemas distribuídos, Multicast atômico, Falhas bizantinas.

Lista de ilustrações

Lista de abreviaturas e siglas

CDF	*Cumulative Distribution Function*
BFT-SMaRt	*Byzantine Fault-Tolerant State Machine Replication*
API	*Application Programming Interface*
LCA	*Lowest Common Ancestor*
CDF	*Cumulative Distribution Function*

Sumário

1 Introdução

Com o crescimento das aplicações baseadas em sistemas distribuídos, como bancos de dados em nuvem e redes de blockchain, garantir a resiliência e a escalabilidade dessas aplicações se tornou um dos principais desafios da computação moderna, como proposto por Tanenbaum e Steen (2017). Esses sistemas estão sujeitos a diferentes tipos de falhas, desde falhas simples de comunicação até falhas maliciosas, o que exige o desenvolvimento de soluções robustas para garantir a continuidade dos serviços. À medida que essas aplicações crescem em escala, a necessidade de garantir resiliência, escalabilidade e tolerância a falhas torna-se ainda mais crítica.

Sistemas distribuídos desempenham um papel fundamental no suporte a uma ampla gama de aplicações modernas. Sua capacidade de distribuir carga de trabalho entre vários nós oferece alta disponibilidade, essencial para serviços que exigem operação ininterrupta, como plataformas de *streaming*, redes sociais e sistemas de pagamento. Além disso, a escalabilidade desses sistemas permite que empresas ampliem seus serviços sem comprometer o desempenho.

Sistemas distribuídos são compostos por múltiplos computadores interconectados, também chamados de nós, que colaboram para atingir um objetivo comum (TANENBAUM; STEEN, 2017). Esses sistemas proporcionam benefícios como a distribuição de carga e a alta disponibilidade, mas também trazem desafios, como a coordenação entre nós e a tolerância a falhas (COULOURIS et al., 2012). Garantir a consistência e confiabilidade dos dados em cenários com falhas é um dos maiores desafios. Em casos onde nós se comportam de maneira incorreta ou maliciosa, ocorrem falhas bizantinas, exigindo protocolos capazes de lidar com comportamentos arbitrários, mantendo o sistema funcional, como descrito por Lamport, Shostak e Pease (1982).

Um conceito central em sistemas distribuídos é o *multicast*, que permite que uma mensagem seja enviada simultaneamente para múltiplos nós (TANENBAUM; STEEN, 2017). Dentro desse contexto, o *multicast* atômico é uma técnica que garante que todas as réplicas em um sistema recebam as mensagens na mesma ordem e de maneira confiável, independentemente de falhas no sistema. Isso é particularmente importante em cenários onde a ordem das mensagens pode afetar diretamente a consistência dos dados. O *multicast* atômico é fundamental para implementar replicação de máquinas de estado, uma técnica que permite que múltiplas cópias de um serviço sejam executadas em paralelo em diferentes máquinas. Nessa abordagem, cada réplica do serviço processa as mesmas requisições na mesma ordem, garantindo consistência entre todas as instâncias, mesmo diante de falhas. A execução é determinística, ou seja, as réplicas sempre alcançam o mesmo

resultado quando submetidas às mesmas entradas, assegurando que, mesmo se algumas réplicas falharem, o serviço continue disponível.

O *ByzCast* é um protocolo de *multicast* atômico desenvolvido para operar em sistemas distribuídos sujeitos a falhas bizantinas, como apresentado por Coelho et al. (2018). Sua função é garantir que as mensagens sejam entregues de forma consistente entre os nós, mesmo na presença de comportamentos maliciosos ou falhas arbitrárias. O *ByzCast* é baseado no protocolo *BFT-SMaRt* (*Byzantine Fault-Tolerant State Machine Replication*) (BESSANI; SOUSA; ALCHIERI, 2014), um protocolo de replicação de máquinas de estado tolerante a falhas bizantinas. Na replicação de máquinas de estado, cada réplica processa as mesmas requisições na mesma ordem, assegurando consistência entre as instâncias e garantindo a continuidade do serviço em caso de falhas. Para garantir a ordenação das mensagens, o *ByzCast* organiza os nós em dois tipos de grupos: **grupos auxiliares** e **grupos de destino**. Os grupos auxiliares têm a função de ajudar na organização e roteamento das mensagens, enquanto os grupos de destino são responsáveis por processá-las. Embora essa abordagem seja funcional, ela requer um número elevado de nós, o que aumenta a complexidade e os custos de implementação, além de impactar no desempenho do protocolo.

Este trabalho propõe uma reimplementação do protocolo *ByzCast*, eliminando os grupos auxiliares. Na nova abordagem, apenas os grupos de destino serão mantidos, e eles serão responsáveis tanto pelo roteamento quanto pelo processamento das mensagens. Ao remover os grupos auxiliares, espera-se simplificar o protocolo, reduzir o número de nós necessários e, consequentemente, diminuir o *overhead* de comunicação e os recursos computacionais exigidos, sem comprometer a robustez e a tolerância a falhas bizantinas do sistema.

A proposta de reimplementação do protocolo *ByzCast* sem os grupos auxiliares visa não apenas simplificar a arquitetura do protocolo, mas também reduzir significativamente os custos de implantação e operação em ambientes distribuídos. Ao remover os grupos auxiliares, o número de nós necessários para a operação do protocolo é reduzido, o que implica em menor utilização de recursos computacionais e de rede, resultando em uma solução mais leve e acessível. A diminuição da complexidade também reflete diretamente na facilidade de configuração e manutenção, uma vez que o número de componentes a serem gerenciados é menor.

Essa abordagem pode beneficiar especialmente cenários em que a escalabilidade é crítica, mas os recursos são limitados. Ela permite uma maior flexibilidade na escolha das estruturas de sobreposição, enquanto a versão atual mantém uma configuração mais rígida e dependente de grupos auxiliares. Em Oliveira (2023), foi proposta uma automatização do processo de implantação de um protocolo *multicast* tolerante a falhas bizantinas, mostrando que a redução no número de nós e na complexidade do protocolo pode acelerar

o processo de testes e validação, além de minimizar o risco de erros durante a configuração. De maneira semelhante, espera-se que a reimplementação do *ByzCast* sem os grupos auxiliares traga benefícios tangíveis na redução do *overhead* de comunicação e no tempo necessário para implantar novas topologias.

Como futuras melhorias, vislumbra-se a possibilidade de uma atualização da topologia sob demanda, ou seja, adaptando automaticamente a arquitetura do protocolo de acordo com as mudanças nas condições de carga ou falhas do sistema. A proposta de remover os grupos auxiliares é um primeiro passo em direção a essa flexibilidade, ao simplificar a topologia e reduzir o número de nós gerenciados. Com menos elementos envolvidos, torna-se mais viável introduzir mecanismos dinâmicos de reconfiguração, possibilitando que o sistema se adapte rapidamente a novos cenários, mantendo a robustez e a eficiência operacional.

1.1 Objetivos

O objetivo principal deste trabalho é desenvolver uma versão otimizada do protocolo *ByzCast* sem o uso de grupos auxiliares, visando simplificar sua estrutura e reduzir os custos operacionais, sem comprometer a robustez contra falhas bizantinas.

Os objetivos específicos incluem:

- Reimplementar o protocolo *ByzCast*, eliminando os grupos auxiliares, tornando-o mais simples e eficiente;
- Comparar o desempenho do protocolo modificado com a versão original do *ByzCast*, utilizando métricas como latência, *throughput* e escalabilidade;
- Analisar o impacto da remoção dos grupos auxiliares na comunicação e no *overhead* de rede;
- Validar a eficácia do protocolo modificado, assegurando que os principais aspectos do *atomic multicast*, como a entrega ordenada de mensagens e a consistência do estado entre as réplicas, sejam preservados.

1.2 Justificativa

A implementação original do *ByzCast*, embora eficaz, apresenta uma estrutura mais complexa devido ao uso de grupos auxiliares para organizar e rotear mensagens. Em ambientes com recursos limitados, essa abordagem pode ser custosa e desnecessária. A proposta de remover os grupos auxiliares visa simplificar a topologia e reduzir os recursos computacionais e de rede necessários, sem comprometer aspectos fundamentais

do *multicast* atômico, como a preservação da ordem das mensagens e a consistência do estado entre as réplicas. Isso tornará o protocolo mais acessível e eficiente, mantendo a integridade do sistema.

2 Revisão Bibliográfica

Este capítulo apresenta os conceitos fundamentais necessários para a compreensão do trabalho. Serão abordados temas como consenso em sistemas distribuídos, *atomic broadcast*, *atomic multicast*, as diferentes maneiras de organizar nós em redes distribuídas, e uma análise do protocolo *ByzCast*. Esses conceitos formam a base teórica que sustenta o desenvolvimento e a reimplementação proposta neste trabalho.

2.1 Consenso

Em sistemas distribuídos, o problema do consenso é um dos mais estudados e complexos, sendo fundamental para garantir que diferentes nós de uma rede cheguem a um acordo comum, mesmo na presença de falhas ou comportamentos inesperados. O objetivo do consenso é fazer com que os nós corretos decidam sobre um único valor, independentemente de possíveis falhas na comunicação ou da existência de nós que possam se comportar de maneira maliciosa ou arbitrária.

2.1.1 O Problema do Consenso

O problema do consenso pode ser formalmente definido pela necessidade de um grupo de nós distribuídos chegar a uma decisão comum em relação a um valor ou ação. Este problema se torna especialmente difícil em ambientes onde pode haver falhas bizantinas, isto é, situações em que alguns nós se comportam de forma arbitrária, potencialmente maliciosa, comprometendo a confiabilidade do sistema.

Para garantir que o consenso seja alcançado em um sistema distribuído, três propriedades devem ser satisfeitas (LYNCH, 1996):

- **Término**: O protocolo deve garantir que todos os nós corretos eventualmente tomem uma decisão, ou seja, cheguem a uma conclusão em um tempo finito.
- **Validade**: Se todos os nós corretos propõe um valor, então todos os nós corretos devem decidir este valor.
- **Acordo**: Todos os nós não defeituosos devem concordar com o mesmo valor ao final do processo.

Essas propriedades são essenciais para garantir a consistência e a continuidade do sistema distribuído, principalmente em cenários onde há falhas bizantinas. Protocolos

de consenso como o **Paxos** (LAMPORT, 2001) e o **PBFT** (*Practical Byzantine Fault Tolerance*) (CASTRO; LISKOV et al., 1999) são amplamente utilizados para resolver esse problema em diferentes contextos.

Em muitos sistemas distribuídos, o problema do consenso precisa lidar com falhas bizantinas, que ocorrem quando um ou mais nós agem de forma arbitrária ou maliciosa. Essas falhas tornam o problema do consenso mais desafiador, já que os protocolos precisam garantir que a rede alcance um acordo comum, mesmo que parte dos nós esteja se comportando de maneira incorreta ou tentando comprometer o processo. Para isso, é necessário que os algoritmos sejam robustos o suficiente para lidar com esses comportamentos imprevisíveis.

O trabalho pioneiro de Lamport, Shostak e Pease (1982) sobre o Problema dos Generais Bizantinos introduziu o conceito de falhas bizantinas e destacou a complexidade de resolver o consenso em sistemas onde nós podem enviar informações contraditórias ou tentar enganar outros nós. O protocolo *Practical Byzantine Fault Tolerance* (PBFT), desenvolvido por Castro, Liskov et al. (1999), foi uma das soluções mais influentes para alcançar o consenso em redes com falhas bizantinas, oferecendo uma solução eficiente para redes distribuídas tolerantes a falhas maliciosas.

2.1.2 Consenso no Contexto do *ByzCast*

No contexto do protocolo *ByzCast*, o consenso é utilizado para garantir que todas as réplicas em um sistema distribuído recebam e processem as mensagens na mesma ordem. O *ByzCast* adota o modelo de *multicast* atômico para garantir que as mensagens sejam entregues a todos os nós de maneira consistente, mesmo que parte do sistema esteja comprometida por falhas bizantinas. O protocolo baseia-se em técnicas de replicação de máquinas de estado, onde cada réplica processa as mesmas requisições na mesma ordem, garantindo que todas mantenham um estado consistente (COELHO et al., 2018).

A importância do consenso no *ByzCast* reside em sua capacidade de manter a integridade do sistema e evitar inconsistências entre as réplicas, permitindo que o sistema continue a operar corretamente, independentemente de possíveis falhas em parte dos nós. Isso é particularmente relevante para aplicações críticas, como redes de *blockchain* e sistemas financeiros distribuídos, que dependem de uma execução confiável e de alta disponibilidade.

2.2 *Atomic Broadcast*

O *atomic broadcast* é essencial em sistemas distribuídos, permitindo que mensagens sejam entregues a todos os nós do sistema de maneira ordenada e confiável. Ele é uma extensão do *reliable broadcast*, uma técnica que garante que, se uma mensagem é entregue

a um nó correto, eventualmente será entregue a todos os nós corretos. No entanto, o *reliable broadcast* não impõe uma ordem de entrega das mensagens, o que pode levar a inconsistências entre as réplicas de um sistema distribuído.

O *atomic broadcast*, por sua vez, resolve esse problema ao garantir não apenas que todas as mensagens sejam entregues, mas também que elas sejam recebidas na mesma ordem por todos os nós. Isso é crucial em sistemas que utilizam replicação de máquinas de estado, onde a ordem das operações deve ser preservada para que todas as réplicas mantenham estados consistentes (COULOURIS et al., 2012).

2.2.1 Primitivas do *Atomic Broadcast*

Para implementar o *atomic broadcast*, é necessário definir algumas primitivas básicas, que são comumente utilizadas para controlar a comunicação entre os nós de um sistema distribuído (LYNCH, 1996):

- ***Broadcast*(m)**: Essa primitiva permite que um nó envie uma mensagem m para todos os outros nós do sistema.
- ***Deliver*(m)**: Essa primitiva garante que uma mensagem m enviada a um nó será eventualmente entregue e processada por todos os nós corretos.

Essas primitivas são suficientes para o *reliable broadcast*, mas o *atomic broadcast* introduz a necessidade de garantir que as mensagens sejam entregues na mesma ordem a todos os nós. Para alcançar isso, o protocolo deve garantir as seguintes propriedades adicionais (COULOURIS et al., 2012):

- **Entrega Total**: Se dois nós entregam duas mensagens, ambas as mensagens devem ser entregues na mesma ordem em ambos os nós.
- **Entrega Confiável**: Se uma mensagem é entregue a um nó correto, ela deve ser eventualmente entregue a todos os nós corretos.

2.2.2 Relação entre *Reliable Broadcast* e *Atomic Broadcast*

O *atomic broadcast* pode ser visto como uma extensão do *reliable broadcast*, com a adição da propriedade de ordem total. O *reliable broadcast* garante apenas que uma mensagem seja entregue a todos os nós corretos, mas não faz nenhuma suposição sobre a ordem de entrega das mensagens. Em contrapartida, o *atomic broadcast* garante que, além de todas as mensagens serem entregues, elas serão entregues na mesma ordem em todos os nós, garantindo consistência global no estado das réplicas (SCHNEIDER, 1990).

Essa evolução é particularmente importante em sistemas distribuídos que exigem alta consistência, como os sistemas que utilizam replicação de máquinas de estado. Nesses sistemas, a ordem de entrega das mensagens é crucial para garantir que todas as réplicas permaneçam sincronizadas e mantenham o mesmo estado, independentemente das falhas ou da presença de nós maliciosos.

2.2.3 Propriedades do *Atomic Broadcast*

Um protocolo de *atomic broadcast* deve garantir as seguintes propriedades (RODRIGUES; RAYNAL, 2000):

- **Validade**: Se um participante correto transmite uma mensagem, todos os participantes corretos eventualmente a receberão.
- **Acordo Uniforme**: Se um participante correto recebe uma mensagem, então todos os participantes corretos eventualmente receberão essa mensagem.
- **Integridade Uniforme**: Cada mensagem é recebida por cada participante no máximo uma vez, e apenas se tiver sido previamente transmitida.
- **Ordem Total Uniforme**: As mensagens são ordenadas de forma consistente; se um participante correto recebe a mensagem 1 antes da mensagem 2, todos os outros participantes também devem receber a mensagem 1 antes da 2.

Essas propriedades garantem que todas as réplicas em um sistema distribuído permaneçam consistentes, processando as mesmas operações na mesma ordem, mesmo que parte dos nós falhe ou se comporte de maneira inesperada.

2.2.4 *Atomic Broadcast* no Contexto do *ByzCast*

No *ByzCast*, o *atomic broadcast* é utilizado para assegurar que todas as mensagens sejam entregues a todas as réplicas na mesma ordem, mantendo a consistência do estado entre elas. Isso é especialmente importante em sistemas distribuídos tolerantes a falhas bizantinas, onde nós maliciosos podem tentar alterar a ordem das mensagens para comprometer o sistema. O uso de *atomic broadcast* no *ByzCast* garante que o sistema continue operando de maneira consistente e confiável, independentemente da presença de nós maliciosos (COELHO et al., 2018).

2.3 *Atomic Multicast*

O *atomic multicast* é uma extensão do *atomic broadcast* que permite a comunicação em um sistema distribuído, mas, ao contrário do *broadcast*, onde todas as mensagens são

enviadas para todos os nós, o *multicast* permite que mensagens sejam enviadas apenas para subconjuntos de nós. Isso é particularmente útil em sistemas distribuídos onde diferentes grupos de nós desempenham funções específicas e nem todas as mensagens precisam ser recebidas por todos os nós do sistema (DÉFAGO; URBAN; SCHIPER, 2003).

No *atomic multicast*, a entrega de mensagens deve respeitar as seguintes propriedades, que são semelhantes às propriedades de *atomic broadcast*, com a adição de especificidades referentes ao grupo de destino (DÉFAGO; URBAN; SCHIPER, 2003):

- **Entrega Confiável (Validade)**: Se uma mensagem é transmitida por um processo para um grupo de nós, todos os processos corretos pertencentes ao grupo de destino eventualmente receberão a mensagem.
- **Acordo Uniforme**: Se uma mensagem é entregue a um processo correto de um grupo, então todos os processos corretos do grupo eventualmente receberão essa mensagem.
- **Integridade Uniforme**: Cada mensagem é entregue a um processo no máximo uma vez, e apenas se ela tiver sido previamente transmitida.
- **Ordem Total Uniforme**: Para cada grupo de destino, as mensagens devem ser entregues na mesma ordem a todos os membros corretos do grupo. Isso garante que, se um processo correto entrega a mensagem 1 antes da mensagem 2, todos os outros processos do grupo também entregarão a mensagem 1 antes da 2.

2.3.1 Contexto do *Multicast* em Sistemas Distribuídos

O *atomic multicast* é especialmente importante em sistemas distribuídos que utilizam replicação parcial, onde diferentes réplicas de um sistema podem estar localizadas em subconjuntos de nós que realizam tarefas específicas. Ao contrário do *atomic broadcast*, onde todas as mensagens são entregues a todos os nós, o *atomic multicast* permite que as mensagens sejam enviadas apenas para os grupos que realmente precisam processá-las. Isso reduz a sobrecarga de comunicação e melhora a eficiência do sistema (DÉFAGO; URBAN; SCHIPER, 2003).

Em sistemas que utilizam replicação de máquinas de estado, o *atomic multicast* é fundamental para garantir que as réplicas mantenham um estado consistente dentro de cada grupo. Além disso, ele permite que diferentes grupos de réplicas processem suas respectivas mensagens de forma independente, enquanto ainda garantem a consistência interna de cada grupo.

2.3.2 Comparação entre *Atomic Broadcast* e *Atomic Multicast*

A principal diferença entre *atomic broadcast* e *atomic multicast* reside no escopo de entrega das mensagens. No *atomic broadcast*, todas as mensagens são entregues a todos os nós do sistema, enquanto no *atomic multicast*, as mensagens são entregues apenas a subconjuntos de nós específicos, chamados de grupos de destino. Essa distinção permite que o *multicast* seja mais eficiente em termos de comunicação, pois evita que nós que não necessitam de uma determinada mensagem sejam sobrecarregados com o seu processamento.

Outra diferença importante é que, enquanto o *atomic broadcast* assegura uma ordem total global de mensagens, o *atomic multicast* garante uma ordem total dentro de cada grupo de destino. Isso significa que diferentes grupos podem receber e processar mensagens em ordens diferentes, desde que dentro de cada grupo a ordem seja consistente. Em casos onde dois ou mais grupos compartilham mensagens em comum, essas mensagens seguem a mesma ordem em todos os grupos da interseção, assegurando que a consistência seja mantida para mensagens que transitam entre grupos. Essa flexibilidade torna o *atomic multicast* mais adequado para sistemas distribuídos com topologias hierárquicas ou grupos especializados de nós, onde é necessário equilibrar a independência entre grupos com a consistência nas mensagens compartilhadas.

2.3.3 Atomic Multicast no Contexto do *ByzCast*

No protocolo *ByzCast*, o *atomic multicast* é um componente central que garante a entrega de mensagens de maneira consistente para subconjuntos específicos de nós, chamados de grupos de destino. A principal função do *ByzCast* é assegurar que, mesmo na presença de falhas bizantinas, as mensagens sejam entregues de forma ordenada e confiável dentro de cada grupo de destino, preservando a consistência do estado entre as réplicas (COELHO et al., 2018).

Além disso, o uso de *atomic multicast* no *ByzCast* assegura que, mesmo se alguns nós agirem de maneira maliciosa, os grupos de destino corretos ainda receberão as mensagens na ordem correta, garantindo que o sistema como um todo se mantenha consistente e confiável. A eliminação dos grupos auxiliares na reimplementação do *ByzCast*, como proposta neste trabalho, visa simplificar essa estrutura sem comprometer a robustez e a consistência proporcionadas pelo *atomic multicast*.

Suponha que no *ByzCast* original, com a topologia de três níveis ilustrada na Figura 1, duas mensagens sejam enviadas: a primeira destinada aos grupos de destino 1 e 2, e a segunda destinada aos grupos de destino 2 e 4. O *ByzCast* utiliza o conceito de ***Lowest Common Ancestor* (LCA)**, ou "ancestral comum mais baixo", para definir o ponto mais próximo na hierarquia em que as mensagens devem ser processadas e ordenadas antes

de serem enviadas aos grupos de destino. Assim, para a primeira mensagem, que precisa alcançar os grupos de destino 1 e 2, o LCA é o grupo global 2, que conecta diretamente esses grupos de destino. Portanto, a primeira mensagem é enviada diretamente ao grupo global 2, onde a ordem é definida antes de ser encaminhada aos grupos de destino 1 e 2.

Para a segunda mensagem, cujo LCA entre os grupos de destino 2 e 4 é o grupo global principal, ela é primeiramente enviada ao nível superior, onde sua ordem é estabelecida. Em seguida, a mensagem é distribuída aos grupos globais 2 e 3, que por sua vez a retransmitem para os grupos de destino 2 e 4. O uso do LCA no *ByzCast* permite otimizar o processo de ordenação ao evitar que todas as mensagens passem pelo grupo global principal, distribuindo-as de forma eficiente e garantindo que cada grupo de destino processe as mensagens na mesma sequência, mantendo a consistência do sistema.

2.4 O Protocolo *ByzCast*

O *ByzCast* é um protocolo de *multicast* atômico tolerante a falhas bizantinas, desenvolvido para garantir a entrega confiável de mensagens entre grupos de nós em um sistema distribuído, mesmo na presença de comportamentos maliciosos ou arbitrários. O *ByzCast* baseia-se no protocolo *BFT-SMaRt* (*Byzantine Fault-Tolerant State Machine Replication*) (BESSANI; SOUSA; ALCHIERI, 2014), que utiliza replicação de máquinas de estado para garantir que todas as réplicas mantenham consistência de estado, mesmo diante de falhas (COELHO et al., 2018). A replicação de máquinas de estado é essencial para sistemas distribuídos críticos, onde múltiplas cópias de um serviço devem operar em paralelo, processando as mesmas requisições na mesma ordem.

O *ByzCast* implementa o *atomic multicast*, permitindo que mensagens sejam entregues a subconjuntos específicos de nós, chamados grupos de destino. Cada grupo é responsável por receber e processar as mensagens, e o protocolo garante que a ordem de entrega das mensagens seja a mesma em todas as réplicas de um grupo, preservando a consistência entre elas.

2.4.1 Organização dos Nós no *ByzCast*

No *ByzCast*, os nós são organizados em duas categorias principais: **grupos auxiliares** e **grupos de destino**. Essa organização segue uma estrutura hierárquica, onde os grupos auxiliares ajudam a organizar e rotear as mensagens, garantindo que elas sejam entregues aos grupos de destino na ordem correta (COELHO et al., 2018). Essa abordagem hierárquica tem a vantagem de reduzir a sobrecarga de comunicação, já que nem todos os nós precisam se comunicar diretamente, como ocorreria em uma topologia *all-to-all*.

Os grupos de destino, por sua vez, são responsáveis por processar as mensagens recebidas e executar as requisições associadas. Cada grupo é composto por 3f+1 réplicas,

onde f é o número máximo de nós bizantinos tolerados pelo sistema. Essa estrutura de replicação garante que, mesmo que alguns nós falhem ou ajam de forma maliciosa, o grupo como um todo ainda possa alcançar consenso e manter a consistência do serviço.

2.4.2 Operação do *ByzCast*

O funcionamento do *ByzCast* depende de dois tipos principais de comunicação: **multicast atômico** e **broadcast atômico**. O *multicast atômico* é utilizado para enviar mensagens para subconjuntos específicos de nós (grupos de destino), enquanto o *broadcast atômico* é usado dentro de cada grupo para garantir que todas as réplicas recebam as mensagens na mesma ordem.

Para garantir a entrega ordenada das mensagens entre múltiplos grupos de destino, o *ByzCast* organiza os nós em uma árvore de sobreposição (*overlay tree*). As folhas dessa árvore são os grupos de destino, enquanto os nós internos são os grupos auxiliares. Quando uma mensagem precisa ser enviada para vários grupos, ela é primeiramente ordenada pelo grupo auxiliar comum mais baixo na árvore e, em seguida, propagada para os grupos de destino, garantindo que a ordem seja mantida em todos os grupos.

A figura 1 apresenta uma topologia de três níveis utilizada na implementação original do *ByzCast*. Assumindo uma configuração em que cada grupo é composto por quatro nós, seria necessário um total de 28 nós para suportar essa arquitetura. Ao manter o mesmo número de nós por grupo, essa topologia iria totalizar apenas 16 nós ao ser representada na nova implementação.

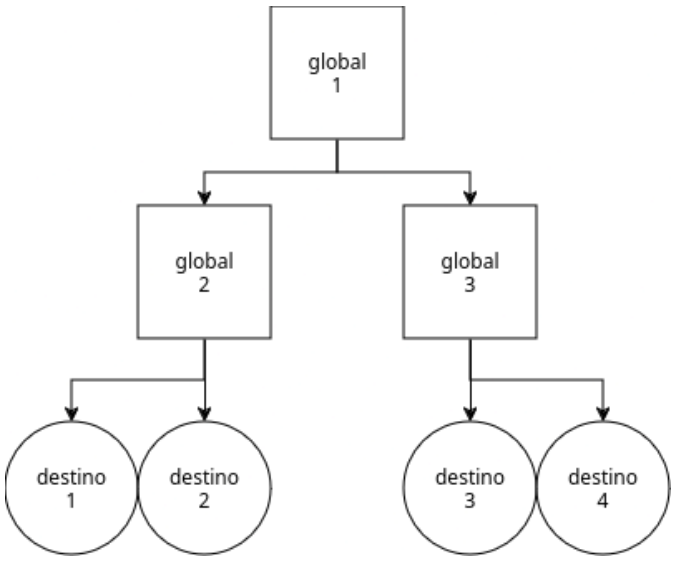


Figura 1 – Topologia em três níveis da implementação original.
Fonte: autoral.

3 Desenvolvimento

Para compreender as melhorias propostas para o protocolo *ByzCast*, é essencial entender o contexto de sua estrutura hierárquica e os desafios associados a essa organização. A seguir, a configuração hierárquica original do protocolo é detalhada, além dos pontos de complexidade e a proposta de simplificação.

3.1 Estrutura Hierárquica do ByzCast e Proposta de Melhorias

O protocolo *ByzCast* implementa uma estrutura hierárquica para gerenciar a comunicação e ordenação de mensagens entre diferentes grupos de nós. Essa hierarquia é composta por grupos auxiliares e grupos de destino, organizados em uma estrutura de árvore, com o objetivo de garantir a consistência e a entrega ordenada de mensagens em sistemas distribuídos sujeitos a falhas bizantinas, como descrito por Coelho et al. (2018). Na estrutura hierárquica, cada grupo de destino é responsável por processar as mensagens que recebe, enquanto os grupos auxiliares têm a função de organizar e direcionar as mensagens para os grupos de destino corretos.

Para organizar a entrega de mensagens de maneira eficiente, o *ByzCast* utiliza o conceito de ***Lowest Common Ancestor* (LCA)**, ou "ancestral comum mais baixo", que identifica o ponto mais próximo da árvore onde as mensagens podem ser ordenadas antes de serem transmitidas aos grupos de destino. Por exemplo, se uma mensagem precisa ser enviada para múltiplos grupos de destino, o LCA desses grupos será o responsável por definir a ordem das mensagens, evitando a sobrecarga de ordenar cada mensagem individualmente em um nível centralizado. Essa estratégia permite que a ordem seja mantida de maneira distribuída ao longo da árvore hierárquica, com cada grupo de destino processando as mensagens na mesma sequência, conforme estabelecido pelo grupo LCA.

No entanto, a organização hierárquica do *ByzCast* apresenta alguns pontos que podem ser melhorados. O uso de grupos auxiliares adiciona complexidade ao protocolo, pois cada mensagem precisa ser roteada e ordenada por múltiplos níveis antes de alcançar os grupos de destino. Essa abordagem, embora eficaz em termos de consistência, pode gerar uma sobrecarga considerável de comunicação e processamento, especialmente em cenários de grande escala.

A proposta de reimplementação neste trabalho busca simplificar a estrutura do *ByzCast* eliminando os grupos auxiliares e permitindo que apenas os grupos de destino executem as funções de roteamento e ordenação de mensagens. Com essa simplificação, espera-se reduzir a quantidade de nós necessários, além de diminuir a sobrecarga de comu-

nicação e processamento, tornando o protocolo mais eficiente e direto, sem comprometer a robustez da entrega de mensagens em ordem.

3.1.1 Estrutura inicial e limitações

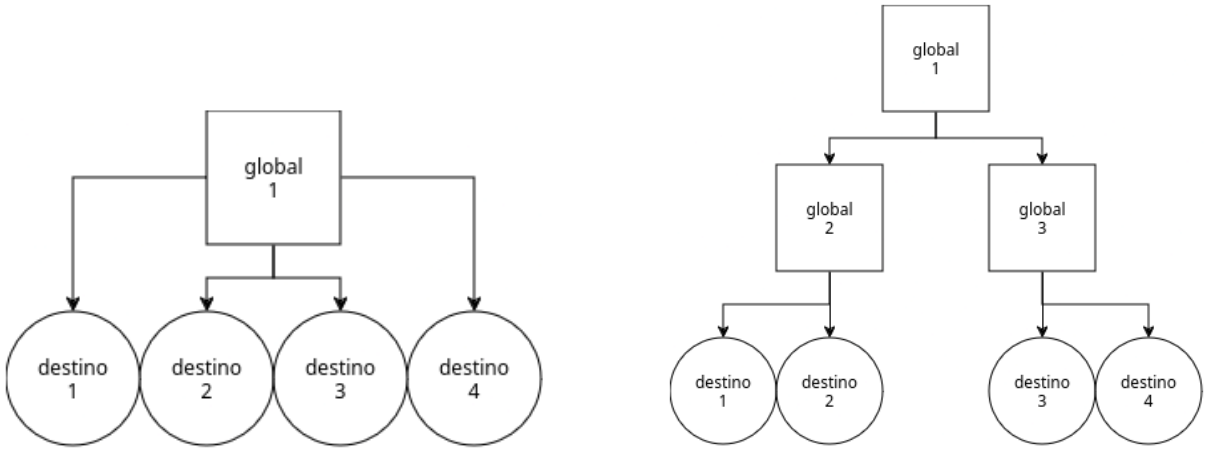
A versão original do *ByzCast* apresentava uma estrutura hierárquica rígida, sendo compatível apenas com duas topologias específicas: uma de dois níveis e outra de três níveis. Na topologia de dois níveis, a implementação incluía um grupo global conectado a quatro grupos locais. Nesse caso, o LCA de todas as mensagens entre os grupos era o próprio grupo global, sendo necessário que todas as comunicações passassem por ele antes de alcançar o grupo de destino. Essa configuração, embora funcional, implicava em uma certa rigidez no fluxo de mensagens, aumentando a carga sobre o grupo global.

Na nova implementação proposta, todos os grupos são tratados como locais, eliminando a necessidade de um grupo global intermediário. Em vez disso, um dos quatro grupos locais é configurado para se conectar diretamente aos outros três, mantendo a conectividade sem a sobrecarga de um intermediário central. Essa alteração não apenas simplifica a estrutura, mas também reduz o número total de nós de 20 para 16, resultando em uma economia de quatro nós sem perda de flexibilidade ou aumento na complexidade de comunicação. Isso pode ser visualizado pelas figuras 3 e 2.

Na configuração de três níveis, conforme visto pelas figuras 3 e 2, as diferenças são ainda mais significativas. Originalmente, o protocolo necessitava de um grupo global adicional para gerenciar a comunicação entre os grupos de segundo nível e os grupos de destino, elevando o número total de nós para 28. A nova organização elimina esse grupo intermediário e todos os grupos de destino operam de maneira autônoma. Com essa simplificação, o total de nós necessários é reduzido para 16, representando uma economia de 12 nós.

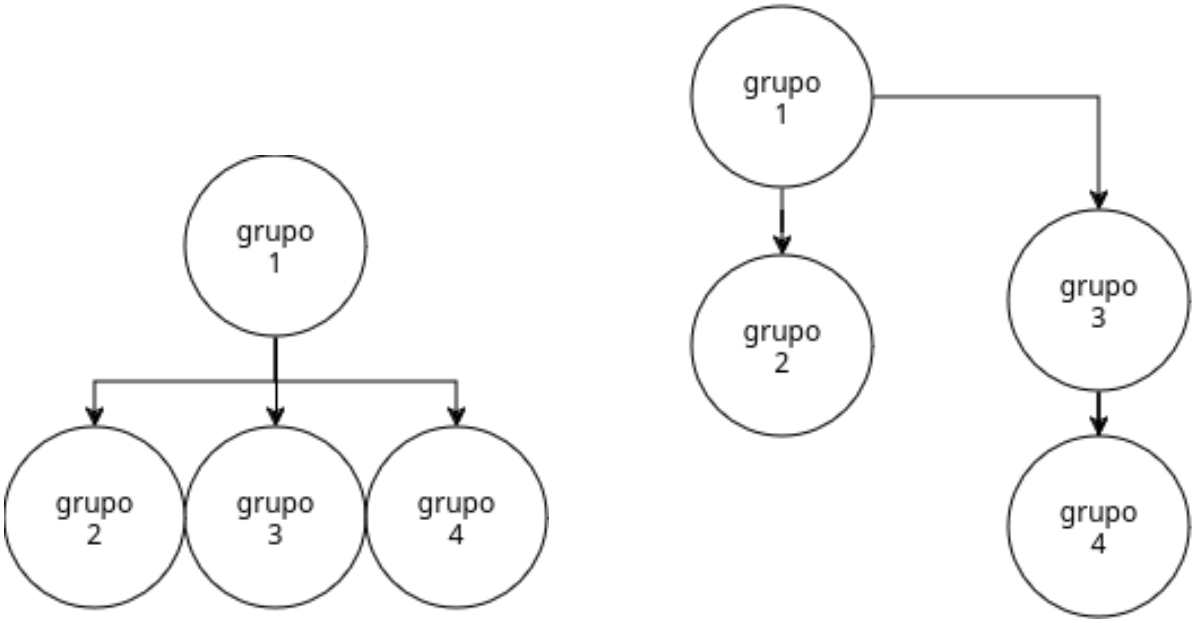
Além da economia estrutural, a topologia de três níveis na nova implementação favorece a localidade das comunicações. Em cenários onde as mensagens são predominantemente trocadas entre pares específicos de grupos, essa estrutura permite utilizar LCAs distintos para cada par de grupos. Por exemplo, quando as mensagens são trocadas entre duas duplas de grupos diferentes, a comunicação requer que as mensagens passem por apenas dois nós para alcançar o destino, melhorando a eficiência e reduzindo a latência.

Portanto, ao simplificar a hierarquia e permitir uma organização mais flexível, a nova implementação do *ByzCast* oferece melhorias substanciais na economia de nós e na eficiência de comunicação, mantendo a robustez e a consistência exigidas para aplicações distribuídas sujeitas a falhas bizantinas.



(a) Topologia de 2 níveis. (b) Topologia de 3 níveis.

Figura 2 – Topologias de 2 e 3 níveis da implementação original.
Fonte: autoral.



(a) Topologia de 2 níveis. (b) Topologia de 3 níveis

Figura 3 – Topologias de 2 e 3 níveis da nova implementação.
Fonte: autoral.

3.2 Reimplementação

O desenvolvimento deste trabalho foi planejado como um processo gradual, fundamentado em um entendimento profundo do código original do protocolo *ByzCast*. A análise inicial do código, escrito em Java 8, revelou uma estrutura rígida, com suporte limitado a apenas duas topologias: uma de dois níveis e outra de três níveis. A complexidade do código e a falta de flexibilidade dificultaram o entendimento e a adaptação do protocolo, sendo necessários estudos detalhados para desvendar sua lógica e funcionamento.

Optou-se por uma reimplementação completa em Java 21, aproveitando as melhorias dessa versão da linguagem para simplificar a arquitetura e eliminar os grupos auxiliares. Essa abordagem visa tornar os grupos de destino responsáveis tanto pelo roteamento quanto pelo processamento de mensagens, resultando em um sistema mais leve e adaptável, além de reduzir o número de nós necessários.

3.2.1 Visão Geral da Reimplementação

O processo de reimplementação do protocolo *ByzCast* exigiu uma análise cuidadosa do código original, que estava escrito em Java 8. Desde o início, ficou evidente que o código apresentava um alto nível de complexidade, em parte devido à estrutura rígida e às limitações das funcionalidades disponíveis na versão de Java utilizada. Além disso, o protocolo dependia da API da biblioteca *BFT-SMaRt* (BESSANI; SOUSA; ALCHIERI, 2014), cuja compreensão era crucial para a adaptação às novas diretrizes e simplificações propostas. A API do *BFT-SMaRt* oferece vários componentes que precisam ser usados de diferentes maneiras para gerenciar o ciclo de recebimento de mensagens, incluindo classes específicas para o processamento de mensagens e outras para personalizar o formato de resposta ao cliente final. Essa estrutura complexa da API demandou um estudo aprofundado, uma vez que o comportamento do protocolo *ByzCast* depende diretamente de como essas classes são integradas.

Com a decisão de reimplementar o sistema em Java 21, abriu-se a oportunidade de aproveitar funcionalidades mais avançadas da linguagem, o que não apenas simplificaria o código, mas também melhoraria sua legibilidade e manutenção. Uma das inovações que mais contribuiu para essa simplificação foi a introdução da palavra-chave *var*, que permite a inferência de tipo local. Essa funcionalidade reduziu a verbosidade do código, permitindo que o foco permanecesse nas operações principais do protocolo, sem a necessidade de declarações de tipo explícitas em cada linha. Como resultado, o código tornou-se mais conciso e legível, o que facilitou o processo de reimplementação.

Outro recurso importante foram os *streams*, que possibilitaram o processamento de dados de maneira mais funcional e eficiente. No contexto do *ByzCast*, onde há uma constante necessidade de manipulação de conjuntos de mensagens e nós, o uso de *streams* permitiu operações de filtragem, transformação e coleta de dados de forma mais simplificada e direta. Por exemplo, operações que antes exigiam múltiplos loops foram substituídas por uma única expressão de *stream*, reduzindo a complexidade do código e melhorando seu desempenho.

A reimplementação também incluiu um esforço substancial para melhorar a documentação. Na versão original, a falta de comentários e explicações detalhadas dificultava o entendimento do fluxo de mensagens e a lógica de roteamento, principalmente para desenvolvedores não familiarizados com o protocolo. Com isso em mente, foram introdu-

zidos *JavaDocs* abrangentes, oferecendo explicações sobre o propósito de cada classe e método, além de detalhes sobre parâmetros e exceções tratadas. Essa documentação não apenas facilita a manutenção futura, mas também serve como uma referência valiosa para entender a lógica subjacente do sistema, especialmente em pontos críticos, como a escolha do *Lowest Common Ancestor* (LCA) e a definição de regras de roteamento.

Portanto, a escolha de reimplementar o *ByzCast* em Java 21 foi guiada tanto pela necessidade de simplificar o código quanto pelo desejo de modernizar a base de código, aproveitando as novas funcionalidades da linguagem para criar uma implementação mais legível, robusta e documentada. A seguir, serão explorados os detalhes específicos da nova estrutura de classes, métodos principais e as adaptações realizadas para otimizar o funcionamento do protocolo.

3.2.2 Implementação do Código Principal

A implementação do código principal foi estruturada em torno de quatro classes principais: *ServerNode*, *RequestHandler*, *ServerReplier* e *ServerState*. Cada uma dessas classes desempenha um papel essencial para garantir que o protocolo funcione corretamente, respeitando as restrições de consistência e comunicação entre grupos de nós. A figure 4 representa uma visão geral das classes, cujos relacionamentos serão discutidos a seguir.

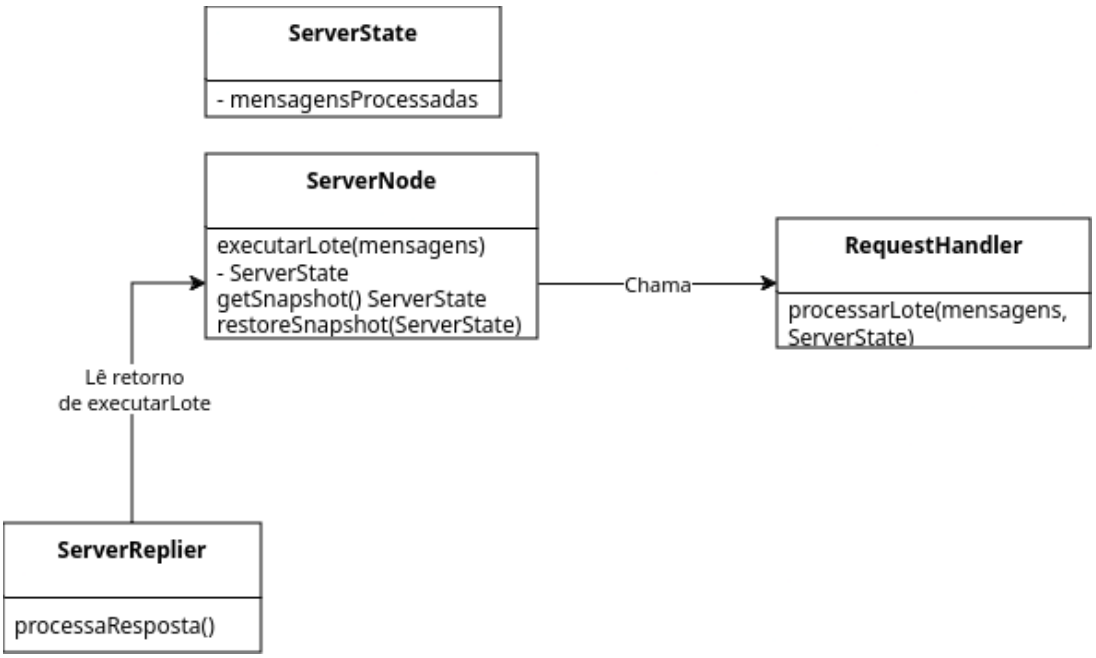


Figura 4 – Visão abstraída geral das classes.
Fonte: autoral.

A classe *ServerNode* foi projetada para adaptar-se à API do *BFT-SMaRt*, implementando os métodos obrigatórios *getSnapshot* e *restoreSnapshot*, que permitem capturar

e restaurar o estado do sistema para manter a consistência entre as réplicas. Além desses, a *ServerNode* também implementa métodos de processamento em lote e processamento individual de mensagens, essenciais para gerenciar o fluxo de dados conforme as necessidades do protocolo. Todos esses métodos — tanto os de *snapshot* quanto os de processamento — são executados por meio de uma interação direta com o *RequestHandler*, que centraliza a lógica de manipulação das mensagens e do estado. Dessa forma, o *ServerNode* atua como o ponto de entrada do sistema, delegando ao *RequestHandler* a maior parte do processamento, o que garante uma organização modular e facilita a manutenção da consistência entre as réplicas de um grupo.

A classe *ServerState* é responsável por manter o estado compartilhado do servidor, que deve ser idêntico entre as réplicas de um grupo. Esse estado é acessado e modificado por meio de métodos específicos que a *ServerState* expõe para outras classes, permitindo que operações como o processamento de mensagens e a atualização do sistema sejam realizadas de forma controlada e consistente. Instanciada dentro da *ServerNode*, a *ServerState* é também o objeto que é modificado durante a execução do método de restauração de *snapshot*, assegurando que o estado do sistema seja atualizado conforme necessário para manter a sincronização entre as réplicas. Dessa forma, a *ServerState* centraliza e gerencia o estado do servidor, funcionando como o núcleo de consistência para o protocolo.

A *ServerReplier* é utilizada como parte da API do *BFT-SMaRt* e foi configurada para ajustar as regras de resposta aos clientes. Ela contém uma lógica que permite o *delay* de respostas, essencial para lidar com o cenário onde todos os nós funcionais de um grupo enviam mensagens ao receber uma requisição. Quando uma mensagem é recebida de outro grupo em vez de um cliente, é necessário aguardar o recebimento de pelo menos $N - F$ instâncias dessa mensagem (onde N é o número total de nós e F é o número de falhas toleradas) antes de processá-la. A *ServerReplier* possibilita essa espera, já que, na API do *BFT-SMaRt*, um replier personalizado é necessário para "aguardar"o processamento sem enviar uma resposta imediata ao cliente. Essa lógica aparentemente complexa sobre as requisições não compromete as propriedades do algoritmo de *multicast* atômico sujeito a falhas bizantinas, pois o *BFT-SMaRt* assegura que a ordem de envio das mensagens seja a mesma para todas as réplicas, mantendo assim a integridade do sistema.

A classe *RequestHandler* centraliza a lógica de gerenciamento de mensagens e desempenha um papel essencial na coordenação das operações de *multicast* e no processamento das respostas finais para os clientes. Além de interagir com a *ServerState* e a *ServerReplier*, o *RequestHandler* também é responsável pelo envio de mensagens para outros grupos em cenários de *multicast*. Quando uma mensagem requer comunicação entre múltiplos grupos, o *RequestHandler* coordena o envio, garantindo que cada grupo receba e processe as mensagens de maneira sincronizada.

Ao processar uma mensagem recebida de outro grupo, o *RequestHandler* verifica

se ela já foi recebida $N - F$ vezes; caso contrário, marca-a como pendente e instrui o *ServerReplier* a aguardar antes de responder ao cliente. Quando o número necessário de confirmações é atingido, ele finaliza o processamento, informando o *ServerReplier* que a resposta está completa e pronta para ser enviada ao cliente.

Além disso, o *RequestHandler* coordena a geração da resposta final, considerando o conteúdo dos grupos envolvidos no *multicast*. Isso assegura que a resposta ao cliente reflita o estado consolidado de todos os grupos relevantes, mantendo a integridade e a consistência dos dados.

Essa organização de classes e lógica de processamento permite que o protocolo *ByzCast* opere de maneira eficiente, com suporte à consistência de estado e capacidade de lidar com a comunicação entre grupos sem comprometimento da segurança ou da ordem das mensagens.

3.2.2.1 A classe *RequestHandler*

Uma das principais dificuldades de implementação enfrentadas na classe *RequestHandler* foi o gerenciamento de requisições em lote. A API do *BFT-SMaRt* já oferece a capacidade de entregar requisições em lote para um nó , mas essas requisições consistem simplesmente em várias mensagens entregues de uma vez. A implementação original do *ByzCast* aplicava uma lógica que agrupava essas mensagens individuais em uma única mensagem, do ponto de vista do *BFT-SMaRt*, cujo conteúdo incluía várias mensagens individuais. Assim, essa "mensagem em lote"era tecnicamente uma única mensagem, mas com múltiplas mensagens encapsuladas em seu conteúdo.

Esse procedimento de agrupamento de mensagens em lote complica a implementação da lógica $N-F$, pois, ao receber uma mensagem de lote, torna-se necessário desagregar o lote, analisando cada mensagem individual para garantir o número necessário de confirmações antes de processá-las. Somente após essa análise é que o lote pode ser agregado novamente para a geração de uma resposta em lote. Além disso, um lote pode conter apenas algumas mensagens concluídas, exigindo que o sistema aguarde a conclusão de todas as mensagens antes de enviar uma resposta consolidada.

Essa complexidade se dá pela necessidade de lidar com dois tipos de mensagens: a individual e a em lote. A mensagem em lote "existe"apenas do ponto de vista do *ByzCast*; para o *BFT-SMaRt*, todas as mensagens enviadas em lote são tratadas como mensagens individuais, com seu conteúdo como um conjunto de bytes a serem interpretados.

No entanto, essa abordagem de mensagens em lote representa uma melhoria significativa para o protocolo *ByzCast*, pois permite que mais mensagens sejam enviadas de uma só vez, com menor latência na comunicação entre grupos. Essa otimização reduz o tempo necessário para a troca de mensagens e melhora o desempenho geral do sistema,

especialmente em cenários com alto volume de requisições.

Esse desafio na implementação de requisições em lote é amplificado por uma característica específica do *BFT-SMaRt*: os lotes entregues a diferentes nós não são garantidos de serem idênticos. Isso significa que dois grupos podem receber lotes distintos de mensagens, embora a ordem interna das mensagens seja consistente. Em essência, os lotes entregues a diferentes grupos nada mais são do que subconjuntos uns dos outros: um lote pode simplesmente conter mensagens adicionais ou estar pendente de algumas, mas a ordem de entrega entre as mensagens é sempre preservada.

Essa característica inviabilizou uma abordagem considerada inicialmente, que consistia em tratar as mensagens em lote, do ponto de vista do *ByzCast*, como uma única unidade, sujeita à mesma lógica de $N - F$ das mensagens individuais. Essa abordagem simplificada permitiria que, ao receber um lote o número necessário de vezes, todas as mensagens individuais nele contidas fossem automaticamente confirmadas como processadas. Contudo, como o *BFT-SMaRt* pode entregar lotes diferentes para certos nós, cada nó pode acabar gerando um conjunto de mensagens em lote ligeiramente distinto, o que impede que $N - F$ seja completado para alguns desses lotes. Esse comportamento demandou a implementação da abordagem atual, em que cada mensagem individual no lote precisa ser separada e tratada isoladamente, permitindo que a verificação $N - F$ ocorra de maneira precisa para cada mensagem específica.

Além de gerenciar o processamento de mensagens, a classe *RequestHandler* também foi configurada para gerar *logs* detalhados que registram cada etapa do ciclo de vida de uma mensagem. Esses *logs* documentam o status da mensagem desde o momento de seu recebimento inicial, passando pela contagem de confirmações até $N - F$, o processamento efetivo, o envio para outros grupos, e, finalmente, o retorno da resposta ao cliente. Esses registros detalhados foram cruciais para a etapa subsequente de validação das propriedades do protocolo, permitindo uma análise precisa do comportamento de cada mensagem e facilitando a identificação de qualquer inconsistência ou problema durante o processamento.

3.3 Validação das Propriedades

A validação das propriedades do protocolo foi facilitada pelo desenvolvimento de *scripts* específicos, motivados pela necessidade de uma forma prática de verificar a consistência e a ordem de processamento das mensagens. Esses *scripts*, implementados em *Python*, utilizaram-se dos *logs* estruturados e dos IDs de mensagens gerados pela classe *RequestHandler*, conforme mencionado anteriormente.

A partir dos logs, os scripts coletaram os IDs de todas as mensagens processadas por cada nó, respeitando a ordem exata em que foram processadas. Com esses dados, cada

grupo foi comparado com todos os outros grupos, e as mensagens com IDs comuns foram filtradas para análise. Esse processo permitiu garantir que a ordem de processamento de mensagens em comum fosse equivalente entre os grupos, assegurando a consistência necessária para o correto funcionamento do protocolo.

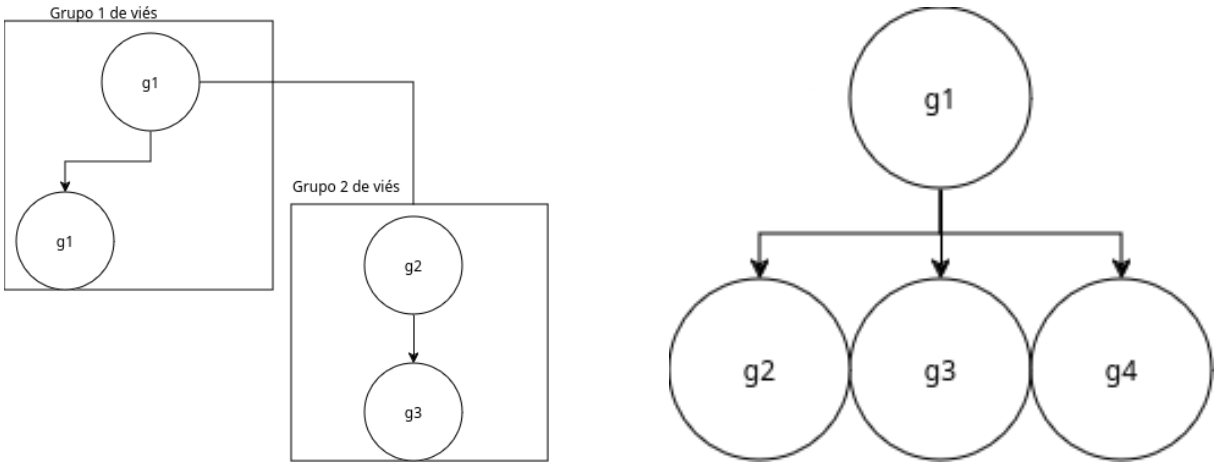
Além da verificação de ordem, os *scripts* também construíram um grafo com os IDs das mensagens. Neste grafo, cada mensagem era mapeada para as mensagens que ocorriam imediatamente depois dela. Essa estrutura permitiu visualizar o encadeamento de mensagens por todo o sistema. Observa-se que uma mensagem X pode ter mais de uma mensagem imediatamente posterior, pois X pode ser entregue a múltiplos grupos, e, nesses grupos, a mensagem seguinte pode diferir, já que ela não precisa ser comum a ambos.

Nesse contexto, o *script* utilizou o grafo para assegurar que não houvesse ciclos entre as mensagens no sistema. Um ciclo poderia ser exemplificado como segue: no grupo 1, as mensagens seguem a ordem $A \rightarrow B \rightarrow C$; no grupo 2, a ordem é $C \rightarrow D$; e no grupo 3, $D \rightarrow B$. Embora a comparação de pares de grupos mantenha a ordem equivalente das mensagens em comum, ao analisar o sistema como um todo, surge um ciclo na sequência $D \rightarrow B \rightarrow C \rightarrow D$.

Assim, esses *scripts* de validação foram cruciais para assegurar que as propriedades de consistência e ordem atômica do *multicast* bizantino fossem preservadas em todas as execuções do protocolo, fornecendo uma camada adicional de verificação prática ao sistema.

4 Implantação e avaliação de desempenho

Após a implementação da nova versão do *ByzCast*, tornou-se necessário executar ambos os protocolos — o original e a nova versão — em condições similares, de modo a garantir que o desempenho de cada um pudesse ser analisado em bases equivalentes. As topologias utilizadas em ambos os casos estão ilustradas nas figuras 2 e 3, que mostram as configurações de 2 e 3 níveis para cada implementação. Os testes foram realizados com um número-alvo de clientes, os quais foram ativados gradualmente com uma pausa média de 300 ms entre cada ativação, até que todos estivessem em execução. Uma vez que o número-alvo de clientes foi atingido, todos eles rodaram continuamente por um período de 2 minutos, enviando requisições em um laço constante até o término do teste. Essa configuração foi projetada para simular uma carga contínua no sistema e observar o comportamento de cada protocolo sob intensidades de uso variadas. Para a nova implementação em 3 níveis, as requisições possuíam um viés, gerando localidade nas mensagens, como visível na figura 5a. 40% das requisições foram destinadas ao grupo 1 de viés, 40% ao grupo 2 de viés e o restante consistia de uma seleção aleatória de pares entre os 4 grupos. Para o caso de 2 níveis, a topologia era sem viés como visto na figura 5b, dado que qualquer requisição para 2 grupos teria um LCA igual.



(a) Topologia de 3 níveis com viés. (b) Topologia de 2 níveis sem viés.

Figura 5 – Topologias da nova implementação usadas para testes.
Fonte: autoral.

4.1 Automação da Implantação

Os testes foram conduzidos no ambiente de experimentação CloudLab (The University of Utah, 2024), que oferece recursos para a criação e gerenciamento de ambientes de rede de forma escalável e controlada. Para a implantação dos protocolos e a execu-

ção dos experimentos, utilizou-se o *Ansible* (Red Hat, Inc., 2024), uma ferramenta de automação poderosa que facilita a configuração e o gerenciamento de sistemas distribuídos, permitindo a coordenação de ambientes de maneira reprodutível e sem intervenção manual.

A versão original do *ByzCast* foi configurada com base no projeto de Oliveira (2023), que fornece um conjunto de *playbooks Ansible* para automação da configuração e execução do protocolo. Esse projeto serviu como base para as configurações e garantiu que o protocolo original fosse implantado em conformidade com as especificações originais. Para a nova implementação, foi desenvolvido um projeto *Ansible* ad-hoc inspirado no trabalho de Oliveira, com ajustes e personalizações específicos para acomodar as particularidades da nova versão do *ByzCast*. Essa automação adicional permitiu que a nova versão fosse implantada de forma otimizada, mantendo as mesmas condições de teste e execução da versão original.

Além disso, foram criados *scripts* personalizados em *Python* para apoiar a automação dessa implementação. Esses *scripts* foram empregados para realizar tarefas fundamentais, como iniciar os nós do protocolo e gerar automaticamente uma pasta de distribuição contendo todos os arquivos necessários para a execução do sistema. Para simplificar ainda mais o processo de configuração, foi desenvolvido um *plugin* de inventário personalizado em *Ansible*. Esse *plugin* permite que as propriedades da topologia sejam definidas diretamente em um arquivo *Python*, reduzindo a necessidade de interagir manualmente com o arquivo de inventário padrão do *Ansible*. Essa abordagem aprimorou a flexibilidade e a reprodutibilidade dos experimentos, facilitando a configuração de diferentes topologias e cenários de teste com mínima intervenção manual.

4.2 Coleta de Estatísticas e Análise de Performance

Durante a execução dos experimentos, cada cliente foi configurado para registrar estatísticas detalhadas sobre as requisições enviadas, incluindo informações sobre latência e o tempo total de processamento das mensagens. A latência, definida como o intervalo de tempo entre o envio de uma mensagem pelo cliente e o recebimento de sua respectiva resposta, foi calculada exclusivamente do lado do cliente. Esses dados foram coletados de maneira estruturada, permitindo que cada cliente registrasse os tempos de envio, processamento e retorno das mensagens recebidas pelo protocolo. As estatísticas de latência foram essenciais para avaliar o comportamento de ambos os protocolos sob as mesmas condições de carga e topologia.

Os dados de latência e de envios de mensagens registrados por cada cliente foram posteriormente utilizados em conjunto com *scripts* de análise de performance, que processaram as informações coletadas para identificar métricas chave de desempenho. Esses

scripts foram projetados para analisar a eficiência de cada versão do *ByzCast* em diferentes condições de operação, oferecendo uma visão aprofundada sobre o comportamento do sistema em resposta à variação de carga e à estrutura da topologia. Esse processo de coleta e análise de dados proporcionou uma base sólida para a avaliação de ambos os protocolos, permitindo que os resultados fossem posteriormente analisados de forma objetiva.

4.3 Resultados

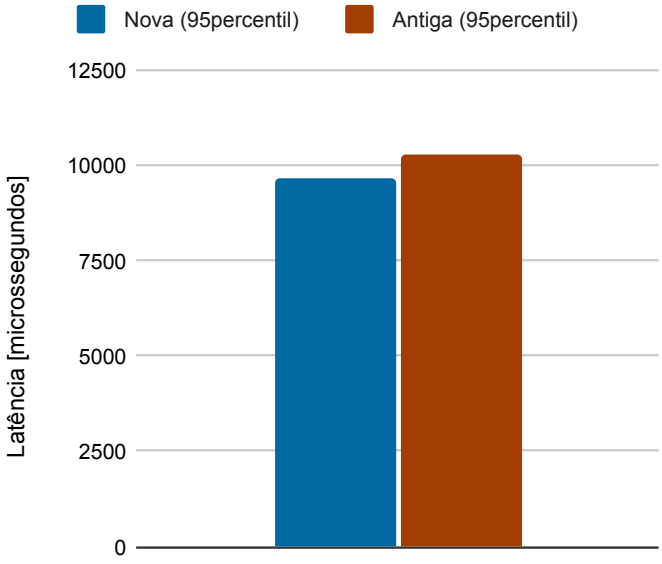


Figura 6 – Latência topologias de 2 níveis com 10 clientes.
Fonte: autoral.

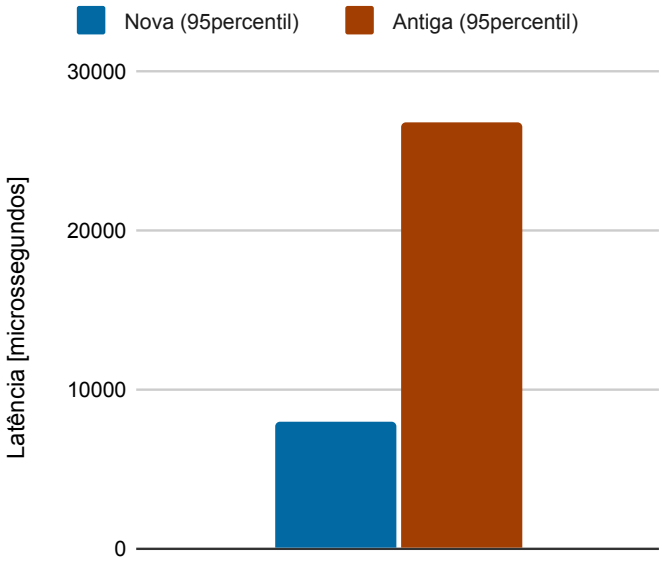


Figura 7 – Latência topologias de 3 níveis com 10 clientes.
Fonte: autoral.

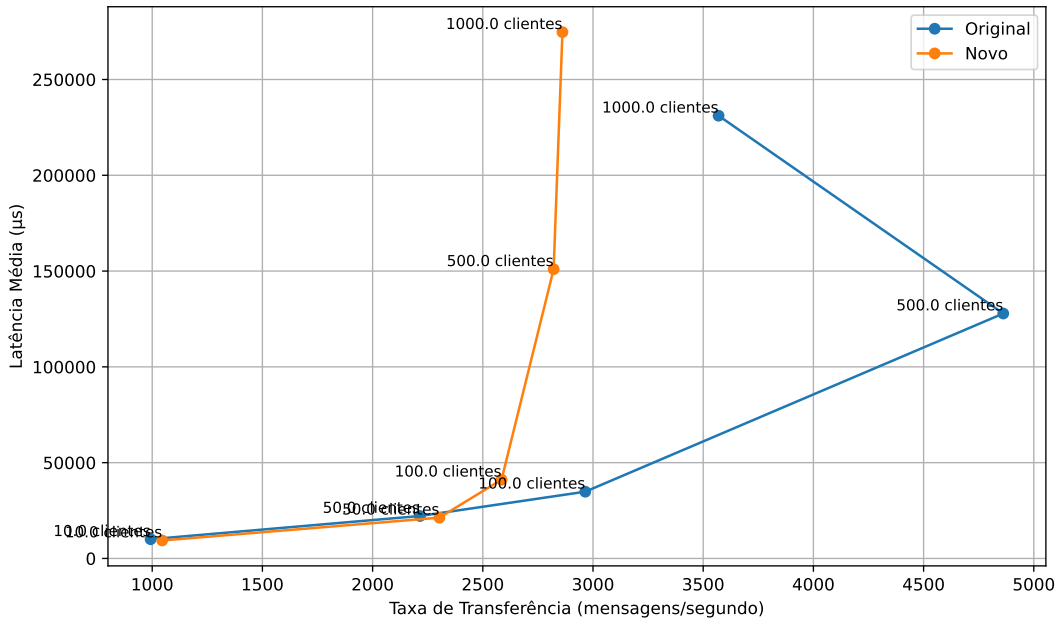


Figura 8 – Latência vs taxa de transferência ambas implementações 2 níveis.
Fonte: autoral.

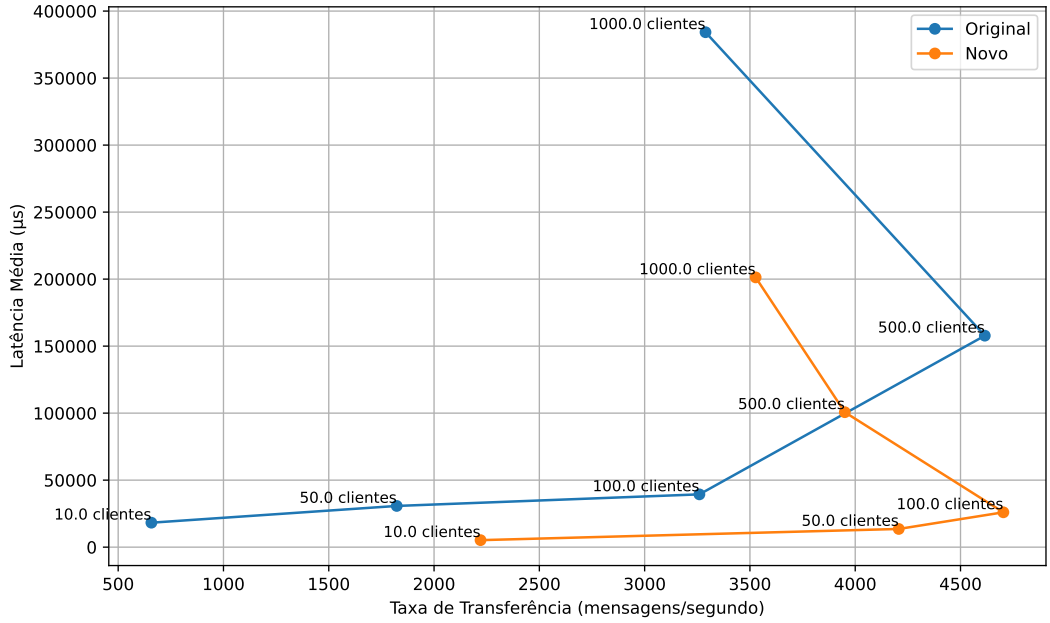


Figura 9 – Latência vs taxa de transferência ambas implementações 3 níveis.
Fonte: autoral.

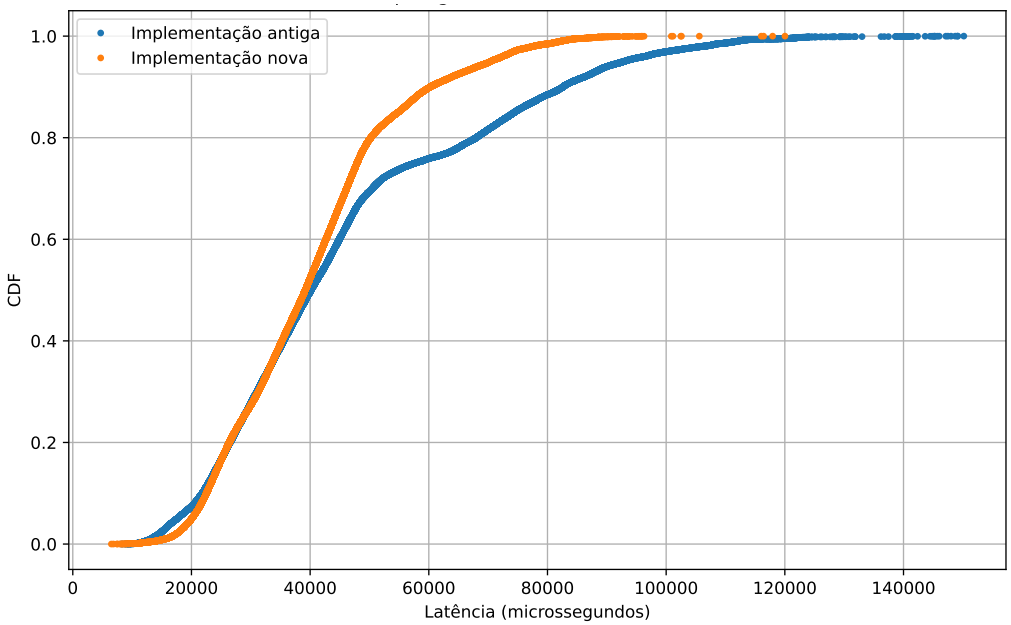


Figura 10 – CDF - Topologias 2 níveis.
Fonte: autoral.

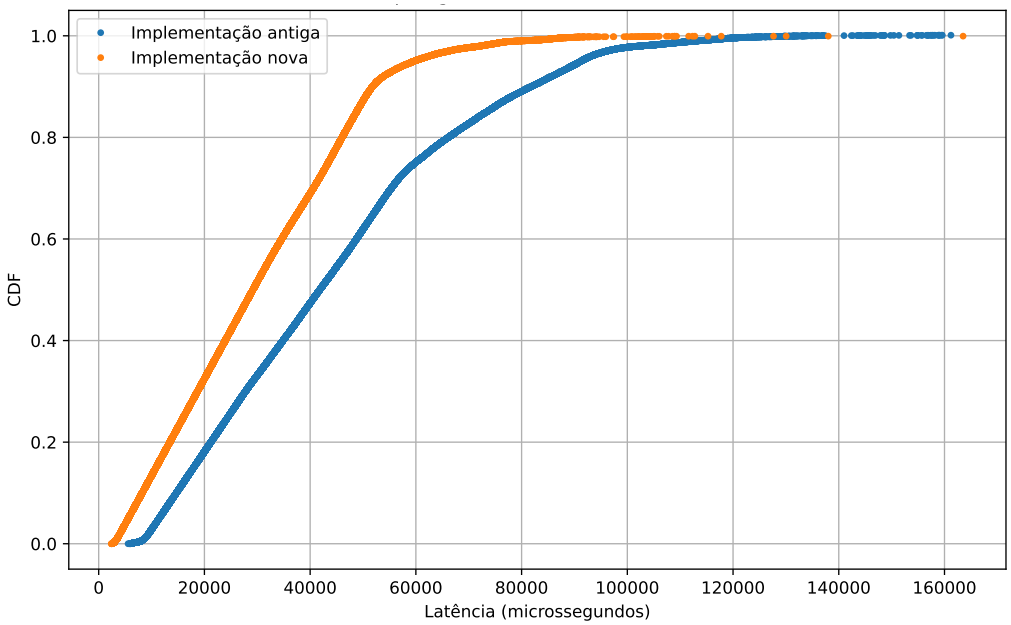


Figura 11 – CDF - Topologias 3 níveis.
Fonte: autoral.

A sigla CDF, utilizada nos gráficos, refere-se à *Cumulative Distribution Function* (Função de Distribuição Acumulada). Essa função representa a probabilidade acumulada de uma variável aleatória assumir valores menores ou iguais a um determinado valor. No contexto dos gráficos apresentados, a CDF é usada para analisar a distribuição da latência observada nos testes, facilitando a comparação entre as implementações ao evidenciar como os tempos de resposta se distribuem ao longo dos experimentos.

A figura 6 apresenta a latência nas topologias de dois níveis, mostrando uma equivalência entre a implementação original e a nova, o que era esperado dado que ambas as estruturas compartilham características semelhantes nessa configuração. O comportamento em dois níveis permite um fluxo de comunicação onde as mensagens percorrem apenas um nível de reenvio, resultando em latências comparáveis entre os protocolos, apesar das diferenças estruturais na implementação otimizada.

Já nas topologias de três níveis, como ilustrado na Figura 7, a nova implementação apresentou um desempenho superior em termos de latência. Esse ganho está relacionado à ausência dos grupos auxiliares, característica da reimplementação que reduziu o número de saltos intermediários e, consequentemente, melhorou a eficiência na entrega das mensagens. Adicionalmente, o viés introduzido na carga em três níveis reforçou a localidade das comunicações, criando condições favoráveis para a nova abordagem, que se beneficiou da simplificação na hierarquia de nós.

Em relação à vazão, calculada com base no numero de respostas obtidas pelo tempo para um grupo de clientes, as Figuras 8 e 9 evidenciam uma variabilidade nos resultados de desempenho. Observamos tanto casos de superioridade quanto de inferioridade no desempenho da implementação otimizada em comparação ao protocolo original. Esses resultados parecem estar associados a detalhes específicos da implementação, além de características das diferentes topologias utilizadas. Essa variabilidade indica oportunidades para refinamentos futuros na implementação, visando uma performance mais consistente.

Por fim, a análise das funções de distribuição cumulativa (CDF) para ambas as topologias oferece uma visão agregada do comportamento de latência do sistema em diferentes percentis. Como visto nas Figuras 10 e 11, a distribuição das latências reforça a vantagem da nova implementação em cenários de três níveis, enquanto mantém resultados similares nos casos de dois níveis.

5 Conclusão

O objetivo central deste trabalho foi alcançado, pois foi possível desenvolver uma implementação otimizada do protocolo *ByzCast* que reduz a quantidade de nós necessários e permite uma estrutura mais flexível para a configuração da topologia. Essa nova versão trouxe uma simplificação significativa, eliminando a necessidade de grupos auxiliares e organizando o protocolo de modo a facilitar a replicação e a comunicação entre os grupos de destino.

No que se refere à performance, os testes mostraram resultados variados: em alguns cenários, a nova implementação apresentou desempenho semelhante ao da versão original, em outros obteve uma melhora, enquanto em situações de carga elevada a performance mostrou-se inferior. Esses resultados podem estar relacionados a diferenças nos detalhes de implementação, que afetam a resposta do protocolo em contextos de alta demanda e podem ser refinados em trabalhos futuros.

Com esta implementação, obtivemos uma solução mais acessível em termos de configuração e menos custosa em recursos, atendendo plenamente à proposta de um protocolo mais enxuto.

Durante o desenvolvimento deste trabalho, algumas dificuldades foram enfrentadas, desde o entendimento da implementação original do *ByzCast* até a adaptação da lógica para garantir a robustez na nova versão otimizada do protocolo. O código-fonte original, escrito em Java 8, apresentou uma complexidade inicial considerável, com uma estrutura rígida e fortemente acoplada à lógica de *atomic multicast*. Essa rigidez exigiu um estudo aprofundado da lógica original para que a nova implementação pudesse manter a integridade das funções essenciais ao mesmo tempo que se adaptava à nova estrutura simplificada e sem a dependência de grupos auxiliares.

Outro desafio significativo foi a implementação do mecanismo de confirmações $N - F$ para requisições em lote, uma tarefa complexa devido às diferentes interpretações das mensagens entre o *ByzCast* e o *BFT-SMaRt*. No *ByzCast*, as mensagens em lote são vistas como múltiplas mensagens individuais, cada uma necessitando de uma confirmação independente. Já no *BFT-SMaRt*, uma mensagem em lote pode ser tratada como uma única unidade, mesmo que contenha várias mensagens do ponto de vista do *ByzCast*. Esse descompasso entre as duas abordagens exigiu o desenvolvimento de uma lógica específica para separar e gerenciar cada mensagem individual no lote, garantindo que todas as confirmações necessárias fossem obtidas antes de se considerar o processamento como concluído. Esse tratamento diferenciado tornou-se crucial para a integridade do protocolo, assegurando que a ordem e a consistência das mensagens fossem preservadas em toda a

comunicação entre os grupos de destino.

Por fim, os testes no *CloudLab* apresentaram um desafio logístico e de automação. Embora a plataforma tenha atendido aos requisitos, a necessidade de uma infraestrutura externa para testar o protocolo em sua escala total exigiu um conjunto abrangente de ferramentas de automação para garantir a consistência dos testes e a reprodutibilidade dos experimentos. Essa automação tornou-se essencial, tanto para configurar corretamente os nós na topologia desejada quanto para coordenar a execução dos testes em uma plataforma externa.

Essas dificuldades foram superadas ao longo do desenvolvimento, proporcionando uma base sólida para a reimplementação do *ByzCast* e confirmando a eficácia da abordagem otimizada proposta.

Para trabalhos futuros, vislumbram-se algumas possibilidades de melhoria e evolução do protocolo. Uma das principais propostas é a adaptação dinâmica da topologia conforme a carga medida em tempo real. Essa abordagem permitiria que o protocolo ajustasse automaticamente a organização dos grupos e a distribuição das mensagens, respondendo de forma mais eficiente a variações na demanda e otimizando a utilização dos nós disponíveis.

Outro possível direcionamento é uma investigação mais aprofundada da implementação, com o objetivo de alinhar de forma mais consistente a performance da versão otimizada aos níveis do protocolo original, ou mesmo superá-los. Considerando os resultados observados nos testes, onde a nova implementação apresentou desempenhos variáveis, essa análise permitiria identificar e ajustar aspectos específicos que possam estar contribuindo para as diferenças de performance, proporcionando ao *ByzCast* otimizado uma execução ainda mais robusta e eficiente.

Por fim, o código-fonte desenvolvido como parte deste trabalho está disponível publicamente em um repositório online, permitindo que outros pesquisadores possam replicar os resultados obtidos ou utilizá-lo como base para futuras investigações. O repositório está documentado de forma a facilitar o entendimento da implementação, e pode ser acessado através do link associado à entrada bibliográfica Silva (2024).

Referências

BESSANI, A.; SOUSA, J.; ALCHIERI, E. E. State machine replication for the masses with bft-smart. In: IEEE. **2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks**. [S.l.], 2014. p. 355–362. Citado 3 vezes nas páginas 9, 18 e 23.

CASTRO, M.; LISKOV, B. et al. Practical byzantine fault tolerance. In: **OsDI**. [S.l.: s.n.], 1999. v. 99, n. 1999, p. 173–186. Citado na página 13.

COELHO, P.; JUNIOR, T. C.; BESSANI, A.; DOTTI, F.; PEDONE, F. Byzantine fault-tolerant atomic multicast. In: IEEE. **2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)**. [S.l.], 2018. p. 39–50. Citado 6 vezes nas páginas 9, 13, 15, 17, 18 e 20.

COULOURIS, G.; DOLLIMORE, J.; KINDBERG, T.; BLAIR, G. **Distributed Systems: Concepts and Design**. 5th. ed. [S.l.]: Addison-Wesley, 2012. Citado 2 vezes nas páginas 8 e 14.

DÉFAGO, X.; URBAN, P.; SCHIPER, A. Total order broadcast and multicast algorithms: Taxonomy and survey. 2003. Citado na página 16.

LAMPORT, L. Paxos made simple. **ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121, December 2001)**, p. 51–58, 2001. Citado na página 13.

LAMPORT, L.; SHOSTAK, R.; PEASE, M. The byzantine generals problem. **ACM Transactions on Programming Languages and Systems (TOPLAS)**, ACM, v. 4, n. 3, p. 382–401, 1982. Citado 2 vezes nas páginas 8 e 13.

LYNCH, N. **Distributed Algorithms**. [S.l.]: Morgan Kaufmann Publishers, 1996. Citado 2 vezes nas páginas 12 e 14.

OLIVEIRA, J. F. Uma proposta de automatização da implantação em nuvem de um protocolo multicast tolerante a falhas bizantinas. Universidade Federal de Uberlândia, 2023. Citado 2 vezes nas páginas 9 e 30.

Red Hat, Inc. **Ansible Automation Platform**. 2024. [Acessado em: 31/10/2024]. Disponível em: <https://www.ansible.com>. Citado na página 30.

RODRIGUES, L.; RAYNAL, M. Atomic broadcast in asynchronous crash-recovery distributed systems. In: IEEE. **Proceedings 20th IEEE international conference on distributed computing systems**. [S.l.], 2000. p. 288–295. Citado na página 15.

SCHNEIDER, F. B. Implementing fault-tolerant services using the state machine approach: A tutorial. **ACM Computing Surveys (CSUR)**, ACM New York, NY, USA, v. 22, n. 4, p. 299–319, 1990. Citado na página 14.

SILVA, D. A. **Protocolo ByzCast**. [S.l.]: GitHub, 2024. <https://github.com/agstrc/tcc-byzcast>. Citado na página 36.

TANENBAUM, A. S.; STEEN, M. V. **Distributed systems**. [S.l.]: CreateSpace Independent Publishing Platform, 2017. Citado na página 8.

The University of Utah. **CloudLab**. 2024. [Acessado em: 31/10/2024]. Disponível em: <https://www.cloudlab.us>. Citado na página 29.