ESCOLA POLITÉCNICA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO
DOUTORADO EM CIÊNCIA DA COMPUTAÇÃO

TARCISIO CEOLIN JUNIOR

CONTRIBUIÇÕES PARA ESCALABILIDADE EM REPLICAÇÃO MÁQUINA DE ESTADOS

Porto Alegre
2023

PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO GRANDE DO SUL
ESCOLA POLITÉCNICA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

CONTRIBUIÇÕES PARA ESCALABILIDADE EM REPLICAÇÃO MÁQUINA DE ESTADOS

TARCISIO CEOLIN JUNIOR

Tese apresentada como requisito parcial à obtenção do grau de Doutor em Ciência da Computação na Pontifícia Universidade Católica do Rio Grande do Sul.

Orientador: Prof. Fernando Luís Dotti

Porto Alegre
2023

Ficha Catalográfica

C398c Ceolin Junior, Tarcisio

Contribuições para escalabilidade em replicação máquina de estados / Tarcisio Ceolin Junior. – 2023.

85 f.

Tese (Doutorado) – Programa de Pós-Graduação em Ciência da Computação, PUCRS.

Orientador: Prof. Dr. Fernando Luís Dotti.

1. Replicação máquina de estados. 2. Escalabilidade. 3. Execução paralela. 4. Multicast atômico. I. Dotti, Fernando Luís. II. Título.

Elaborada pelo Sistema de Geração Automática de Ficha Catalográfica da PUCRS com os dados fornecidos pelo(a) autor(a).

Bibliotecária responsável: Clarissa Jesinska Selbach CRB-10/2051

TARCISIO CEOLIN JUNIOR

CONTRIBUIÇÕES PARA ESCALABILIDADE EM REPLICAÇÃO MÁQUINA DE ESTADOS

Tese apresentada como requisito parcial para obtenção do grau de Doutor em Ciência da Computação do Programa de Pós-Graduação em Ciência da Computação, Escola Politécnica da Pontifícia Universidade Católica do Rio Grande do Sul.

Aprovado(a) em 28 de Março de 2023.

BANCA EXAMINADORA:

Prof. Dr. Odorico Machado Mendizabal (INE/UFSC)

Prof. Dr. Rogério Corrêa Turchetti (PPGEPT/UFSM)

Prof. Dr. Luís Gustavo Leão Fernandes (PPGCC/PUCRS)

Prof. Fernando Luís Dotti (PPGCC/PUCRS - Orientador)

DEDICATÓRIA

Sinéia, esta tese é dedicada a você com todo o meu amor.

“A tarefa não é tanto ver aquilo que ninguém viu, mas pensar o que ninguém ainda pensou sobre aquilo que todo mundo vê.”
(Arthur Schopenhauer)

AGRADECIMENTOS

Esta tese é o resultado de uma longa jornada que envolveu superar diversos desafios. Concluir este trabalho significa para mim mais do que um marco acadêmico, é a realização de um sonho de longa data que só foi possível graças a todos que me apoiaram durante esse processo.

Em especial, gostaria de agradecer ao meu orientador, Prof. Fernando Dotti, por sua inestimável orientação e paciência. Sob sua orientação aprendi lições valiosas sobre como conduzir pesquisa de alta qualidade e sobre como lidar com desafios complexos que levarei comigo para toda a vida. Também agradeço ao Prof. Fernando Pedone, sua intuição e conhecimento na área foram fundamentais para que eu pudesse alcançar este marco importante em minha carreira acadêmica.

Gostaria de expressar minha gratidão aos membros da comissão examinadora: Prof. Rogério Turchetti, Prof. Odorico Mendizabal e Prof. Luís Gustavo. Seus comentários e sugestões foram fundamentais para moldar o resultado final deste trabalho.

Gostaria de reconhecer a assistência inestimável de meus colegas, que me forneceram feedback valioso e suporte ao longo deste trabalho: Daniel Cason, Daniele Sciascia, Eliã Batista, Enrique Fynn, Long Le, Ruda Moura, Renan Louzada, Paulo Coelho, Mojtaba Kelorazi, Leandro Pacheco, Pietro Bressana e Theo Jepsen. Seu encorajamento e disposição em compartilhar sua experiência foram cruciais para o meu sucesso.

Obrigado Marcos Ferreira, pela sua orientação médica, paciência e compreensão, que foram essenciais para meu equilíbrio emocional e físico durante esse período intenso de estudos.

Agradeço também a todos funcionários da PUCRS que tornaram minha vida menos estressante. Em especial, gostaria de agradecer ao Secretário do PPGCC, Régis Escobal.

Serei eternamente grato pela oportunidade que a UFSM possibilitou para o meu aperfeiçoamento profissional. Obrigado Profª. Martha Adaime e Prof. Félix Soares por acreditarem no meu potencial. Também agradeço aos meus queridos colegas de trabalho: Andréia Melo, Ângela Oliveira, Márcia Segabinazzi, Michael Rossato, Fabrícia Iansen, Gerson Lima, Harrison Yago e Simone Marion.

Quero agradecer a minha família pelo amor e apoio inabaláveis. Minha irmã Simone, sua crença em mim me manteve em movimento durante os momentos difíceis e

tornou esta jornada ainda mais significativa. Osmar, seu incentivo constante foi inestimável para mim. Vocês dois são meus exemplos de inspiração.

A minha esposa Sinéia, por estar sempre presente e me apoiar em todas as etapas dessa caminhada, seu amor e carinho foram fundamentais para que eu pudesse concluir este trabalho. Sem você, nada disso seria possível.

Também gostaria de agradecer às instituições que forneceram apoio para essa pesquisa, incluindo a HP sob concessão HP-PROFACC e a *Swiss Federal Commission for Scholarships for Foreign Students* sob concessão do *Swiss Government Excellence Scholarship*.

A todos que contribuíram para esta tese de alguma forma, obrigado do fundo do meu coração. Sua ajuda, incentivo e apoio foram fundamentais para a minha conquista.

CONTRIBUIÇÕES PARA ESCALABILIDADE EM REPLICAÇÃO MÁQUINA DE ESTADOS

RESUMO

O uso crescente de serviços online tem gerado a necessidade de arquiteturas que ofereçam alta disponibilidade e desempenho. No contexto de alta disponibilidade, a técnica de Replicação Máquina de Estados (RME) é uma solução amplamente utilizada em diversos setores da computação, como computação em nuvem, sistemas de banco de dados, mecanismos de sincronização e comunicação confiável. O conceito de RME é simples: como todas as réplicas iniciam com um mesmo estado e executam comandos deterministicamente na mesma ordem, as mesmas mudanças de estado após a execução de cada comando são realizadas entre todas as réplicas do sistema, garantindo consistência forte. No entanto, esse mesmo modelo básico de funcionamento da RME limita o seu desempenho.

Para aumentar a escalabilidade da técnica de RME foram propostas diferentes estratégias. Uma possível estratégia para obter ganhos com a técnica é através do melhor aproveitamento dos múltiplos núcleos de processamento comumente disponíveis em servidores modernos. Diferentes arquiteturas paralelas de RME foram formuladas com soluções que introduzem concorrência na ordenação e execução de comandos, considerando que requisições não conflitantes podem ser processadas em paralelo sem afetar a consistência forte. Protocolos de consenso generalizado trabalham com a mesma noção de conflito: comandos que não conflitam não necessitam de ordenação durante o consenso.

Nesta primeira parte do trabalho propõe-se, o uso de informações de conflito provenientes do consenso generalizado para a subsequente execução paralela de comandos na RME. Esta proposta, ainda não encontrada na literatura, foi descrita, implementada e avaliada, mostrando ganhos de desempenho.

Outra estratégia utilizada para aumentar a escalabilidade da técnica de RME é com o particionamento do estado, permitindo que partições trabalhem de forma independentes. Além da ordenação de comandos para cada partição, comandos multi-partição necessitam de ordenação entre as partições envolvidas. Neste contexto o multicast atômico é uma abstração fundamental, pois captura os requisitos de confiabilidade e ordenação. ByzCast foi o primeiro protocolo de multicast atômico tolerante a falhas bizantinas. Para reusar soluções de ordenação para o caso bizantino, ByzCast faz uso de uma estrutura em árvore para disseminação de mensagens entre as partições envolvidas, sendo cada partição implementada por um conjunto de réplicas atuando em consenso bizantino.

Na segunda parte deste trabalho estendemos ByzCast de um protocolo de multicast atômico tolerante a falhas bizantinas para uma RME particionada com tolerância a falhas bizantinas. Para tal, definimos e discutimos diferentes estratégias de tratamento da requisição e do retorno da resposta aos clientes da RME.

Palavras-Chave: Replicação máquina de estados, escalabilidade, execução paralela, multicast atômico.

CONTRIBUTIONS TO SCALABLE STATE MACHINE REPLICATION

ABSTRACT

The increasing use of online services has created the need for architectures that offer high availability and performance. In the context of high availability, the technique of State Machine Replication (SMR) is one of the most widely used solutions in diferent areas such as cloud computing, database systems, synchronization mechanisms, and reliable communication. The concept of SMR is simple: since all replicas start with the same state and execute commands deterministically in the same order, the same changes in the state after the execution of each command are applied across all replicas of the system, ensuring strong consistency. However, this functioning model of SMR limits its performance.

To increase the scalability of the SMR technique, different strategies have been proposed. One possible strategy to gain benefits from the technique is through better utilization of processor cores commonly available in modern servers. Different parallel architectures for SMR have been formulated with solutions that introduce concurrency in the ordering and execution of commands, considering that non-conflicting requests can be processed in parallel without affecting strong consistency. Generalized consensus protocols work with the same notion of conflict: non-conflicting commands do not require ordering during consensus.

In the first part of this thesis, we propose the use of conflict information from generalized consensus protocols to enable parallel execution of commands in SMR. This proposal, not yet found in the literature, has been described, implemented, and evaluated, showing performance gains.

Another strategy used to increase the scalability of the SMR technique is through state partitioning, allowing partitions to work independently. In addition to command ordering for each partition, multi-partition commands require ordering between the involved partitions. In this context, atomic multicast is a fundamental abstraction as it captures reliability and ordering requirements. ByzCast is the first atomic multicast protocol tolerant

to Byzantine faults. To reuse ordering solutions for the Byzantine case, ByzCast uses a overlay tree for message dissemination between the involved partitions, with each partition implemented by a set of replicas operating under Byzantine consensus.

In the second part of this thesis, we extend ByzCast from a Byzantine fault-tolerant atomic multicast protocol to a partitioned SMR with Byzantine fault tolerance. To do so, we define and discuss different strategies for handling request and response return to SMR clients.

Keywords: State machine replication, scalability, parallel execution, atomic multicast.

LISTA DE FIGURAS

LISTA DE ALGORITMOS

LISTA DE SIGLAS

CFC – Componentes Fortemente Conexos

FIFO – *First In, First Out*

GB – *Gigabyte*

GHz – *Gigahertz*

GAD – Grafo acíclico dirigido

LAN – *Local Area Network*

LCA – *Lowest Common Ancestor*

MAC – Menor Ancestral Comum

RME – Replicação Máquina de Estados

SCC – *Strongly Connected Component*

SMR – *State Machine Replication*

SUMÁRIO

1. INTRODUÇÃO

Desde o surgimento da Internet, o uso de serviços *online* tem crescido rapidamente, exigindo que muitos desses serviços sejam projetados para oferecer tanto alta disponibilidade quanto alto desempenho. Uma forma de prover alta disponibilidade é por meio de replicação, e a abordagem de Replicação Máquina de Estados (RME) de Lamport [40] é frequentemente utilizada para esse fim. O conceito de RME é amplamente utilizado em várias áreas da computação, incluindo (i) computação em nuvem (Amazon Web Services, Google Cloud e Microsoft Azure); (ii) sistemas de banco de dados (Apache Cassandra [39], Amazon DynamoDB [20], Microsoft Azure Cosmos DB, Oracle NoSQL); (iii) mecanismos de sincronização (Apache ZooKeeper [33] e Google Spanner [18]); e (iv) comunicação confiável de alto desempenho (Paxos [42, 43], Raft [56] e Zab [34]); entre outros [4, 54, 30].

O conceito de RME, introduzido por Lamport [40] e Schneider [60], é simples. Supondo que (i) as solicitações de um serviço são processadas de forma determinística; (ii) as réplicas do serviço iniciam no mesmo estado; e (iii) as réplicas recebem todas as solicitações na mesma ordem, então todas as réplicas percorrem a mesma sequência de estados [41]. Assim, caso uma réplica falhe a qualquer momento, outras réplicas estarão imediatamente disponíveis para continuar o processamento. Como todas as réplicas evoluem pela mesma sequência de estados, sistemas que se utilizam da técnica de RME se comportam como uma cópia única do serviço, também chamada consistência forte. Dessa forma, RME é apresentada como uma técnica consolidada capaz de oferecer tolerância a falhas sem comprometer a linearização.

No entanto, independentemente do número de réplicas que o serviço possui, o desempenho do sistema permanece o mesmo devido ao modelo básico de funcionamento de uma RME, onde cada réplica executa sequencialmente a mesma lista de comandos. Devido à necessidade da utilização da técnica RME e do seu modelo baseado no processamento sequencial de solicitações para garantir a consistência forte, a questão geral de como aumentar o desempenho de uma RME tem sido estudada na literatura nos últimos anos [44, 45, 57, 36, 50].

Diferentes estratégias foram propostas tendo como objetivo o aumento da escalabilidade de uma RME. Essa tese delimita o seu escopo em duas categorias: (i) soluções que introduzem concorrência na ordenação e execução de comandos; e (ii) soluções que particionam o estado entre múltiplos grupos de RME.

(i) Para relaxar o processamento sequencial de requisições, permitindo assim o processamento concorrente, observamos que a ordem total deve ser mantida para requisições conflitantes ou dependentes. Resumidamente, duas requisições conflitam se houver uma intersecção entre o conjunto de leitura ou escrita de uma com o conjunto

de escrita de outra e com isso, requisições não conflitantes podem ser processadas em paralelo sem afetar consistência forte. Ou seja, após o processamento do mesmo conjunto de comandos, sejam eles conflitantes entre si ou não, duas réplicas convergem necessariamente para um único estado, mesmo que as cargas de trabalho sejam independentes. Essa observação ajudou a formular diferentes arquiteturas de RME paralelas. Assim, várias arquiteturas de RME surgem na literatura que avalia se as requisições são dependentes e, em caso de independência, permite o processamento concorrente das requisições [38, 36, 37].

Como RME é baseado na ordem total das requisições, é necessário usar um protocolo de consenso, ou algum protocolo equivalente, para essa funcionalidade de entrega de mensagens nas réplicas. Embora a investigação das arquiteturas de escalonamento de RME já tenha algumas contribuições na literatura, pouco foi investigado sobre protocolos de consenso que se concentrem no subsequente processamento concorrente de requisições não conflitantes.

O protocolo de consenso estabelece uma ordem total para as requisições, mas para fins de processamento concorrente, essa ordem total é quebrada em uma ordem parcial que ordena apenas requisições conflitantes. Assim, é natural questionar um protocolo de consenso que gerencia diretamente uma ordem parcial e não uma ordem total entre comandos. Embora existam protocolos de consenso que geram ordens parciais, eles têm outro motivador no seu *design*, como [52], ou não geram informações suficientes para o processamento concorrente de acordo com a ordem parcial [57, 44].

(ii) RME particionada é uma técnica que estende a técnica de RME ao adicionar a noção de particionamento do estado. Em ambas as abordagens, clientes submetem solicitações para as réplicas, que executam as solicitações sequencialmente em uma ordem consistente. Na técnica clássica de RME, todas solicitações envolvem todas as réplicas do sistema, que armazenam o mesmo estado da aplicação. Entretanto, em uma RME particionada, o estado da aplicação é dividido em partições, e cada solicitação acessa uma ou mais partições. Dessa forma, clientes devem propagar solicitações para as partições que sejam relevantes a sua solicitação.

Para lidar com a complexidade de ordenação de solicitações, uma RME particionada pode utilizar uma abstração de comunicação chamada de multicast atômico. O multicast atômico fornece os meios para que solicitações sejam propagadas de forma confiável e consistente para um ou mais grupos de réplicas, onde cada grupo de réplicas implementa uma partição.

Embora exista uma extensa literatura sobre multicast atômico [21], os protocolos existentes foram concebidos para tratar exclusivamente de falhas consideradas benignas. Nesse contexto, ByzCast [15] surge a partir de um trabalho colaborativo. ByzCast é um protocolo de multicast atômico tolerante a falhas bizantinas que adota uma estrutura de topologia em árvore para organizar seus grupos. Uma mensagem m é encaminhada

pelo menor ancestral comum dos grupos de destino dessa mensagem. Quando um grupo intermedíario recebe *m*, ele a ordena e, em seguida, a propaga para o próximo nível na árvore até que todos os grupos contidos no destino sejam alcançados. ByzCast pode ser caracterizado como parcialmente genuíno, visto que a entrega de uma mensagem destinada a múltiplos grupos pode envolver grupos intermediários que não pertencem ao conjunto de destino.

Por mais que exista uma categoria de protocolos de multicast atômico definidos pela sua topologia em formato de *anel*, até onde se sabe, ByzCast foi a primeira iniciativa a trazer um modelo de execução de um estado particionado disposto em uma árvore de sobreposição. Essa topologia é relevante pois não existem estudos onde a semântica da aplicação é considerada na definição de modelos de execução de uma requisição em uma RME particionada utilizando uma topologia em árvore de sobreposição.

1.1 Objetivos

Nesta tese de doutorado, abordamos algumas formas para aumentar a escalabilidade da técnica de Replicação Máquina de Estados (RME). Nosso objetivo principal é contribuir para o avanço do conhecimento nessa área, fornecendo novas perspectivas de pesquisa através do uso de consenso generalizado e multicast atômico para a construção de sistemas distribuídos escaláveis, confiáveis e eficientes que utilizam a técnica de RME.

Especificamente, buscamos alcançar os seguintes objetivos:

- Investigar e propor estratégias que explorem o uso de protocolos de consenso generalizado para melhorar o desempenho e a escalabilidade da técnica RME. Isso envolve explorar as informações fornecidas pelo consenso generalizado para permitir a execução paralela de comandos independentes dentro de cada réplica; e
- Estender um protocolo de multicast atômico tolerante a falhas bizantinas considerando o uso da técnica de RME. O objetivo é modelar o comportamento desse protocolo para garantir o tratamento adequado de uma requisição em um estado particionado.

1.2 Contribuições

Este trabalho contribui para o ganho de escalabilidade na utilização da técnica de RME em dois eixos distintos: (i) Verticalmente (§1.2.1), dentro de cada réplica com PePaxos [14], aumentando o paralelismo na execução de comandos independentes; e (ii) Horizontalmente (§1.2.2), com o uso do particionamento do estado com ByzCast [15],

por meio da implementação de um algoritmo assíncrono, sendo posteriormente utilizado como base para a criação e definição de modelos de execução.

1.2.1 Contribuições para RME Paralela

- Nós criamos algoritmos para execução paralela de comandos que utilizam informações de conflito de consenso generalizado;
- Nós implementamos um protótipo usando Egalitarian Paxos (ePaxos) como protocolo de consenso; e
- Nós avaliamos as implicações de desempenho sob vários aspectos, como taxas de conflito, custos de execução de comando e número de núcleos disponíveis.

1.2.2 Contribuições para RME Particionada

- Nós criamos algoritmos para a execução assíncrona de um protocolo de multicast atômico;
- Nós definimos diferentes modelos de execução aplicados a uma árvore de sobreposição no contexto de uma RME particionada; e
- Nós avaliamos tanto vazão quanto latência dos modelos apresentados.

1.3 Organização da Tese

O restante deste trabalho está organizado da seguinte forma. No Capítulo 2, definimos os modelos de sistema considerado e apresentamos algumas definições que são usadas ao longo da tese. Apresentamos no Capítulo 3 os algoritmos utilizados para a execução paralela de comandos independentes, assim como sua implementação e análise de desempenho. No Capítulo 4 apresentamos algoritmos utilizados para o funcionamento assíncrono do ByzCast assim como descrevemos e analisamos os modelos de execução aplicados a sua topologia. Conclusões são apresentadas no Capítulo 6, que sumariza as principais contribuições e identifica áreas para pesquisas futuras.

2. MODELO DE SISTEMA E DEFINIÇÕES

Neste capítulo, apresentamos em detalhes os modelos de sistema utilizados e relembramos algumas noções fundamentais que são empregadas ao longo desta tese. Dado que as contribuições deste trabalho consideram premissas distintas, na Seção 2.1 é apresentado o modelo de sistema utilizado no Capítulo 3. Em seguida, na Seção 2.2, descrevemos o modelo de sistema adotado no Capítulo 4. Posteriormente, na Seção 2.3, detalhamos nosso critério de consistência. Por fim, na Seção 2.4, contextualizamos o que foi apresentado considerando o emprego da técnica RME.

2.1 Modelo de sistema e definições - RME Paralela

2.1.1 Processos e comunicação

Consideramos um sistema distribuído $\Pi = \{p_1, ..., p_n\}$ com processos que se comunicam entre si por meio de troca de mensagens e que não possuem acesso a uma memória compartilhada ou a um relógio global.

Como não é possível a resolução do consenso em um sistema assíncrono [26], adotamos um sistema parcialmente síncrono, conforme definido em [23]. Dessa forma, mensagens podem sofrer atrasos arbitrariamente grandes (mas finitos) além de não existir um limite na velocidade relativa dos processos.

2.1.2 Modelo de falhas por colapso (*Crash-failure model*)

Nesse modelo, processos podem falhar por colapso, mas não experimentam comportamento arbitrário (ou seja, não ocorrem falhas bizantinas). Um processo que nunca falha é considerado correto; caso contrário, é defeituoso. Existem até f réplicas com falhas, de um total de $2f + 1$ réplicas.

Os *links* de comunicação são *fair-lossy*, ou seja, eles não criam, corrompem ou duplicam mensagens e garantem que para qualquer dois processos corretos p e q, e para qualquer mensagem m, se p envia m para q infinitas vezes, então q recebe m um número infinito de vezes.

2.1.3 Consenso

Em um ambiente distribuído composto por múltiplos processos, onde cada processo pode enviar um valor diferente para os seus pares, o uso de um protocolo de consenso é empregado quando se faz necessário assegurar que processos concordem sobre quais valores são entregues e em qual ordem eles devem ser entregues.

Consenso é definido pelas primitivas *propose*(v) e *decide*(v), onde a primeira primitiva é utilizada para propor valores e a segunda é utilizada para decidir sobre um valor proposto. Como estamos interessados em uma sequência de decisões, definimos uma primitiva *decide*(i, v) onde i representa o número da instância do consenso. Esse é um número natural que associa uma ordem crescente e sem lacunas às decisões. No modelo de falhas por colapso adotamos as propriedades definidas pelo consenso uniforme [53, 31]. O consenso uniforme garante que nenhum par de processos decida em valores diferentes, independentemente de estarem corretos ou não. Consenso uniforme satisfaz as seguintes propriedades para cada instância i:

- *Integridade uniforme*: Se um processo decide um valor v, então v foi previamente proposto por algum processo;
- *Acordo uniforme*: Se um processo (seja ele correto ou falho) decide v, então todos os processos corretos eventualmente decidem v;
- *Terminação*: Todo processo correto, em algum momento, decide exatamente um valor.

2.2 Modelo de sistema e definições - RME Particionada

2.2.1 Processos, grupos e comunicação

Estendemos a Seção 2.1.1 ao adicionamos ao modelo de sistema um conjunto $\Gamma = \{g_1, \dots, g_m\}$ como um conjunto de grupos de processos que fazem parte da composição do sistema. Esses grupos são disjuntos, não vazios e satisfazem $\bigcup_{g \in \Gamma} g = \Pi$.

2.2.2 Modelo de falhas Bizantina (*Bizantine-failure model*)

Nesse modelo, processos estão sujeitos a falhas Bizantinas, ou seja, processos que falham podem apresentar qualquer comportamento arbitrário [46]. Processos podem

ser corretos ou falhos. Um processo correto segue sua especificação de funcionamento e um processo falho pode comunicar valores aleatórios para os outros processos. Nesse modelo a falha pode ser resultado de uma operação mal-intencionada gerada por terceiros. Cada grupo possui $3f+1$ processos onde f representa o número máximo de processos falhos por grupo [13].

Técnicas criptográficas para autenticação e cifragem de conteúdo são utilizadas. Assumimos que adversários que detenham controle sobre processos bizantinos possuem capacidades computacionais limitadas e, portanto, são incapazes de romper as técnicas criptográficas empregadas. Adversários podem atrasar os processos corretos a partir da coordenação de processos bizantinos para causar o maior dano possível ao sistema, mas não podem, no entanto, atrasar processos corretos de forma indeterminada.

2.2.3 Multicast Atômico

Para cada mensagem m, $m.dst$ define os grupos na qual m é multicast. Dizemos que uma mensagem m é *local* quando $|m.dst| = 1$ e *global* quando $|m.dst| > 1$.

Um processo realiza um multicast atômico de uma mensagem m invocando a primitiva *a-multicast(m)* e entrega m através de *a-deliver(m)*. Definimos a relação $<$ no conjunto de mensagens entregues por processos corretos da seguinte forma: $m < m'$ se e somente se existe um processo correto que entrega m antes de m'.

Multicast atômico é satisfeito pelas propriedades [31]:

- *Validade*: Se um processo correto p realiza *a-multicast* de uma mensagem m, então em algum momento todos os processos corretos $q \in g$, onde $g \in m.dst$, realizam *a-deliver* de m;
- *Acordo*: Se um processo correto p *a-deliver* m, então em algum momento todos os processos corretos $q \in g$, onde $g \in m.dst$, realizam *a-deliver* de m
- *Integridade*: Para qualquer processo correto p e para qualquer mensagem m, p *a-deliver* m no máximo uma vez, e somente se $p \in g$, $g \in m.dst$, e se m foi previamente *a-multicast*.
- *Ordem de prefixo*: Para quaisquer duas mensagens m e m' e para quaisquer dois processos corretos p e q, tal que $p \in g$, $q \in h$ e $\{g, h\} \subseteq m.dst \cap m'.dst$, se p *a-deliver* m e q *a-deliver* m', então ou p *a-deliver* m' antes de m ou q *a-deliver* m antes de m'.
- *Ordem Acíclica*: A relação $<$ é acíclica.

2.2.4 Difusão atômica FIFO

Difusão atômica é um caso especial de multicast atômico onde toda mensagem m é endereçada para todos grupos do sistema. Assumimos que cada grupo implementa difusão atômica FIFO, que, além das propriedades apresentadas anteriormente, também garante a seguinte propriedade:

- *Ordem FIFO*: Se um processo correto envia uma mensagem m antes de enviar uma mensagem m', nenhum processo correto entrega m' a menos que tenha entregue anteriormente m.

2.2.5 Consenso

As primitivas de consenso que foram definidas anteriormente na Seção 2.1.3 continuam válidas e aplicáveis neste contexto. No entanto, pela caraterística de falha no modelo Bizantino, a presença de processos falhos pode resultar na decisão de valores arbitrários.

Dessa forma, uma vez que as propriedades do consenso uniforme não podem ser adotadas diante das restrições impostas pelo modelo Bizantino, adotamos as seguintes propriedades [31]:

- *Integridade*: Se um processo correto decide v na instância i, então v foi proposto anteriormente por algum processo em i;
- *Terminação*: Se um processo correto no grupo g propõe um valor na instância i, então todo processo correto em g, em algum momento, decide exatamente um único valor para i;
- *Acordo*: Se um processo correto no grupo g decide v na instância i, então nenhum outro processo correto em g decide $y \neq x$ em i.

2.3 Consistência

Nosso critério de consistência é *linearizabilidade* e uma noção sobre o conceito foi introduzida por Herlihy e Wing [32]. Linearizabilidade se refere à propriedade de um conjunto de operações em um sistema distribuído ser executado de forma a produzir o mesmo resultado que se as operações fossem executadas em sequência, uma de cada vez, fornecendo garantias tanto de leitura quanto escrita sobre um determinado registro

em determinado momento da execução. Essa abordagem é tradicionalmente utilizada em Replicação Máquina de Estados.

Uma execução é linearizável quando existe uma forma de ordenar totalmente as operações de maneira que: (a) ela respeita a semântica dos objetos acessados pelas operações, conforme expresso em suas definições, e (b) ela respeita a ordenação de tempo real das operações na execução. Existe uma ordem de tempo real entre duas operações se uma operação termina em um cliente antes que outra operação comece em um cliente.

2.4 Replicação Máquina de Estados

A técnica de Replicação Máquina de Estados (RME) torna um serviço tolerante a falhas ao replicar o servidor e coordenar a execução de comandos enviados pelos clientes entre as réplicas do sistema [40, 60]. O serviço é definido por uma máquina de estado e consiste de *variáveis de estado* que codificam o estado da máquina de estados e um conjunto de *comandos* que alteram o estado (ou seja, a entrada). A execução de um comando pode (i) ler variáveis do estado; (ii) modificar as variáveis do estado; e (iii) produzir uma resposta para o comando (ou seja, a saída).

A técnica de RME fornece linearização pela forma em que atua na propagação dos comandos enviados pelos clientes pois: i) toda réplica correta recebe todos comandos; e ii) todas réplicas corretas devem concordar com a ordem dos comandos recebidos e executados.

Para garantir que a execução de um comando resultará nas mesmas mudanças de estado e nos mesmos resultados em diferentes réplicas, os comandos são *determinísticos*: as mudanças no estado e a resposta de um comando são uma função das variáveis de estado lidas pelo comando e do próprio comando. Portanto, se os servidores executarem os comandos na mesma ordem, eles produzirão as mesmas mudanças de estado após a execução de cada comando.

Dessa forma, a técnica de RME necessita que todas as réplicas executem os comandos na mesma ordem. Portanto, antes que os comandos sejam executados pelas réplicas, a ordem de execução deve ser acordada entre as réplicas. Esse acordo é alcançado através da utilização de um protocolo de consenso. Sempre que um comando é enviado para uma réplica que utiliza a técnica de RME, ele é proposto no consenso. A RME é implementada quando réplicas executam comandos enviados de acordo com a ordem do consenso, conforme Algoritmo 2.1.

Como os comandos são executados de forma determinística e sequencialmente, toda réplica do sistema obtém o mesmo resultado para todo comando executado, resultando na mesma modificação de seu estado interno. Dessa forma, como as réplicas

1: **data structures**
2: $i : 0$ *{próxima decisão esperada}*
3: Replica's execution works as follows:
4: **upon** decide(i, c) *{sequência de consenso}*
5: *executeAndReplyToClient*(c)
6: $i \leftarrow i + 1$

Algoritmo 2.1 – Execução de uma réplica RME

produzem o mesmo resultado, qualquer resposta enviada por qualquer uma das réplicas aos clientes é sempre suficientemente boa.

A técnica de RME fornece consistência forte. Clientes trabalham com a ilusão de um serviço não replicado, ou seja, a replicação é transparente. Ao contrário de um serviço não replicado, os clientes permanecem alheios às falhas já que o serviço está operacional mesmo com a presença de falha de algumas de suas réplicas (ou seja, até f réplicas defeituosas).

A capacidade de uma RME em adicionar ou remover réplicas ao sistema possibilita um nível de tolerância à falhas configurável. Ocorre que adicionar uma réplica ao sistema, fornecendo assim maior tolerância a falhas, não se resulta necessariamente em ganho de desempenho para o sistema pois, ao considerarmos que a réplica adicional executa as mesmas operações na mesma sequência das demais réplicas, a performance tende a permanecer a mesma.

Protocolos de comunicação como difusão atômica e multicast atômico são formas de tentar suprir os requisitos crescentes em aplicações *online* que necessitam de alta performance, alta disponibilidade e que ao mesmo tempo sejam escaláveis. Nesse sentido, especificar sistemas que combinem escalabilidade e tolerância a falhas pode ser um desafio. Sistemas que garantem níveis de consistência fraca são sistemas que reduzem as garantias ofertadas. Outros sistemas que garantem consistência forte trazem como benefício uma previsível (e intuitiva) execução do algoritmo, mas requerem que requisições originadas pelos clientes sejam ordenadas entre todos participantes do sistema antes de serem executadas em cada uma das réplicas.

Algumas abordagens propostas na literatura visam contornar as limitações de escalabilidade que afetam sistemas que utilizam a técnica RME. A primeira dessas abordagens, descrita no Capítulo 3, tem como objetivo explorar o uso do paralelismo para executar múltiplas requisições de forma concorrente, sem afetar a consistência do sistema. Essa abordagem é conhecida como RME paralela. A segunda abordagem, descrita no Capítulo 4, consiste no particionamento do estado da RME em múltiplas partições, com o objetivo de distribuir a carga de trabalho do sistema entre partições. Essa estratégia é denominada RME particionada e tem como finalidade melhorar a escalabilidade

do sistema, permitindo que múltiplas partições possam processar as requisições de forma independente.

3. CONTRIBUIÇÕES PARA RME PARALELA

A técnica de Replicação Máquina de Estados (RME) é uma abordagem conceitualmente simples e bastante eficaz para tornar sistemas tolerantes a falhas. A ideia básica é que réplicas de servidores executem solicitações enviadas por clientes de forma determinística, sequencial e na mesma ordem [40, 60]. Como resultado as réplicas passam pela mesma sequência de estados, produzindo uma mesma saída. A técnica permite que desenvolvedores se concentrem nas funcionalidades da sua aplicação, evitando assim as dificuldades ao lidar com falhas de réplicas [26]. Por essas características, essa técnica é utilizada com sucesso em diferentes contextos [10, 28, 18].

Diferentes estratégias foram propostas com o objetivo de aumentar a vazão em RME. Uma grande categoria de soluções introduz a concorrência na ordenação e execução de comandos, aproveitando informações fornecidas pela semântica da aplicação. Mais precisamente, dois comandos entram em conflito ao acessarem o mesmo estado e pelo menos um deles atualizar esse estado; caso contrário, os comandos são independentes. Quando comandos são independentes, sua ordem é irrelevante e decisões podem ser realizadas mais rapidamente quando a sua ordenação e execução.

Protocolos de consenso sem líder (*leaderless*) exploraram a semântica da aplicação para otimizar a ordenação dos comandos. Como conflitos podem surgir quando diferentes réplicas propõem comandos de forma concorrente, protocolos como *Generalized Paxos* [44], *Generic Broadcast* [57] e *Egalitarian Paxos* [52] utilizam informações fornecidas pela semântica da aplicação para determinar a ordem de comandos que conflitam. Chamamos estes de protocolos de consenso generalizado.

Em relação a execução de comandos, várias técnicas foram propostas para superar a limitação da execução sequencial e permitir a execução paralela de comandos que não conflitam. Essa questão é especialmente importante se considerarmos arquiteturas modernas de processadores com múltiplos núcleos. O principal desafio é garantir que o mesmo comportamento determinístico de uma réplica na execução paralela de comandos.

Em EVE [37] e *Storyboard* [35], réplicas executam de forma otimista para posteriormente concordar com o resultado, sendo talvez necessário reexecutar alguns comandos. Em *Rex* [30], uma réplica executa e registra as dependências entre comandos para em seguida enviar para consenso esse rastro de dependências entre comandos de forma que as demais réplicas utilizem o mesmo rastro na sua execução. Em várias abordagens [38, 51, 2, 3, 25], o consenso ordena totalmente os comandos para que réplicas identifiquem conflitos, possibilitando concorrência na execução de comandos independentes.

Embora tanto a ordenação quanto a execução de comandos explorem aspectos comuns, ou seja, a semântica da aplicação, não há estudos considerando a integração das abordagens. Neste capítulo, investigou-se até que ponto a ordem parcial resultante do consenso generalizado pode ser benéfica para a execução paralela de comandos nas réplicas. Nossas contribuições são: (i) desenvolvemos algoritmos para a execução paralela de comandos que utilizam informações de conflito fornecido pelo consenso generalizado; (ii) implementamos um protótipo utilizando *Egalitarian Paxos* (ePaxos) como protocolo de consenso; e (iii) avaliamos as implicações de desempenho sob vários aspectos, como taxas de conflito, custo de execução de comandos e número de núcleos de processador disponíveis.

A avaliação experimental revelou que a abordagem utilizada resultou em ganhos relevantes de desempenho a medida que a independência entre comandos e o custo computacional aumentaram. A abordagem apresentada em relação ao ePaxos possuem desempenho similar quando os níveis de conflito são elevados ou quando a quantidade de núcleos de processador é menor. Além disso, os resultados indicam que não há sobrecarga significativa na detecção de conflitos pela execução paralela.

Esse capítulo é organizado da seguinte forma: A Seção 3.1 apresenta as principais premissas e considerações sobre consenso generalizado. A Seção 3.2 detalha a proposta para a utilização de informações de dependência fornecidas pelo consenso generalizado para a execução paralela de comandos. A Seção 3.3 detalha o protótipo e analisa os resultados. A Seção 3.4 conclui o capítulo com uma discussão do que foi apresentado.

3.1 Consenso Generalizado

O uso de consenso permite que um conjunto de réplicas concordem com uma ordem total de comandos a serem executados. Essa ordem total é necessária para que todas as réplicas do sistema mantenham-se consistentes. Entretanto foi observado que comandos que não conflitam entre si podem ser executados em qualquer ordem, ou seja, a ordem relativa de execução entre os comandos que não conflitam é irrelevante. Com isso o estado é mantido consistente se todas as réplicas concordarem na ordem de comandos que conflitam entre si, permitindo que comandos independentes sejam executados em qualquer ordem. Dessa forma, ao invés de uma ordem total, essa abordagem traz a tona o uso de uma ordem parcial na execução de comandos. Com isso, a execução de comandos que respeitam uma ordem parcial é aceita entre as diferentes réplicas de um mesmo sistema.

O protocolo de consenso *Paxos* [42] é amplamente utilizado em diferentes implementações que utilizam RME. *Paxos* e protocolos derivados do *Paxos* são baseados em um processo distinto ou líder para coordenar o consenso. O uso de um único processo para

essa tarefa pode limitar a performance do sistema, e portanto, alternativas surgiram para permitir que mais de um processo coordene o consenso, também chamado de protocolos de consenso sem líder (*leaderless*[1]).

Fast Paxos de Lamport [45] introduz a possibilidade de um coordenador delegar o direito a outros *proposers* para endereçar diretamente *acceptors*, e aos *acceptors* a aceitarem propostas de outros *proposers*. Isso reduz a latência no processamento das mensagens e tem o potencial de aumentar a vazão do sistema ao evitar que um único ponto seja utilizado para o encaminhamento de todas propostas. A desvantagem desse *design* é a possibilidade de colisões de propostas que podem resultar em *live-lock*.

Generalized Paxos [44] estende Fast paxos para lidar com esse aspecto. Ele generaliza a abordagem de máquina de estados para permitir o acordo em um conjunto parcialmente ordenado de instâncias de consenso. Com consenso generalizado, ao invés da RME acordar em um único comando por vez, construindo uma sequência idêntica entre todas réplicas, no consenso generalizado réplicas concordam com uma classe equivalente de sequência de comandos. Informalmente, uma classe equivalente de sequências de comandos é formada por todas as sequências que podem ser derivadas a partir de uma mesma ordem parcial.

Lamport defende que consenso generalizado seja aplicado em diferentes situações e descreve uma abstração genérica denominada estrutura do comando (*c-struct*), onde, dependendo da forma em que é modelada essa *c-struct*, pode-se definir o comportamento do protocolo de consenso para diferentes tipos de aplicações. É através de uma estrutura *c-struct* que se define como instâncias de consenso conflitam entre si ou não.

A principal motivação para essa generalização é aumentar a vazão no consenso: como qualquer classe equivalente de comandos pode ser aceita, a menor incidência de colisões na execução do protocolo de consenso são naturais. Tal característica permite a utilização de protocolos de consenso *leaderless*, permitindo assim que réplicas executem de forma concorrente diferentes instâncias de consenso.

O caso de interesse é o consenso para histórico de comandos (*command histories*). Um histórico de comandos é uma ordem parcial de comandos que relaciona apenas comandos conflitantes. *Generic broadcast* [57] também entrega uma ordem parcial de comandos, sendo equivalente ao consenso generalizado de Lamport para históricos de comandos [44]. Da mesma forma, a semântica de comandos é modelada com uma relação de conflito e um algoritmo de difusão que funciona com qualquer relação de conflito. A seguir, definimos a noção de conflito.

[1] Na prática, a existência de um protocolo de consenso sem um integrante do sistema responsável pelo *propose* de um comando é desconhecida, mas tal denominação *leaderless* é amplamente utilizada na literatura quando aplicada neste contexto.

Definição 1 (Comandos, conjuntos de leitura e escrita, conflito) *Seja C o conjunto de comandos disponíveis em um serviço, ou seja, todos os comandos que um cliente pode emitir. Um comando pode ser qualquer cálculo determinístico envolvendo objetos que fazem parte do estado da aplicação. Denotamos os conjuntos de objetos da aplicação que as réplicas leem e escrevem ao executar um comando c como* readset *e* writeset *de c, ou $RS(c)$ e $WS(c)$, respectivamente. A relação de conflito $\#_C \subseteq C \times C$ entre comandos é definida como:*

$$(c_i, c_j) \in \#_C \; iff \begin{pmatrix} RS(c_i) \cap WS(c_j) \neq \emptyset \vee \\ WS(c_i) \cap RS(c_j) \neq \emptyset \vee \\ WS(c_i) \cap WS(c_j) \neq \emptyset \end{pmatrix}$$

Comandos c_i e c_j *conflitam* (*dependem* ou *interferem*) se $(c_i, c_j) \in \#_C$. Pares de comandos que não estão em $\#_C$ são *não conflitantes* (*independentes* ou *não interferentes*). Dois comandos que não conflitam são *independentes*.

Como mencionado, estamos interessados no consenso generalizado sobre histórico de comandos, ou seja, quando uma ordem parcial de comandos conflitantes é entregue nas réplicas.

Generalized Paxos (em históricos de comandos) e *Generic Broadcast* entregam sequências de comandos que são compatíveis com a ordem parcial construída a partir de suas dependências. As réplicas RME que se baseiam nesses protocolos se comportariam como descrito no Algoritmo 2.1. Essas réplicas podem observar ordens diferentes mas que são equivalentes, que comutam comandos não conflitantes. Dessa forma, executar comandos de acordo com a ordem observada não comprometerá a linearizabilidade. No entanto, as informações de dependência de comandos permanecem encapsuladas pelo protocolo de consenso.

Moraru [52] propôs *Egalitarian Paxos* com algumas das principais características dos protocolos anteriores ao utilizar a semântica da aplicação para decidir comandos que conflitam (no contexto do consenso) com comandos que não conflitam.

Enquanto estes protocolos utilizam a informação de semântica da aplicação a nível de consenso, seja ela codificada em uma *c-struct* ou em uma relação de conflito, a utilização da informação de independência entre comandos no contexto de execução concorrente em uma RME não havia sido explorada.

3.1.1 Egalitarian Paxos

Egalitarian Paxos (ePaxos) [52] é outro protocolo que utiliza a semântica do comando para evitar colisões durante a etapa de consenso. Como diferentes réplicas coordenam o consenso para comandos diferentes de forma concorrente, comandos que não conflitam podem ser entregues de forma independentes. No entanto, o *ePaxos* é orga-

nizado de forma diferente na interface do serviço, permitindo acesso às informações de dependência do comando coletadas durante o consenso. Embora alguns desses aspectos sejam considerados mais de natureza de engenharia, eles são importantes para a escolha do protótipo de consenso generalizado a ser experimentado.

Visão geral da fase de consenso do ePaxos

No *ePaxos*, um comando é uma instância de consenso. Clientes enviam comandos para as réplicas que podem propor instâncias de forma concorrente. Uma réplica coordena o consenso, atuando como líder de comando, para as instâncias que ela tenha *propose*. O *ePaxos* usa o termo interferência que é equivalente a conflito, definido anteriormente.

Durante a coordenação, a presença ou ausência de conflitos de instâncias pendentes nas demais réplicas é identificada e registrada em um conjunto de conflitos para cada instância. Mais precisamente, isso acontece na Fase 1 quando um líder de comando envia uma instância para outras réplicas, que avaliam a instância proposta em relação às instâncias registradas localmente para identificar um conjunto de instâncias conflitantes.

Uma réplica atuando como líder de comando recebe um conjunto de conflitos como resposta em relação a instância proposta das demais réplicas do sistema. Se todas respostas forem idênticas, possivelmente vazias (ou seja, sem conflito), as réplicas possuem a mesma visão do estado e um *fast path* pode ser realizado. Caso contrário, o líder do comando é responsável por sincronizar o conjunto de conflitos da instância proposta entre todas as réplicas através do *slow path*.

Uma réplica mantém um conjunto de instâncias que progridem durante o consenso, um conjunto I, definido no Algoritmo 3.1, linha 2. As informações de uma instância são compostas por: (i) o comando; (ii) o conjunto de outras instâncias com as quais ela entra em conflito; (iii) um número de sequência a ser usado em caso de resolução de ciclos; (iv) seu estado, que pode ser, em ordem: *Pre-Accepted, Accepted, Committed, and Executed* (conforme linhas 3 a 6). Uma instância está no estado *commited* quando suas informações estão completas e replicadas em todas as réplicas. Do ponto de vista do consenso, uma instância *commited* é uma instância entregue.

Visão geral da fase de execução do ePaxos

Instâncias *commited* são registradas no conjunto de instâncias I e uma ordem parcial de comandos conflitantes é construída. A fase de execução do ePaxos constrói uma ordem total compatível com essa ordem parcial e sequencialmente a executa de acordo com essa ordem. O Algoritmo 3.1 resume a fase de execução. O algoritmo de execução

visita periodicamente o conjunto de instâncias para detectar uma instância *committed* (conforme linha 27).

```
1:  data structures
2:    I : {c × deps × seq × state |
3:         c ∈ C,                                          {o comando}
4:         deps ∈ I,                          {outras instâncias das quais esta depende}
5:         seq ∈ ℕ,                                {para resolver ciclos, se necessário}
6:         state ∈ {PreAccepted, Accepted, Committed, Executed}
7:        }
8:    G : (N, E)|                                                {grafo de dependência}
9:         N ⊆ I,                                        {os nós são instâncias do consenso}
10:        E ∈ N × N                                    {as arestas são as dependências}
11: procedure G : buildDepGrapg(i,(N,E))
12:   for all j ∈ i.deps | j.state ≠ Executed do                              {i depende de j}
13:     wait until j.state = Committed
14:     N ← N ∪ {j}
15:     E ← E ∪ {(i, j)}
16:     (N, E) ← buildDepGraph(j, (N, E))
17:   return (N, E)                          {um grafo em que todas as instâncias estão commited}
18: procedure sccList : findSCCs((N,E))
                                  {sccList é uma lista de SCCs em G=(N,E) em ordem topológica reversa}
19: procedure instList : sort((N,E))
                          {instList é uma lista com todas as instâncias i ∈ N em ordem crescente de i.seq}
                                  {utilizado para resolver ciclos de forma determinística entre réplicas}
20: procedure execList(sccList)
21:   for all scc ∈ sccList, in reverse topological order do
22:     instList ← sort(scc)                                                {resolve ciclos}
23:     for all inst ∈ instList, in order do
24:       execute(inst.c)
25:       inst.status ← Executed
26: Replica's execution works as follows:
27: upon i = [r, dep, seq, Committed] ∈ I                              {instância i commited}
28:   dg ← buildDepGraph(i, ({i}, ∅))                                {o grafo inicia apenas com i}
29:   sccList ← findSCCs(dg)
30:   execList(sccList)
```

Algoritmo 3.1 – Fase de Execução do ePaxos

Sempre que uma instância for *committed*, um grafo de dependência dessa instância é construído de forma recursiva (linhas 11 a 17). Todas as instâncias no grafo resultante são *committed* (linha 13). A Figura 3.1 (a) mostra um possível grafo de dependência para o nó 1, onde as instâncias são representadas como círculos com números de sequência e as arestas são dependências. Se as instâncias fossem independentes, seus respectivos grafos de dependência retornariam uma única instância, a própria instância. Quando instâncias conflitam elas serão conectadas no grafo de dependência. Se

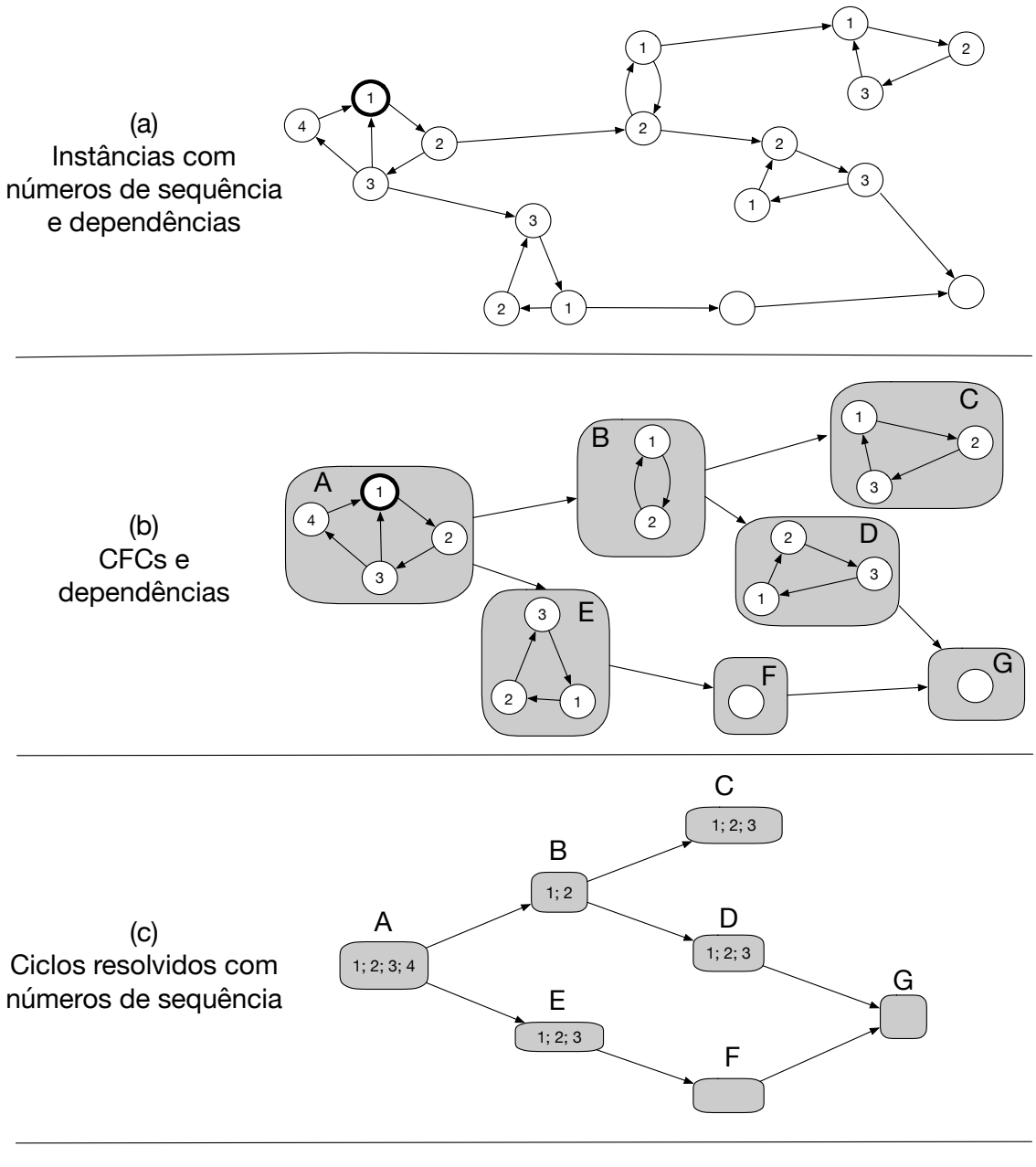

Figura 3.1 – Instâncias e componentes fortemente conexos.

as instâncias entrarem em conflito e colidirem durante o consenso, então componentes fortemente conexos são configurados.

A Figura 3.1 (b) exibe em caixas cinzas os componentes fortemente conexos (CFCs). Os CFCs no grafo de dependência são identificados, resultando em uma lista de CFCs (linha 18) que pode ser obtida através do uso do algoritmo de *Tarjan* [62]. As-

sumimos que CFCs retornam em ordem topológica inversa, ou seja, os elementos mais profundos primeiro, conforme mencionado nas linhas 18 e 21. A Figura 3.1 (d) ilustra uma ordem topológica inversa. Dentro de cada CFC, as instâncias são ordenadas de acordo com o número de sequência fornecido durante o consenso do ePaxos (como por exemplo na linha 19 da Figura 3.1 (c)), e então executadas nessa ordem, se ainda não foram executadas (linhas 23 a 25). Com isso, a mesma ordem total de execução é realizada em cada réplica.

3.2 RME Paralelo a partir de Consenso Generalizado

Protocolos de consenso generalizado evitam a colisão no consenso através da semântica da informação. Essa mesma informação pode ser explorada para obter a execução paralela intra-réplica de comandos independentes, resultando em ganho de vazão em cargas de trabalho que são dominadas por comandos independentes. Nesta seção, nos aprofundamos nesse aspecto.

A Figura 3.2 nos itens (a) e (b) compara uma implementação de RME clássica e abordagens baseadas em consenso generalizado com a nossa proposta, ilustrada no caso (c). Aproveitou-se do consenso generalizado com o objetivo de reduzir a latência e melhorar a vazão ao explorar a identificação de conflitos na etapa do consenso, possibilitando assim escalonar instâncias independentes para execução paralela.

Abordagens paralelas para RME identificam comandos independentes e os processam em paralelo, melhorando assim o *throughput* de execução nas réplicas (Em §5.1 são descritas abordagens relacionadas na literatura). O uso de informações de conflito fornecidas pelo consenso generalizado reduz o custo tanto na identificação quanto na representação de conflitos entre instâncias pendentes de execução nas réplicas do sistema.

3.2.1 Escalonamento de instâncias com base em informações de dependência

A Figura 3.1 (a) mostra as instâncias *commited* e suas dependências, enquanto a Figura 3.1 (b) representa seus CFCs correspondentes identificados. Dois CFCs que não estejam diretamente ou transitivamente conectados pelas arestas de dependência podem ser executados concorrentemente, como CFCs C, D e F na Figura 3.1 (c).

Usando o *ePaxos* como protocolo de consenso, propomos a execução concorrente de instâncias *commited*. Mais precisamente, introduzimos o Algoritmo 3.2, que é derivado do Algoritmo 3.1, descrito na Subseção 3.1.1. O Algoritmo 3.2 mostra em ciano (ou cinza claro) as partes não modificadas do Algoritmo 3.1 e em preto as modificações necessárias para executar concorrentemente CFCS sempre que possível.

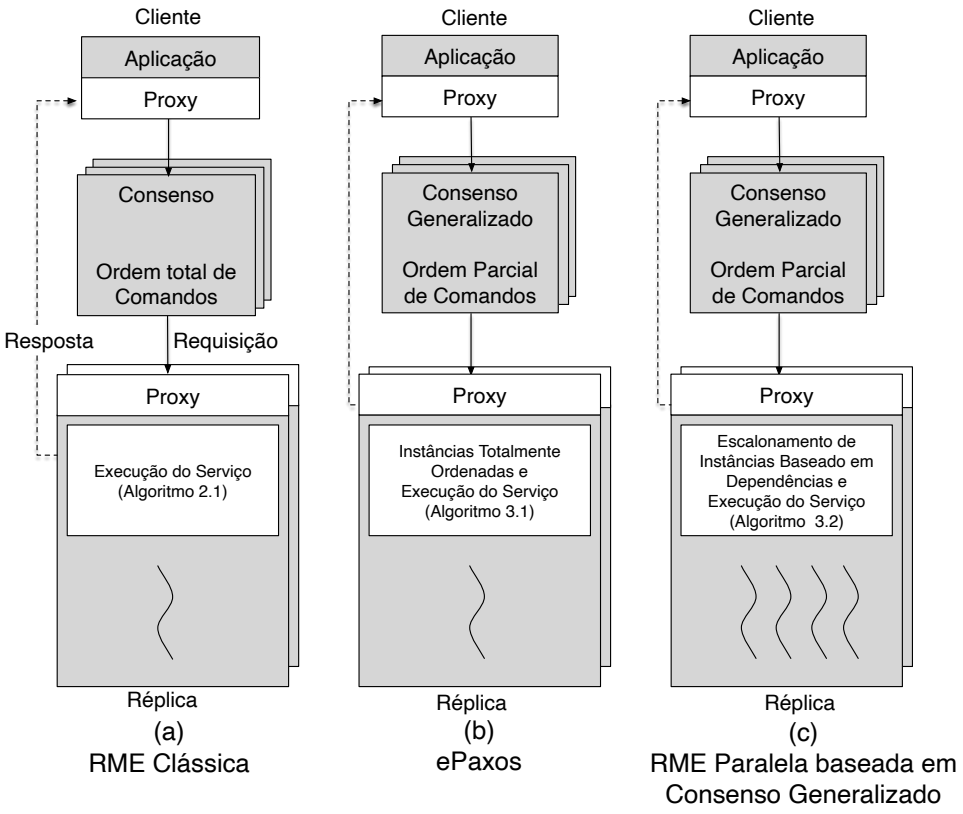

Figura 3.2 – Arquiteturas de RME e RME Paralelo.

Assim como no Algoritmo 3.1, no Algoritmo 3.2, periodicamente e sequencialmente as instâncias são verificadas se foram *committed* (linha 38). O grafo de dependência de uma instância é construído (linhas 15 a 22), e os CFCs são identificados (linha 40). O Algoritmo 3.2 preserva a visita sequencial ao conjunto de instâncias para construir o grafo de dependência.

Entretanto, diferentemente do Algoritmo 3.1, os CFCs identificados são lançados para execução concorrente (linha 36). Agora, ao invés de visitar sequencialmente para encontrar instâncias *committed*, temos essa busca sendo realizada em paralelo. Para evitar que uma instância seja executada duas vezes, ela é marcada como *Executing* quando incluída em um grafo de dependência (linha 16). *Executing* é um novo estado de instância definido para este propósito (linha 7). Este passo não era necessário no algoritmo original tendo em vista que as instâncias mudavam diretamente de *Committed* para *Executed*, garantindo que fossem executadas exatamente uma vez porque a execução de CFCs e a visita aos conjuntos de instâncias ocorriam de forma sequencial.

Nossa implementação utiliza a criação dinâmica de processos (*goRoutines* em GoLang). No Algoritmo 3.2, isso é representado pelas linhas 36 e 25. Para limitar a população máxima de *threads* de trabalho para o número máximo de *threads* (definido por *nWT* na linha 13), utilizou-se um semáforo. Esse semáforo é inicializado com o número

```
1:  data structures
2:    I : { c × deps × seq × state |
3:          c ∈ C,
4:          deps ∈ I,
5:          seq ∈ ℕ,
6:          state ∈ {PreAccepted, Accepted, Committed,
7:                   Executing                          {novo estado de uma instância}
8:                   Executed}
9:        }
10:   G : (N, E)|                                       {grafo de dependência}
11:         N ⊆ I,                                      {os nós são instâncias do consenso}
12:         E ∈ N × N                                   {as arestas são as dependências}
13:   nWT : maximum number of worker threads
14:   avWT : countingSemaphore(nWT)   {crédito de threads trabalhadoras que podem ser lançadas}
15: procedure G : buildDepGraph(i,(N,E))
16:   i.state ← Executing                               {marca como em execução}
17:   for all j ∈ i.deps | j.state ∉ {Executing, Executed} do
18:     wait until j.state = Committed
19:     N ← N ∪ {j}
20:     E ← E ∪ {(i, j)}
21:     (N, E) ← buildDepGraph(j, (N, E))
22:   return (N, E)                  {todas as instâncias são marcadas como Executing}
23: procedure sccList : findSCCs((N,E))
                                          {sccList é uma lista de SCCs em G=(N,E)}
24: procedure instList : sort((N,E))
                {instList é uma lista com todas as instâncias i ∈ N em ordem crescente de i.seq}
                {utilizado para resolver ciclos de forma determinística entre réplicas}
25: procedure concExec(scc) dynamically created thread
26:   instList ← sort(scc)
27:   for all i ∈ instList, in order do                 {para cada instância, em ordem}
28:     for all j ∈ i.deps \ instList do                {as dependências para outros...}
29:       wait j.state = Executed                       {...scc's devem ser resolvidos}
30:     execute(i.c)
31:     i.state ← Executed
32:   avWT.up()                                         {incrementa ou desbloqueia}
33: procedure concExecList(sccList)
34:   for all scc ∈ sccList do
35:     avWT.down()                                     {decrementa ou bloqueia se 0}
36:     start concurrent thread to concExec(scc)
37: Replica's execution works as follows:
38: upon i = [r, dep, seq, Committed] ∈ I              {instância i commited}
39:   dg ← buildDepGraph(i, ({i}, ∅))                 {o grafo inicia apenas com i}
40:   sccList ← findSCCs(dg)
41:   concExecList(sccList)
```

Algoritmo 3.2 – Execução paralela do ePaxos

máximo de *threads* de trabalho e tem seu valor decrementado a medida que uma thread é criada (linha 35) e incrementada quando alguma *thread* de trabalho finaliza (linha 32).

A transição de *Committed* para *Executing* e, em seguida, de *Executing* para *Executed* permite separar o processo de: (i) encontrar sequencialmente os CFCs e lançá-los para execução concorrente de (b) sua execução concorrente.

Assim como no Algoritmo 3.1, *findSCCs* é usado com o grafo de dependência (linha 40) para gerar uma lista de CFCs. Cada CFC nessa lista é então lançado para execução concorrente (linha 36). Embora os CFCs possam ser executados em paralelo se as suas dependências permitirem, as instâncias internas de cada CFC têm uma ordem total (linha 26). Para cada instância pertencente a um CFC, na ordem total, suas dependências que tem relação com instâncias pertencentes a outras CFCs devem ser resolvidas (linhas 28 e 29), e então a instância pode ser executada.

Durante a execução de cada CFCs, as dependências de cada instância são garantidas na linha 28. Isso significa que para executar uma instância, todas as outras instâncias em que ela depende de outros SCCs (excluindo *instList*, que são os nós do SCC) devem ser resolvidas antes.

3.2.2 Corretude

Argumentamos agora que o Algoritmo 3.2 garante que comandos conflitantes são executados na mesma ordem entre as réplicas. Lembramos que ePaxos garante que todas as réplicas tenham as mesma informações de dependência e número de sequência para uma mesma instância.

O Algoritmo 3.2 preserva a visita sequencial periódica e recursiva ao conjunto de instâncias para construir o grafo de dependência. A recursão termina quando não são encontradas dependências ou quando a instância que está sendo visitada já foi executada ou se está sendo executada no momento. Como as instâncias não podem depender de instâncias futuras, o conjunto de dependências consideradas ao construir um grafo de dependência é finito. Além disso, decorrido um tempo, todas as instâncias em que uma instância específica depende são *committed* e podem ser executadas, ou já foram executadas e são resolvidas. Portanto, as instâncias são progressivamente executadas sempre que o ePaxos as efetiva.

Com relação à execução, cada CFC pode ser executado assim que suas dependências em relação a outros CFCs forem resolvidas. CFCs têm uma ordem topológica, ou seja, por definição, não há ciclos entre CFCs. Portanto, não existe a possibilidade de *deadlock*. Dentro de um CFC as instâncias são totalmente ordenadas por um número sequencial, quebrando a possibilidade de ciclos de forma homogênea entre as réplicas. Portanto, basta seguir a ordem dentro do CFC para garantir que, para cada instância, as dependências em relação a outros CFCs sejam respeitadas. Isso garante a mesma ordem

entre CFCs uma vez que todas as réplicas têm as mesmas informações de conflito para cada instância.

3.3 Avaliação de desempenho

Nesta seção apresentamos o *PePaxos*, um protótipo de uma implementação de execução paralela do ePaxos, assim como apresentamos e discutimos os resultados de nossa avaliação experimental.

3.3.1 Implementação

O algoritmo de escalonamento concorrente proposto foi desenvolvido e integrado ao Egalitarian Paxos. Nosso protótipo é baseado na implementação original do ePaxos[2], desenvolvida na linguagem de programação Go em sua versão 1.13 e encontra-se disponível publicamente[3]. Cada CFC é enviado para execução concorrente através de uma *goRoutine*, conforme descrito na linha 36 do Algoritmo 3.2. As linhas 28 e 29 foram implementadas utilizando uma estratégia *busy-wait* que verifica se instâncias específicas em outros CFCs foram *Executed*. O semáforo foi implementado por um canal que possui como tamanho o número total de *threads* de trabalho. Esse canal é definido como cheio e uma operação *down* representam leituras (e uma leitura remove um item) e operações *up* representam escritas de itens de ou para o canal.

3.3.2 Aplicação

Ao avaliar o desempenho de protocolos que utilizam a semântica da aplicação, um aspecto importante a considerar é a taxa de comandos não conflitantes na carga de trabalho. Por exemplo, de acordo com Burrows [10], em uma análise de 10 minutos, *Chubby* experimentou menos de 1% de comandos que poderiam gerar conflitos. Spanner [18] relata que menos de $0,3\%$ de todas as operações no sistema de publicidade do Google (F1) podem gerar conflitos. Já de acordo com Moraru [52], uma probabilidade de conflitos entre 0% e 2% são as mais realistas.

Outro aspecto relevante a considerar é o custo de execução de um comando. O custo de execução de um comando depende essencialmente da natureza da aplicação. Dependendo desse custo, maiores ou menores sobrecargas de paralelização se tornam

[2] https://github.com/efficient/epaxos
[3] https://github.com/tarcisiocjr/pepaxos

aceitáveis. Abordagens de RME paralelas têm considerado uma variedade de aplicações, desde redes sociais [47], base de dados chave-valor [52, 48] e sistemas de arquivos [48].

Ao invés de considerarmos uma aplicação em específico, nós avaliamos o PePaxos através de uma lista ligada (*linked list*) onde tanto o custo do comando quanto o nível de conflito podem ser configuráveis. Diferentes tamanhos de listas são facilmente configuráveis e nos permitem avaliar diferentes custos de execução de comandos. Uma lista ligada possui as seguintes operações:

- *contains*(*int*): verifica se um elemento (ou seja, um número inteiro) está presente na lista; retorna *true* se o elemento *i* está na lista, caso contrário, retorna *false*;
- *add*(*int*): adiciona um elemento na lista; retorna *true* se o item não estiver na lista e *false* caso contrário; e
- *remove*(*int*): remove uma entrada da lista; retorna *false* se o item não estiver na lista e *true* caso contrário.

A partir de agora, nos referimos às operações que verificam se um item está na lista como operações de *leitura* e às operações que adicionam ou removem um item na lista como operações de *escrita*. No modelo de concorrência para esta aplicação, as operações de leitura não entram em conflito umas com as outras, mas entram em conflito com as operações de escrita, que entram em conflito com todas as operações. As operações de escrita bloqueiam toda a lista. Como operações de escrita entram em conflito com qualquer outra operação, utilizamos a probabilidade de conflito como a probabilidade de operações de escrita. O parâmetro inteiro usado em uma operação de leitura é escolhido aleatoriamente. Para manter o custo de execução estável e os experimentos mais controláveis, fixamos a população da lista para os valores desejados (1, 10k, 100k e 1M) durante todo o experimento. Caso contrário teríamos que executar os experimentos até atingir uma população estável e apresentar os resultados de acordo com ela. Portanto, nos experimentos, começamos com uma lista populada e as operações de escrita apenas substituem elementos, mantendo uma população estável.

3.3.3 Metodologia e ambiente de testes

Todos os experimentos foram executados em uma rede local de computadores (LAN). Com objetivo de tolerar até uma falha de parada, tanto PePaxos quanto ePaxos foram configurados com três réplicas. Entre 3 e 3000 clientes foram distribuídos de forma uniforme entre 10 servidores. Cada réplica executou em uma máquina distinta, sendo que cada uma delas possuí quatro processadores AMD Opteron 6366HE com 16 núcleos, executando a 1.8 Ghz, 128GB de memória RAM, discos SATA de estado sólido (SSD), interligadas em uma rede ethernet de 1Gbps. Clientes executam em um servidor com processador

AMD Opteron 2122 com velocidade de 2Ghz, 4GB de memória RAM e em uma interface de rede *ethernet* de 1Gpbs. Todos os servidores foram configurados com o sistema operacional GNU/Linux Ubuntu 18.04 LTS 64bits. O tempo total entre o envio e o recebimento de uma requisição (RTT) entre os servidores é de aproximadamente de 0,1ms .

O estado de cada réplica é mantido na memória principal. Réplicas respondem aos clientes apenas após a execução do comando. Toda mensagem em nossos experimentos tiveram um tamanho fixo de 16 bytes. Utilizou-se agrupamento de mensagens (*batches*) com o objetivo de aumentar a performance. Ou seja, a cada 5ms, ou 1000 comandos, cada *proposer* envia um lote com todos os comandos em sua fila.

O nível de concorrência em nosso protótipo variou entre 1 e 60 rotinas Go permitindo assim a execução paralela de comandos (parâmetro nWT, linha 13 do Algoritmo 3.2). Foram conduzidos experimentos em uma lista de tamanhos 1, $10k$, $100k$ e $1M$, representando operações com diferentes custos de execução. A probabilidade de conflitos variou em 0, 1, 2, 25 e 100%, indicando o percentual de operações de escrita. Tanto o ePaxos quanto PePaxos serializam dois lotes de comandos quando ambos possuem comandos que conflitam entre si. Em nossos experimentos, o tamanho do lote diminui a medida que o tempo de execução dos comandos aumentam. Já onde o custo de execução é elevado, um lote contém aproximadamente um comando. Em todos os experimentos executou-se uma fase de aquecimento de 30 segundos, sendo coletado nos clientes a taxa de transferência e latência do sistema para cada comando emitido nos próximos 60 segundos.

Mediu-se a taxa de transferência máxima, latência e a taxa de transferência no ponto de vazão mais elevado do experimento. O ponto de vazão mais elevado indica o ponto onde a razão entre a vazão e a latência é o máximo. Esse é um indicativo de um ponto de inflexão em que o sistema atinge o seu pico de vazão antes que a latência aumente devido aos efeitos de enfileiramento. Esse ponto pode ser considerado como uma situação de trabalho antes da saturação eminente do sistema.

3.3.4 Resultados

Os resultados a seguir apresentam a vazão e latência alcançados pelo PePaxos, respectivamente, para aplicações com uma lista contendo uma população de 1, 10k, 100k e 1M elementos. Cada figura representa uma configuração entre a possibilidade de conflitos e o número máximo de *threads*. As Figuras 3.3, 3.5, 3.7 e 3.9 representam a vazão máxima atingida pelo sistema. Já os gráficos descritos nas Figuras 3.4, 3.6, 3.8 e 3.10 exibem a vazão e latência no ponto mais elevado de vazão do sistema alcançado em cada configuração. Ressalta-se que não correspondem a valores de uma mesma carga de tra-

balho. Nas Figuras 3.11 e 3.12 tem-se valores de latência e vazão para uma mesma carga de trabalho em uma lista de tamanhos de 10 e 100k elementos.

Operações com custo leve de execução

As Figuras 3.3 e 3.4 mostram os resultados para operações com de custo de execução muito leve. Como pode-se observar, a execução paralela não compensa a sobrecarga gerada sobre o escalonador em múltiplas *threads*. Na Figura 3.3, observamos que a vazão do ePaxos é levemente superior do que a do PePaxos com uma thread: ∼290k ops/s contra ∼275k ops/s. Observamos na Figura 3.3 que à medida que adicionamos *threads*, a taxa de transferência é gradualmente afetada. Além disso, observamos que taxas de conflito mais baixas não levam necessariamente a uma vazão maior.

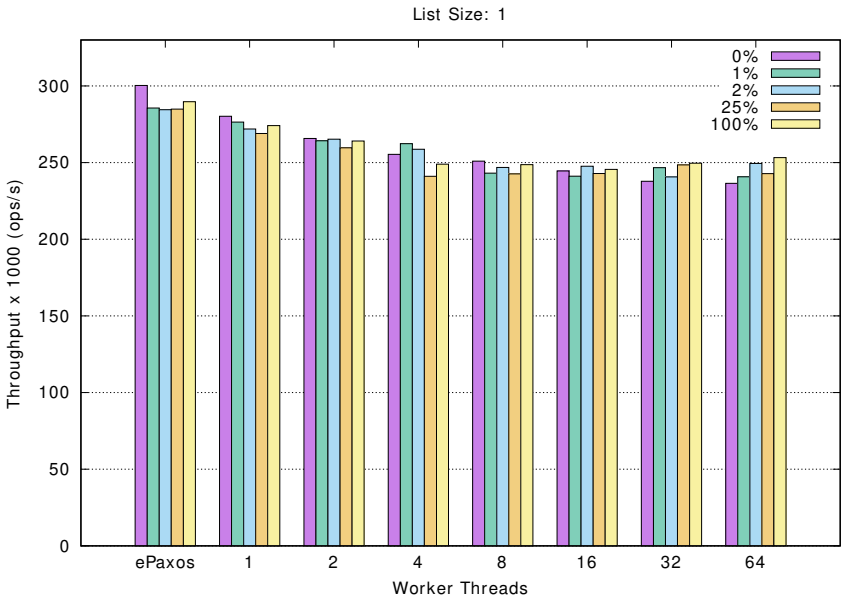

Figura 3.3 – Taxa de transferência máxima variando o número de *threads* e percentual conflitos, para operações de custo muito baixo (tamanho da lista 1)

Como citado anteriormente, o algoritmo possui uma etapa sequencial utilizada para identificar CFCs a serem executados para só então lançá-los para a execução paralela. Essa observação nos permite concluir que a parte sequencial do algoritmo se torna um gargalo em operações com custo leve de execução. A execução sequencial de comandos é mais rápida nessa situação pois a sobrecarga na criação de *threads* por comando é maior do que a execução dos comandos nesse caso e, portanto, o nível de conflito não afeta os resultados de vazão.

Na Figura 3.4 observamos a latência e a vazão associados ao ponto de maior vazão para cada uma das configurações. Todos os valores de latência estabilizam-se em um intervalo próximo, assim como a taxa de transferência. Este efeito é esperado: como a adição de *threads* não ajuda neste caso, o comportamento se aproxima da execução sequencial (ePaxos).

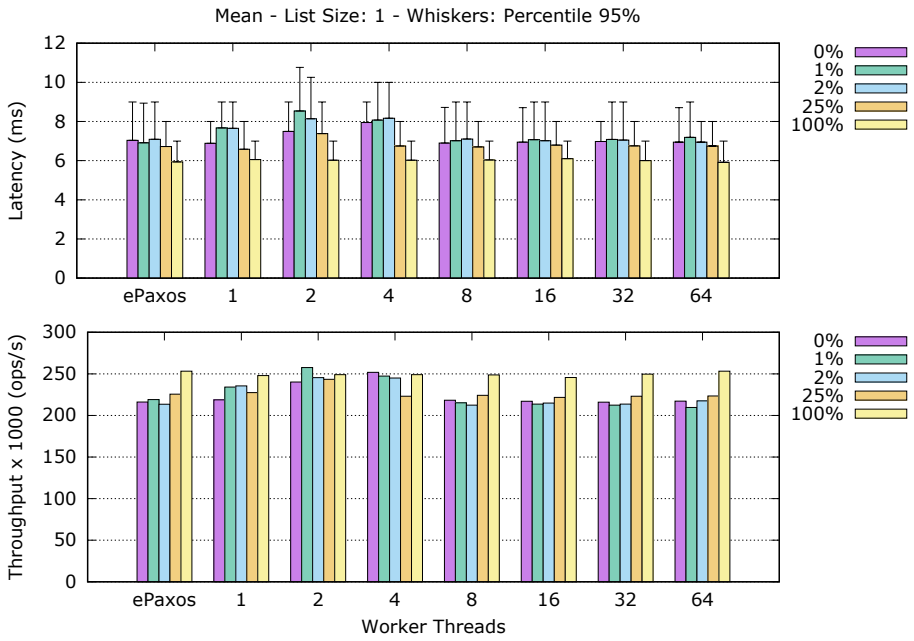

Figura 3.4 – Ponto de vazão mais elevado: latência e vazão para operações de custo muito baixo (tamanho da lista 1)

Operações com custo moderado de execução

As Figuras 3.5 e 3.6 mostram os resultados quando a aplicação lida com uma população de 10k elementos. Nesse gráfico pode-se observar o benefício em utilizar a técnica proposta. Na Figura 3.5 observamos que a taxa de transferência aumenta à medida que adicionamos até 8 *threads*. Para 0%, 1%, 2% e 25% de conflito, PePaxos com 8 *threads* executa respectivamente $\sim 5\times$, $\sim 3\times$, $\sim 2.7\times$ e $\sim 1.2\times$ mais rápido que o ePaxos.

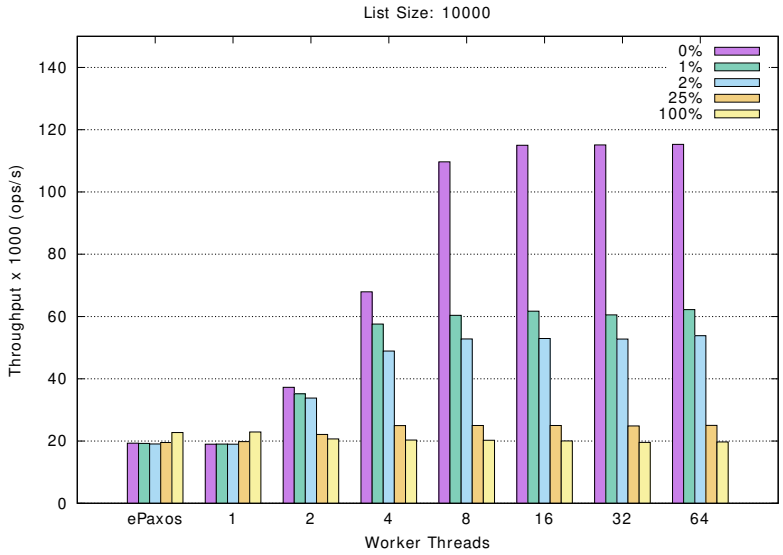

Figura 3.5 – Taxa de transferência máxima para operações de custo moderado (tamanho da lista 10k)

PePaxos com 1 *thread* e 100% de conflitos comporta-se tão bem quanto ePaxos, apenas perdendo desempenho a medida que adiciona-se *threads* devido à sobrecarga adicional gerada pela carga de trabalho sequencial. Após 8 *threads*, observamos os mesmos valores de vazão, indicando que a etapa sequencial do algoritmo impede que a vazão aumente a medida que são adicionadas mais *threads*.

Na Figura 3.6 temos os pontos de maior vazão para cada uma das configurações. Para 0% de conflito, notamos um aumento na latência à medida que o número de *threads* aumenta. Isso se deve pois para cada número de *threads* a maior relação de vazão escolhida teve aumento de throughput, gerado por diferentes cargas de trabalho.

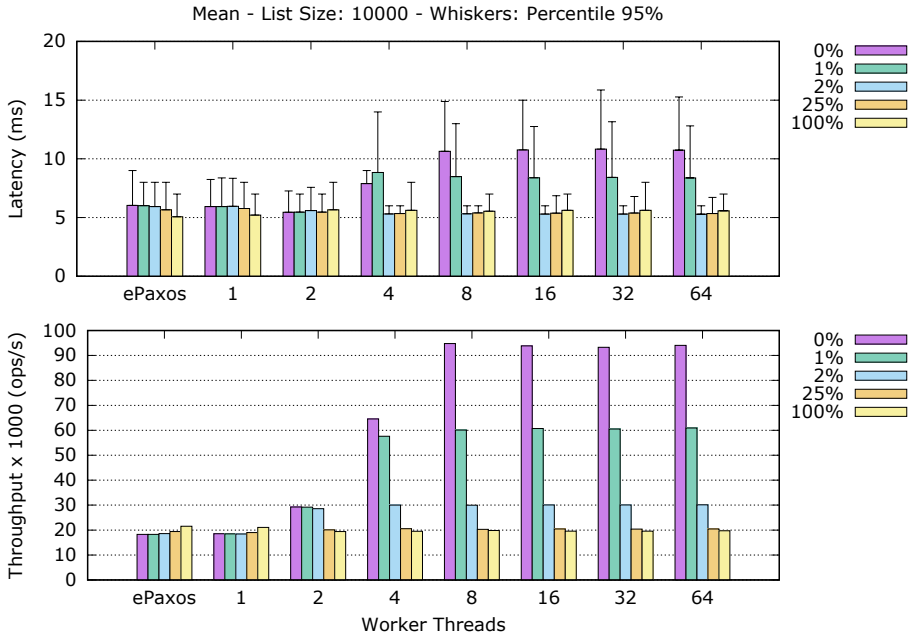

Figura 3.6 – Ponto de vazão mais elevado: latência e vazão para operações de custo moderado (tamanho da lista 10k)

Operações com custo elevado de execução

As Figuras 3.7 e 3.8 mostram os resultados quando a aplicação lida com uma população de 100k elementos, ou seja, 10× a população de execução com custos moderado. Assim, neste cenário, o ePaxos mostra uma perda de vazão de ∼20k ops/s para ∼2k ops/s e um aumento na latência de ∼5 para ∼50 ms se comparado ao caso moderado. Na Figura 3.7 novamente PePaxos com 1 thread mostra resultados de vazão e latência compatível com ePaxos. Com até 64 *threads*, o PePaxos escala a vazão. Para taxas de 0%, 1%, 2% e 25% de conflitos, PePaxos com 64 *threads* executa respectivamente ∼ 18×, ∼ 9×, ∼ 7.2× e ∼ 1.5× mais rápido que ePaxos. Com 100% de conflito e para qualquer número de *threads*, PePaxos funciona tão bem quanto o ePaxos.

Os pontos de maior vazão da Figura 3.8 mostram que em cada configuração, a vazão aumenta com o número de *threads* também para os pontos antes da saturação.

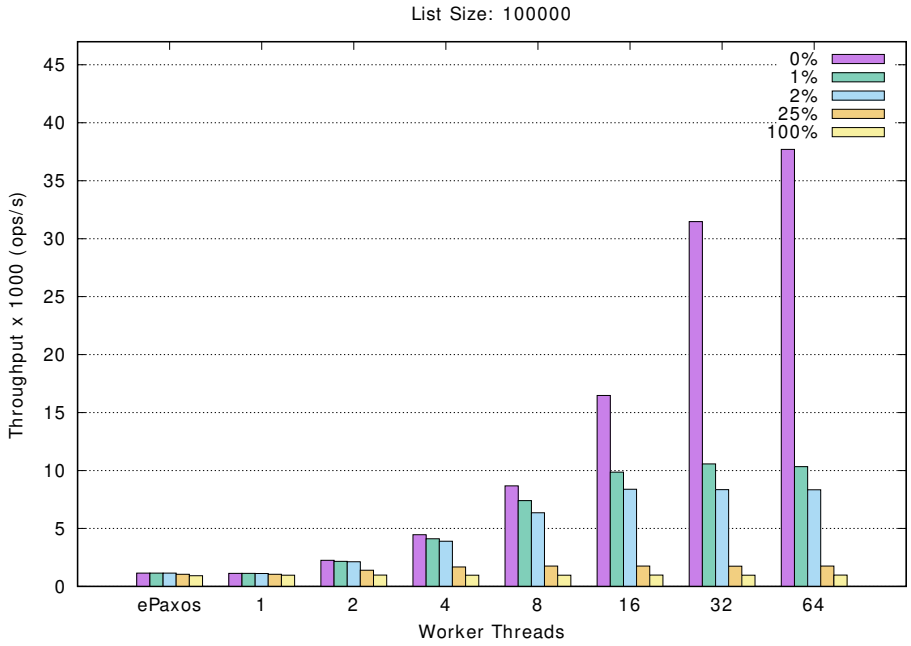

Figura 3.7 – Vazão máxima para operações de custo elevado (tamanho da lista 100k)

Para esses pontos, em geral, as latências acompanham a vazão, estando associadas à população de comandos que estão sendo tratados pela parte sequencial do algoritmo em uma réplica. No intervalo de 16 a 64 *threads*, para conflitos de 0%, são observados ganhos de vazão consideráveis.

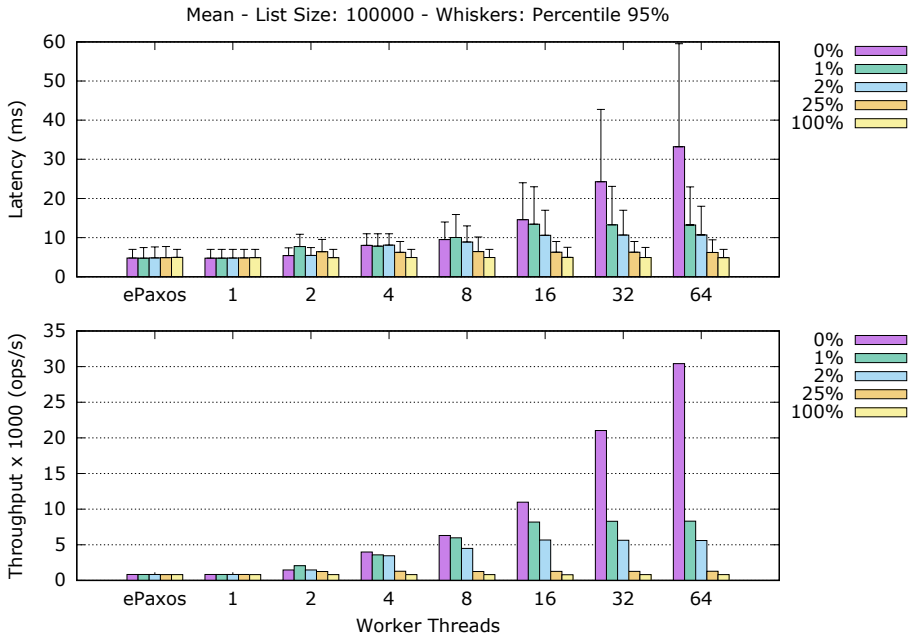

Figura 3.8 – Ponto de vazão mais elevado: latência e vazão para operações de custo elevado (tamanho da lista 100k)

Operações com custo muito elevado de execução

As Figuras 3.9 e 3.10 mostram os resultados quando a aplicação lida com uma população de 1 milhão de elementos. Na Figura 3.9 observamos o mesmo comportamento de vazão da Figura 3.7, porém em um intervalo diferente devido ao tamanho da lista ser 10× maior, o que implica em maiores tempos de execução de comandos.

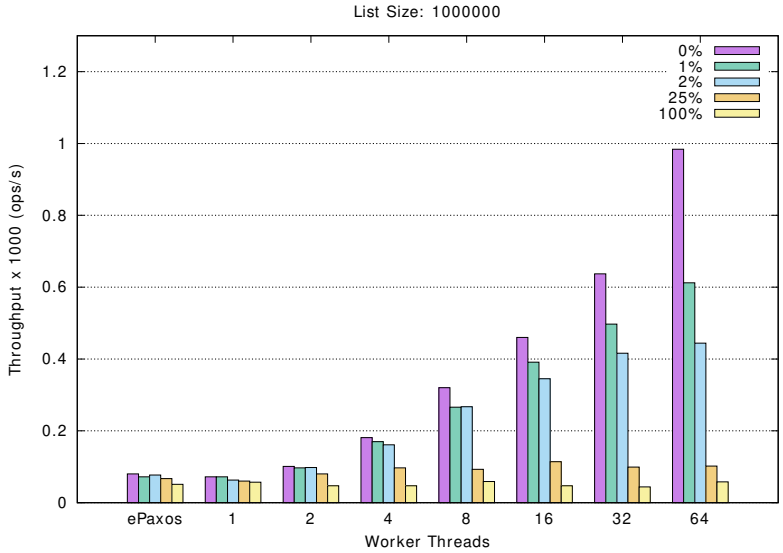

Figura 3.9 – Vazão máxima para operações de custo muito elevado (tamanho da lista 1M)

Diferentemente da Figura 3.8, porém, na Figura 3.10 observamos um comportamento diferente das latências para os pontos de maior vazão. Aqui, as latências geralmente diminuem à medida que a vazão aumenta. Isso revela que tempos maiores de execução de comandos desempenham um papel mais importante comparado à parte sequencial do algoritmo.

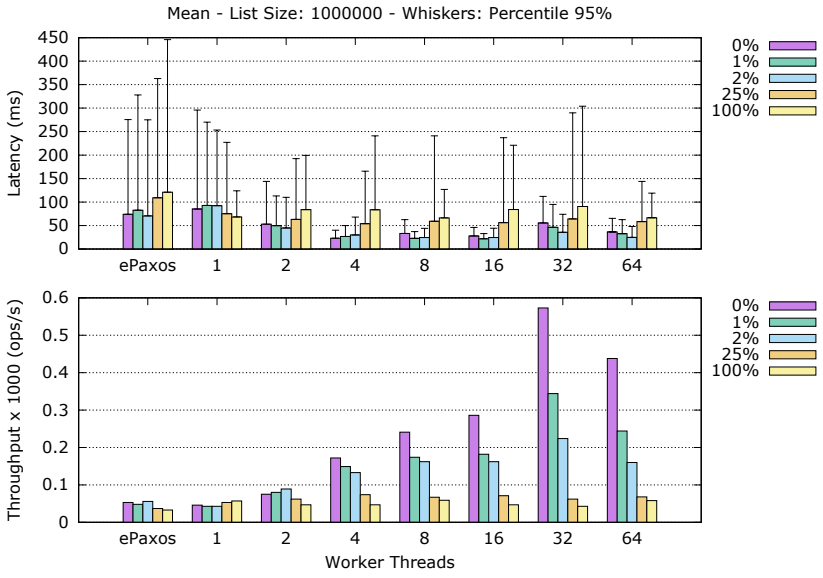

Figura 3.10 – Ponto de vazão mais elevado: latência e vazão para operações de custo muito elevado (tamanho da lista 1M)

Resultados para uma mesma carga de trabalho

Nos experimentos relatados anteriormente, as cargas de trabalho variaram para cada barra pois selecionamos a vazão máxima ou o ponto de vazão mais elevado para cada configuração. Agora ajustamos a carga de trabalho e observamos o comportamento com diferentes configurações. Os resultados são mostrados nas Figuras 3.11 e 3.12.

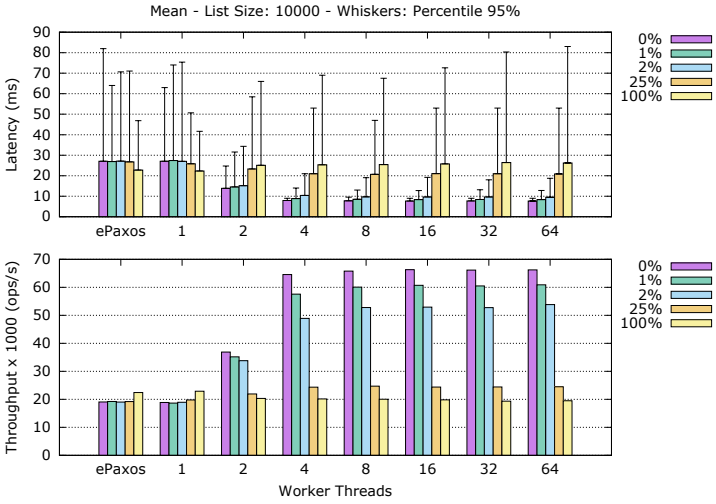

Figura 3.11 – Vazão e latência considerando uma mesma carga de trabalho para uma lista de tamanho 10k

À medida que adicionamos *threads* ao experimento, para uma mesma taxa de conflitos temos um aumento na vazão e uma diminuição na latência. Para a mesma configuração de número de *threads* e para taxas de conflitos crescentes, temos uma diminuição na vazão e um aumento na latência. Essas observações se generalizam para diferentes tamanhos de lista, mas aqui mostramos as configurações de 10k e 100k, com 500 clientes em cada caso. Esse comportamento é seguido, com variações, para diferentes cargas de trabalho (ou diferente número de clientes).

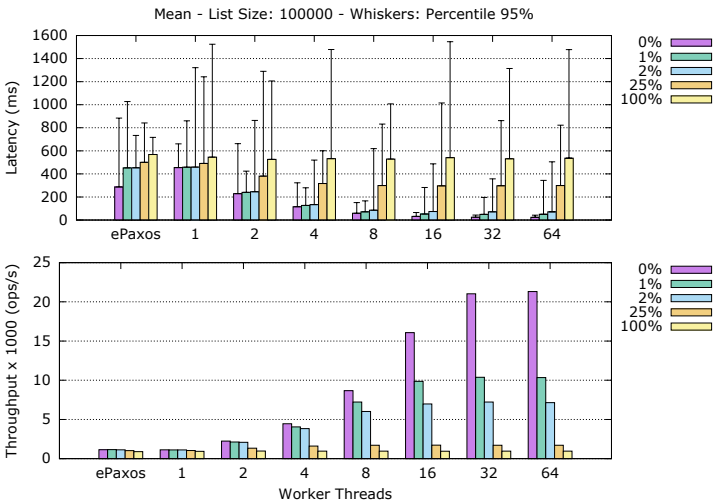

Figura 3.12 – Vazão e latência considerando uma mesma carga de trabalho para uma lista de tamanho 100k

3.4 Discussão

A técnica RME é uma abordagem estabelecida para construir serviços tolerantes a falhas. Em busca de maior vazão na RME, surgiram abordagens que exploram semântica da aplicação na ordenação e execução de comandos. Consenso generalizado e arquiteturas para RME paralelas são dois exemplos de abordagens que fazem uso da semântica da aplicação na ordenação e execução de comandos, respectivamente. Embora ambas as abordagens tenham se mostrado eficazes isoladamente, nenhum estudo na literatura considerou sua integração.

PePaxos [14] defende uma abordagem que se beneficia do consenso generalizado para escalonar a execução paralela de comandos independentes em uma RME. Essa é uma abordagem natural pois as mesmas informações de conflito utilizadas previamente durante a resolução do consenso são utilizadas durante a execução dos comandos. Ao contrário das arquiteturas para RME paralelas que impõem uma ordem total por meio de abordagens típicas de consenso e, em seguida, calculam as dependências dos comandos para sua execução, a abordagem proposta favorece tanto a ordenação quanto a execução de forma integrada.

Neste capítulo, investigamos a integração do consenso generalizado e da RME paralela. Derivamos algoritmos para paralelizar a execução de comandos com base na ordenação fornecida pelo consenso. Como protótipo, estendemos o Egalitarian Paxos e realizamos diversos experimentos variando taxas de conflito, custos computacionais dos comandos e número de núcleos nas réplicas.

Em comparação com o Egalitarian Paxos, a abordagem (a) resulta em importantes ganhos de vazão à medida que a independência dos comandos e o custo computacional aumentam e (b) converge para o mesmo desempenho com altas taxas de conflito ou redução do número de núcleos. Os resultados apresentados nos leva a concluir que utilizar informações de dependência do consenso para favorecer a execução paralela de comandos em RME não é apenas viável, mas também resulta em importantes ganhos de desempenho.

4. CONTRIBUIÇÕES PARA RME PARTICIONADA

Uma das formas utilizadas para obter-se ganho de performance configurável em RME é através do uso de particionamento [8, 47, 17, 16]. Nesse modelo, ao invés do estado da aplicação permanecer totalmente replicado em cada uma das réplicas do sistema, o estado é particionado e cada partição é replicada. Proporcionar um particionamento adequado do estado entre as réplicas de forma a garantir que requisições utilizem o mínimo de partições é um desafio. Nesse sentido, para garantir a propriedade de linearizabilidade em um modelo de execução com operações que alteram mais de uma partição, alguma forma que garanta a ordem sobre o estado existente nas partições deve ser utilizado.

Multicast atômico[1] [29] é uma abstração fundamental utilizada no *design* de sistemas distribuídos que fornece um nível de consistência forte, possibilitando que mensagens sejam propagadas para grupos de processos com confiabilidade e garantias de ordenação. De forma intuitiva, todos processos que não falham e que são destinatários de uma determinada mensagem devem: i) entregar a mensagem e; ii) concordar com a ordenação da mensagem entregue.

Considerando que mensagens podem ser disseminadas para diferentes grupos de destinatários, implementar um protocolo de ordenação utilizando multicast atômico para esse tipo de sistema distribuído é um desafio. Uma maneira simples (e não eficiente) de implementar um protocolo de multicast atômico é através da utilização da difusão atômica. Nessa variação chamada de não-genuína, o protocolo envia a mensagem para todos processos e aqueles processos que não fazem parte do destino da mensagem descartam o conteúdo e não processam a requisição. Ocorre que isso aumenta o custo do protocolo, ou seja, como um protocolo de difusão atômica necessita de uma maioria de processos decidindo, quanto maior for o grupo de processos envolvidos, pior tende a ser o desempenho do sistema.

Para ser eficiente, um algoritmo de multicast atômico necessita ser genuíno. Guerraoui e Schiper [29] introduzem a noção de minimalidade para definir um protocolo de multicast atômico genuíno. Minimalidade é a propriedade do multicast atômico onde uma mensagem é entregue apenas nos destinatários que de fato precisam recebê-la. Um protocolo multicast atômico genuíno é aquele em que apenas os processos endereçados pela mensagem fazem parte do protocolo. Ou seja, a propriedade define que uma mensagem m enviada para um conjunto destinatário $Dst(m)$ envolva apenas o processo de envio da mensagem e o(s) processo(s) contido(s) no conjunto de destino da mensagem ($Dst(m)$). Isso é importante porque evita que mensagens sejam enviadas desnecessariamente para destinatários que não necessitam delas, o que pode causar sobrecarga na

[1] Multicast atômico (*atomic multicast*) não deve ser confundido com primitivas de comunicação de rede como *ip multicast* que oferece garantias de melhor esforço (*best-effort*) de entrega.

rede e reduzir o desempenho do sistema. Um protocolo multicast atômico genuíno não depende de um grupo fixo de processos e não necessita envolver necessariamente todos os processos.

Cabe ressaltar que existem outras variações possíveis do protocolo, como por exemplo uma versão parcialmente-genuína. A variação parcialmente-genuína de um protocolo multicast é uma implementação que garante a entrega para todos os processos destinatários da mensagem, mas que pode envolver processos que não destino desta mensagem. Isso pode ser útil em situações em que a entrega única das mensagens seja importante para processos destinatários, mas não para todos processos do sistema.

Intuitivamente um algoritmo de multicast atômico é mais eficiente, pois sendo possível segmentar o sistema composto de vários processos em distintos subgrupos, o ganho de performance é evidente devido a necessidade de um quórum menor de processos que decidem sobre a ordenação de uma mensagem.

Através do multicast atômico, processos podem enviar mensagens para diferentes grupos de destino com a garantia de que todos os destinos entregam uma mensagem em uma ordem acíclica. Uma ordem acíclica implica que todos os destinos entregam mensagens comuns de forma consistente.

Embora pesquisas em protocolos de multicast atômico explorem a utilização de diferentes topologias para ordenar comandos a partir da semântica da aplicação, não existem estudos onde a semântica é considerada para definir modelos de execução envolvendo tanto a requisição quanto a resposta em uma RME particionada. Neste capítulo investigou-se os ganhos de funcionalidade que modelos de execução aplicados ao retorno podem possibilitar. Nossas contribuições são: (i) desenvolvemos algoritmos para a execução assíncrona de um protocolo de multicast atômico; (ii) definimos diferentes modelos de execução aplicados a uma árvore de sobreposição no contexto de uma RME particionada; e (iii) avaliamos tanto vazão quanto latência das versões apresentadas considerando os modelos propostos;

Esse capítulo é organizado da seguinte forma: A Seção 4.1 contextualiza o trabalho com o uso do ByzCast. A Seção 4.2 apresenta os modelos de execução e sua aplicabilidade. A Seção 4.3 detalha o protótipo e analisa os resultados e na Seção 4.4 o capítulo é concluído.

4.1 ByzCast

Por mais que pesquisas na construção de protocolos de multicast atômico eficientes estejam estabelecidos na literatura [22, 27, 58], até onde se tem conhecimento, com exceção do ByzCast todos os demais protocolos existentes tem como objetivo tolerar falhas benignas (i.e. falhas por colapso).

O trabalho descrito em ByzCast [15] apresenta um protocolo de multicast atômico parcialmente genuíno construído sobre múltiplas instâncias de um protocolo de difusão atômica. De forma ampla, a construção do ByzCast foi feita com dois objetivos principais:

- Reaproveitar soluções já existentes: Pesquisas realizadas em protocolos tolerantes à falhas bizantinas são consideradas consolidadas atualmente (PBFT [13], BFT-SMART [6]). Dessa forma, ao invés de construir um novo protocolo desde o seu princípio, ByzCast consegue reutilizar partes de um protocolo de difusão atômica com suporte a falhas bizantinas;
- Ser um protocolo escalável: Genuinidade é a propriedade que melhor captura escalabilidade em multicast atômico e ao exigir que apenas grupos de destino de uma mensagem coordenem realizem a ordenação dessa mensagem, um protocolo multicast genuíno consegue escalar a medida que o número de grupos aumenta, economizando recursos e ganhando desempenho.

Conforme definido em [15], ByzCast utiliza uma árvore de sobreposição onde cada nó da árvore representa um grupo de processos. Cada grupo de processos representa uma instância de difusão atômica. Com isso, uma requisição de multicast atômico para um grupo refere-se a utilização de uma instância de difusão atômica implementado pelo grupo de destino. Todavia, mensagens endereçadas a múltiplos grupos necessitam de tratamento especial na sua ordenação.

No ByzCast, a forma utilizada para realizar a ordenação de mensagens destinadas para múltiplos grupos é através do uso de grupos auxiliares em comum entre os grupos de destinos das mensagens (nesse caso, é utilizado o menor ancestral comum[2]). Dessa forma, uma mensagem m destinada a múltiplos grupos é recursivamente ordenada pelo seu ancestral comum até chegar aos seus grupos de destinos. A principal invariante no ByzCast é que grupos mais baixos na árvore sempre irão preservar a ordem induzida pelo seu ancestral comum.

A Figura 4.1 ilustra como a lógica descrita é aplicada na execução de uma solicitação enviada por um cliente quando uma mensagem é endereçada para mais de um grupo da árvore. Exceto pelas requisições de um cliente, que são sempre únicas, réplicas de um determinado grupo só processam mensagens enviadas por um grupo superior quando receberem $f + 1$ mensagens *a-deliver* do mesmo. Tanto clientes quanto grupos intermediários aguardam $f + 1$ respostas corretas para prosseguir no processamento da resposta de uma mensagem.

Conforme propriedades do multicast atômico (§2.2.3), todo grupo x no ByzCast, seja ele auxiliar ou destino da mensagem, implementa difusão atômica FIFO. O funciona-

[2] Menor ancestral comum ou *lowest common ancestor (LCA)* de dois nós v e w em uma árvore ou de um grafo acíclico dirigido (DAG) T é o nó mais baixo (ou seja, mais profundo) que tem v e w como descendentes.

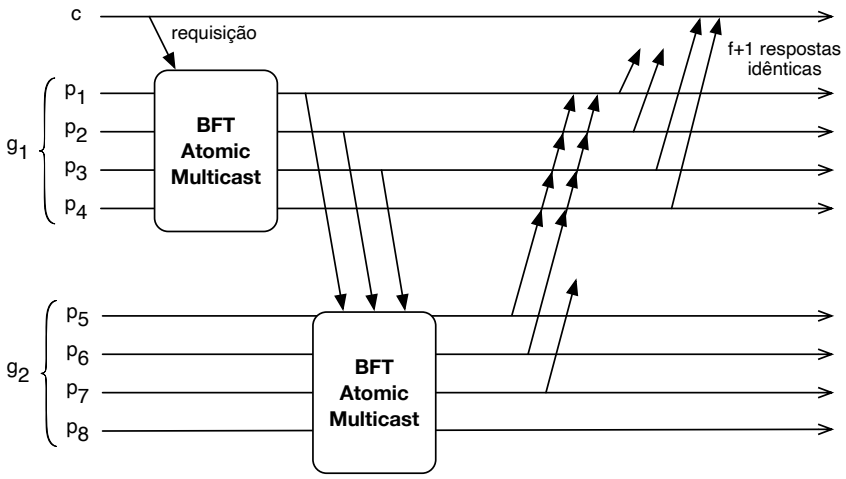

Figura 4.1 – Execução de uma mensagem global no ByzCast destinada a ser executada em $\{g_1, g_2\}$. Cada grupo possui quatro processos tolerando falhas bizantinas de um processo.

mento básico do ByzCast é descrito no Algoritmo 4.1. Para enviar uma mensagem m, um cliente realiza *a-broadcast* em cada processo do grupo de *ancestral comum* dos destinos da mensagem.

```
1:  Initialization
2:    T is an overlay tree with groups Γ ∪ Λ
3:    A-delivered ← ∅
4:  To a-multicast message m:
5:    x_0 ← lca(m.dst)                    {menor ancestral comum de m.dst}
6:    x_0-broadcast(m)
7:  Each server process p in group x_k executes as follows:
8:    when x_k-deliver(m)
9:      if k = 0 or x_k-delivered m (f + 1) times then
10:       for each x_{k+1} ∈ children(x_k) such that
                    m.dst ∩ reach(x_{k+1}) ≠ ∅ do
11:         x_{k+1}-broadcast(m)
12:       if x_k ∈ m.dst and m ∉ A-delivered then
13:         a-deliver(m)
14:         A-delivered ← A-delivered ∪ {m}
```

Algoritmo 4.1 – ByzCast

Assume-se que os clientes conhecem a topologia da árvore e que através de uma função sobre os destinos da mensagem $m.dst$, é possível obter-se o ponto de entrada da mensagem na árvore. Ou seja, para calcular o caminho até um grupo x, o cliente executa uma função $reach(x)$, que é definida pelo conjunto de destinos possíveis de serem alcançados de x ao percorrer a árvore.

Quando m é x_k-*deliver* pelos processos em x_k, cada processo x_{k+1}-*broadcast* m nos grupos filhos de x_k se x_{k+1} fazem parte do destino ou são necessários para alcançarem o destino de $m.dst$.

Esse processo continua de forma recursiva até que *a-deliver*(m) venha a ser executado nos grupos de destino de $m.dst$.

De forma intuitiva, as garantias de ordenação fornecidas pelo ByzCast são consequências de duas invariantes [15]:

- Quaisquer duas mensagens m e m' enviadas através de multicast atômico são ordenadas por um grupo em comum na árvore.
- Se m é ordenado antes de m' no grupo em comum (x_k) do destino, então m é ordenado antes de m' em qualquer outro grupo que também venha a ordenar as duas mensagens (Graças a difusão atômica FIFO utilizada em cada grupo da árvore).

Para garantir tolerância a falhas bizantinas em um grupo x_k, processos em x_{k+1} somente aceitam m quando m foi x_{k+1}-*delived* pelo menos $f+1$ vezes. Isso garante que m foi x_{k+1}-*broadcast* por ao menos um processo correto de x_k e por indução m foi *a-multicast* por um cliente, e não fabricada por um servidor malicioso.

4.2 ByzCast: de Multicast para RME particionada

ByzCast foi proposto como um algoritmo de multicast atômico, conforme descrito anteriormente. Entretanto, propomos o uso do mesmo como ponto de partida para contemplar tanto o multicast atômico como diferentes modelos de execução envolvendo tanto a requisição quanto a resposta em uma RME particionada.

Entendemos cada nó do ByzCast como uma partição RME e caracterizamos como *modelo de execução* a forma como as requisições são tratadas e respondidas ao cliente pelo conjunto de partições envolvidas em uma requisição.

Na Figura 4.2 propomos 3 modelos utilizando uma árvore ByzCast: (a) Uma requisição multicast se disseminando, (b) uma requisição multicast no modelo RME (com retorno) onde toda resposta recebida de um nível inferior é concatenada com a resposta do nível atual antes de ser encaminhada para o nível acima e (c) Uma requisição multicast no modelo RME com alguma função de tratamento aplicada sobre as respostas antes das mesmas serem encaminhadas acima. Esses modelos são detalhados nas próximas seções.

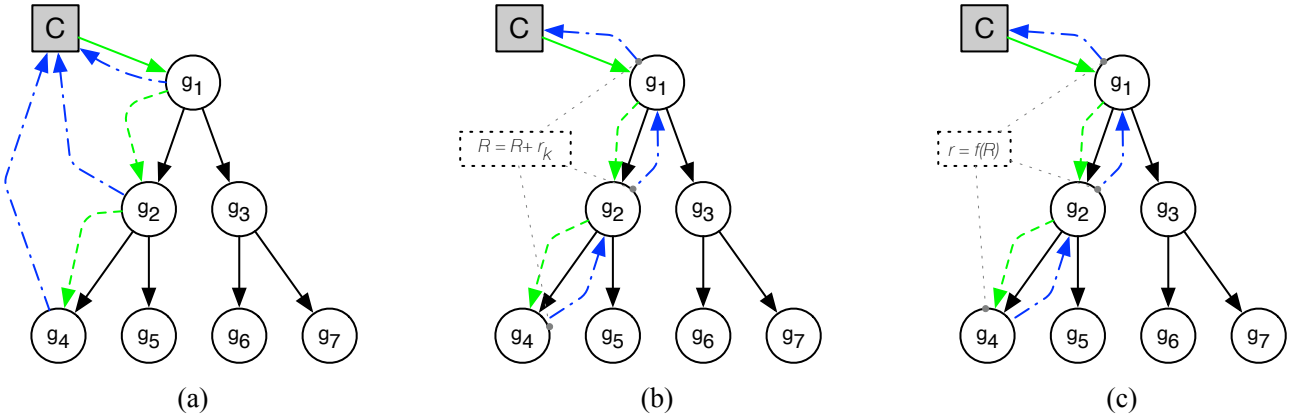

Figura 4.2 – Modelos de execução propostos utilizando topologia em árvore de sobreposição do ByzCast.

4.2.1 RME Particionada - Modelo A

No presente modelo, o protocolo de multicast atômico garante somente a entrega da requisição, tal como definido pelas propriedades do protocolo (§2.2.3). A Figura 4.3 (a) apresenta o modelo de execução considerando apenas as propriedades do multicast atômico.

Por não necessitar responder diretamente ao cliente, aplicações baseadas neste modelo tendem a serem mais simples e performáticas, mas considerando uma abordagem que emprega a técnica de RME onde é esperado o retorno de uma resposta para o cliente, identificamos duas formas quanto ao formato do retorno da resposta: (i) uma resposta contendo a modificação ou leitura do estado da respectiva partição; ou (ii) uma resposta vazia contendo apenas uma confirmação de entrega.

Conforme ilustrado na Figura 4.3 (b), tanto a execução quanto o retorno ao cliente são realizados por cada partição de destino da requisição, cabendo ao cliente o tratamento da resposta a cada requisição enviada.

4.2.2 RME Particionada - Modelo B

Apresentamos agora uma extensão do modelo para contemplar o tratamento e retorno das requisições. Para possibilitar esse tratamento, cada requisição é submetida à execução em cada partição envolvida, e a partir disso cada partição aguarda o resultado de seus descendentes, caso existam, e então retorna esses resultados concatenados ao seu ancestral. O funcionamento ocorre de maneira recursiva.

Nesse modelo, conforme Figura 4.4 (b), cada nó deve manter o controle se uma requisição está em andamento e em que fase ela se encontra, a fim de possibilitar o tra-

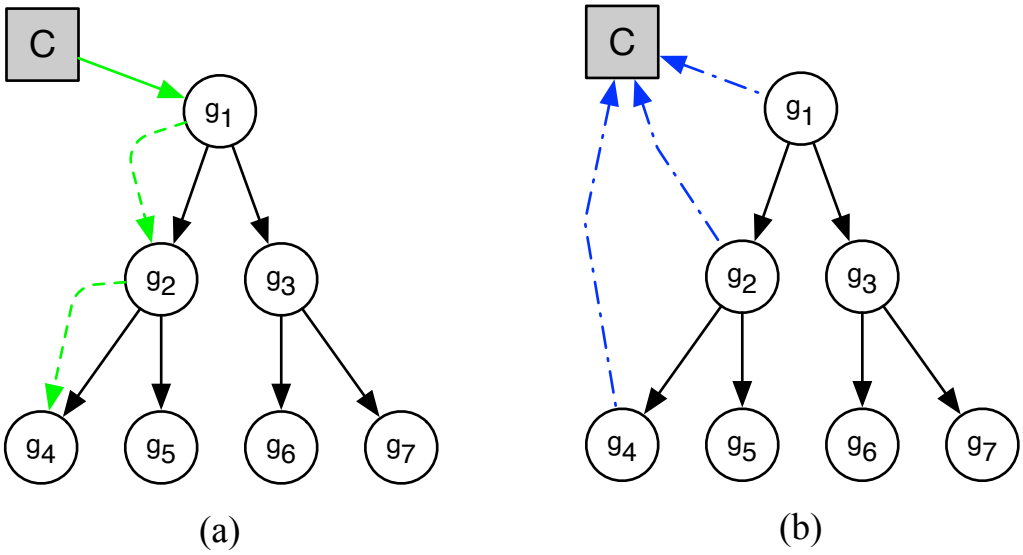

Figura 4.3 – Uma árvore de sobreposição do ByzCast representando (a) disseminação de uma mensagem m através de multicast atômico para os grupos $\{g_1, g_2, g_4\}$ e (b) uma resposta considerando o uso da técnica RME

tamento das respostas. Identificamos duas formas principais para realizar este controle, detalhadas a seguir.

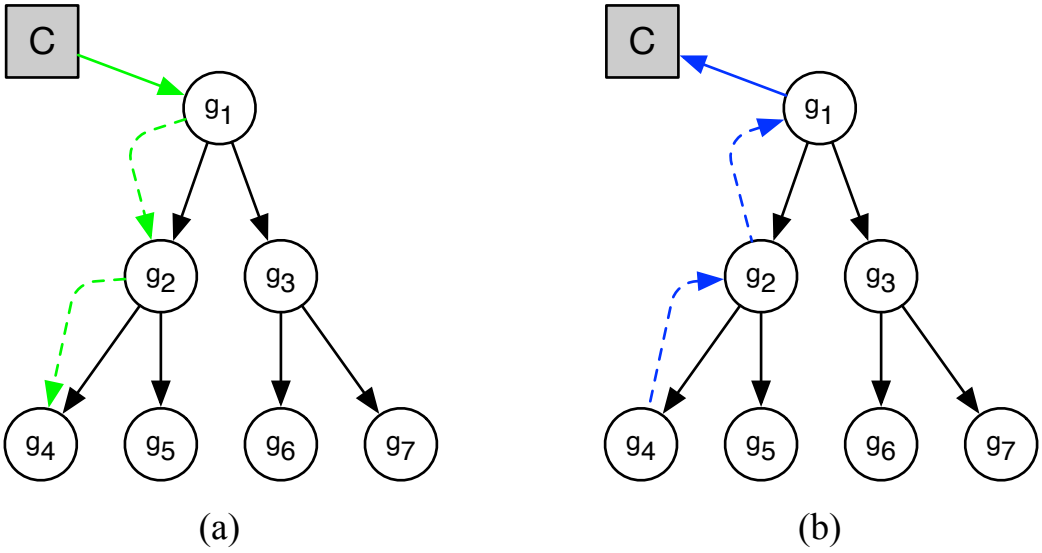

Figura 4.4 – Uma árvore de sobreposição do ByzCast representando (a) disseminação de uma mensagem m através de multicast atômico para os grupos $\{g_1, g_2, g_4\}$ e (b) resposta concatenada para o cliente no modelo RME

Variante 1 - Execução Síncrona

Nessa variante, cada nó ao receber uma mensagem avalia se o próprio grupo deve executar e/ou para quais descendentes deve repassar essa mensagem. Caso o nó seja um destinatário da requisição, ele a executa e cria *threads* para repassar a requisição

para os seus descendentes. Cada uma destas *threads* repassa o pedido e fica à espera do resultado. Quando todas as *threads* recebem seus retornos, pode-se concatená-los ao resultado da execução local e então retornar o resultado para o ancestral que fez a solicitação original.

Dado o funcionamento recursivo, cada nó que dissemina a requisição também aglutina resultados conforme a sua posição na topologia. Nesta variante temos a criação dinâmica de *threads* onde o número de *threads* em um nó é proporcional ao número de requisições em tratamento pelos nós. Na prática, um pedido gera uma *thread* de execução local e uma *thread* para repassar e aguardar o retorno para cada descendente que receber a requisição. Naturalmente surge o questionamento sobre como limitar o número de *threads* na arquitetura de disseminação, dando origem à próxima variante.

Variante 2 - Execução Assíncrona

No algoritmo de repasse assíncrono proposto nesta variante, quando um nó recebe uma mensagem, ele gera um registro local do pedido em andamento, repassa aos seus nós descendentes, e executa se for o caso. O retorno dos descendentes ocorre de forma assíncrona. Quando um descendente gera um retorno, uma operação configurável é chamada pela plataforma de comunicação. Neste evento, o computo dos resultados sobre o estado local é realizado e se o resultado está completo, é feito o retorno para o seu ancestral. Assim, a modelagem de um nó funciona de forma reativa: reage tanto para solicitações de ancestrais quanto para retornos de seus descendentes.

Descrito em detalhes a seguir, o Algoritmo 4.2 descreve o funcionamento da versão assíncrona proposta.

Definido na linha 3, uma instância é uma estrutura composta por informações que indicam: *orig*: origem da requisição; *id*: identificador único da requisição; *msg*: o seu conteúdo da mensagem a ser executada; *nReq*: o número de requisições que já foram recebidas desta mesma requisição do nível acima; *rRep*: o número de respostas recebidas dos níveis abaixo; *st*: o valor do estado atual da instância; e *repContent*: o conteúdo das respostas recebidas.

Nesse contexto, são duas opções possíveis para definirmos a origem (*orig*) de uma requisição: Ou ela foi enviada por um cliente ou foi enviada por um grupo acima na topologia da árvore ByzCast.

O estado (*st*) de uma instância pode ser definido (linha 4) da seguinte forma: *rcvQinc*: a instância não possui quórum completo de requisições; *rcvd*: a instância foi recebida;*forwarded*: a instância foi repassada;*executed*: a instância foi executada pela aplicação;*repQinc*: a instância não possui quórum completo de respostas;*repQcomp*: a instância formou $f + 1$ respostas e formou quórum; e *replied*: a instância foi respondida para o nível acima da árvore ou para o cliente.

1: Each server process p in group x_k executes as follows:
2: **Types**
3: $Instance \rightarrow \{orig, id, msg, nReq, nRep, st, repContent\}$
4: st : {rcvQinc, rcvd, forwarded, executed, repQinc, repQcomp, replied }
5: $orig$: *{identificador do grupo acima ou do cliente}*
6: id : Nat *{identificador único da mensagem}*
7: $nRcv$: Nat *{quantidade de requisições recebidas}*
8: $nRep$: Nat *{quantidade de respostas recebidas}*
9:
10: $repContent$: conteúdo do reply
11: **Variables**
12: ist : instance set $\leftarrow \emptyset$ *{registro de cada instancia e seu andamento em p}*
13: **when** g_k-deliver$_{bftsmart}(\langle orig, id, msg \rangle)$
14: **if** $id \in ist[id]$ **then** *{essa instância é conhecida?}*
15: **if** $g_k = lca(m.dst)$ **then** *{estou recebendo do cliente?}*
16: $ist \leftarrow ist \cup \{\langle orig, id, msg, 1, 0, rcvd, null \rangle\}$ *{considera recebida}*
17: **else** *{recebendo do grupo de cima}*
18: $ist \leftarrow ist \cup \{\langle orig, id, msg, 1, 0, rcvQinc, null \rangle\}$
19: **else** *{quórum incompleto}*
20: $ist[id].nReq++$
21: **if** $ist[id].nReq = (f+1) \wedge ist[id].st = rcvQinc$ **then** *{quórum completo}*
22: $ist[id].st \leftarrow rcvd$ *{considera recebida}*
23: **when** $\exists\, i \in ist : i.st = rcvd \wedge children(g_k) \neq \emptyset$ *{pode repassar}*
24: **for each** $g_l \in children(g_k)$ such that
$msg.dst \cap reach(g_l) \neq \emptyset$ **do**
25: g_l-broadcast$(newMsg(id(g_k), i.id, i.msg))$ *{envia para cada processo no grupo l}*
26: **if** $g_k \in i.msg.dst$ **then** *{é destinatário intermediário}*
27: $ist[i.id].repContent \leftarrow ist[i.id].repContent \cup \{upcall$-**aDeliverAndExec**(i.msg) $\}$
28: $ist[i.id] \leftarrow forwarded$
29: **when** $\exists\, i \in ist : i.st = rcvd \wedge children(g_k) = \emptyset$ *{pode executar}*
30: $ist[i.id].repContent \leftarrow ist[i.id].repContent \cup \{upcall$-**aDeliverAndExec**(i.msg)$\}$
31: $ist[i.id] \leftarrow executed$
32: **when** $\exists\, i \in ist : i.st \in \{executed, repQcomp\}$ *{pode responder}*
33: **for each** process $q \in group(i.orig)$ **do**
34: $reply(q, newRep(g_k, i.id, repContent))$
35: $ist[i.id].st \leftarrow replied$ *{marca como respondida}*
36: **when** $upcall_{bftsmart}handleReply(\langle orig, id, repC \rangle)$
37: $ist[i.id].nRep++$
38: $ist[i.id].repContent \leftarrow ist[i.id].repContent \cup repC$ *{coleciona todas as respostas}*
39: **if** $ist[i.id].nRep = (f+1)$ **then**
40: $i(id).st = repQcomp$ *{declara completo e pode responder}*

Algoritmo 4.2 – ByzCast assíncrono

O conjunto de todas instâncias registradas em um processo p é armazenado na variável *ist*.

BFT-SMART é um protocolo que implementa difusão atômica. ByzCast é um protocolo que utiliza o BFT-SMART como uma plataforma de implementação de um protocolo de multicast atômico parcialmente genuíno.

Uma mensagem *msg* é entregue em cada processo p de um grupo g_k pelo BFT-SMART através de uma *upcall* (linha 13). Caso essa mensagem (referenciada como instância no protocolo) tenha sido recebida anteriormente (linha 14), verifica-se a sua origem para tratamento posterior. Como cada grupo conhece a topologia da árvore ByzCast por completo, quando o próprio grupo se identifica como ancestral comum (*lca*(*msg*.*dst*)) da mensagem, entende-se que a mensagem foi recebida diretamente de um cliente. Sendo a mensagem recebida diretamente de um cliente não se faz necessário a formação de quórum de recebimento (*nReq*). Com isso, cada processo p armazena a mensagem em seu conjunto de instâncias e define o status da mesma como recebida (*rcvd*) para a sua posterior resposta (descrito na linha 29).

Sendo o envio da mensagem realizado por um grupo acima na topologia da árvore, a mensagem também é armazenada no conjunto de instâncias (linha 18), mas desta vez com o status de *rcvQinc* indicando a necessidade de outros recebimentos (*nReq*) da mesma mensagem para a formação de quórum, necessário para execução e/ou encaminhamento dessa mensagem.

Uma instância que já foi previamente recebida (linha 14) em p necessita formar quórum em cada processo do grupo no próximo nível (g_{k+1}) da árvore (linha 17). Uma instância é definida como recebida (*rcvd*) quando um quórum com $f + 1$ mensagens são recebidas (*nReq*) do nível acima (linhas 21 e 22).

Instâncias *rcvd* que necessitam ser encaminhadas para uma nível abaixo de g_k são tratadas entre as linhas 23 e 28. Esse envio é realizado por difusão atômica do BFT-SMART (linha 25) para cada filho do grupo g_k que seja o destino de *msg* ou um grupo intermediário necessário para a entrega de *msg* até o seu destino. Caso o grupo atual g_k seja um destinatário da mensagem, o conteúdo da resposta da execução pela aplicação é concatenado no conjunto de respostas (*repContent*) desta instância. Note-se que a instância é definida como *forwarded* pois nessa situação o grupo atual g_k pode ser um destino e/ou um intermediário da mensagem. Ou seja, além de executar a mensagem em seu nível atual (se for um grupo destino da mensagem), cada processo p em g_k também deve aguardar as respostas enviadas pelos níveis abaixos, para somente então responder para o nível acima.

Assume-se que uma instância definida com o status de recebida (*rcvd*) por um grupo qualquer, mas que este esteja localizado na folha da árvore podem executar a mensagem (definindo o status desta instância como *st* = *executada*) e responder ao originador

da mensagem. Note-se que a origem nesse caso em específico pode ser um grupo do nível superior ou um cliente.

A plataforma de difusão atômica utilizada entrega cada uma das respostas do grupo do nível abaixo através de uma *upcall* (linha 36), onde são contabilizadas. Todas as respostas obtidas são concatenadas e armazenadas em *repContent* dessa instância (linha 37) Com isso, cada processo p em g_k, ao obter uma quantidade de respostas que suporte o número de falhas desejado (linha 39), define o status da instância como *repQcomp* (linha 40), podendo esta ser respondida a seguir.

Uma instância pode ser respondida quando seu status encontra-se definido como *executada* (linha 31) ou *repQcomp* (linha 40). Como o algoritmo é recursivo, independente da posição do processo p de g_k na árvore ByzCast, a resposta do nível abaixo é concatenada as respostas do nível atual (se pertinente). Formado o quórum, a instância pode ser submetida para o remetente, sendo este um grupo acima na árvore ou um cliente, caso este último tenha submetido a mensagem diretamente para o grupo que esteja processando a requisição.

Note-se que devido o funcionamento recursivo do algoritmo, um grupo responde uma mensagem para o nível acima somente quando as respostas do grupo de destino de *msg* localizado mais abaixo na árvore são recebidas e contabilizadas. Este modelo de execução pode ser utilizado em todos os casos onde requisições da RME devem ser executadas em mais de uma partição e sua aplicabilidade é discutida em diferentes cenários de forma comparativa na próxima seção.

4.2.3 RME Particionada - Modelo C

Neste modelo, em vez de realizar a concatenação dos resultados obtidos em cada nível da árvore em que a requisição atua, agora o retorno da operação pode ser transformada pela utilização de uma função, conforme a semântica da aplicação. Esse tipo de situação é encontrado, por exemplo, em plataformas de computação de alto desempenho, como o MPI, onde é possível definir que todo retorno de um conjunto de processos passe por um operador, como o *reduce*, que soma todos os valores para gerar o resultado final utilizado pelo requerente.

O Algoritmo 4.3 apresenta a modificação necessária no bloco responsável pelas respostas descritas anteriormente no Algoritmo 4.2.

Para modificar o modelo de execução de respostas do ByzCast para uma função aplicada sobre os resultados, é necessário alterar a forma como o protocolo trata as respostas em cada nó da árvore. Ou seja, além de coletar cada uma das respostas dos níveis de destino da mensagem, o protocolo deve agora aplicar uma função sobre esses resultados antes de encaminhá-los para o grupo superior (ou ao cliente).

1: **Types**
2: *repContent* *{conteúdo da resposta concatenada}*
3: *fnContent* *{conteúdo da resposta após função de tratamento}*
4: **when** $\exists\ i \in ist : i.st \in \{executed, repQcomp\}$ *{pode responder}*
5: $fnContent \leftarrow fnReplies(i.repContent)$ *{função genérica de tratamento}*
6: **for each** process $q \in group(i.orig)$ **do**
7: $reply(q, newRep(g_k, i.id, fnContent))$ *{responde}*
8: $ist[i.id].st \leftarrow replied$ *{marca como respondida}*

Algoritmo 4.3 – Tratamento de respostas

No algoritmo 4.3, adicionou-se na linha 5 uma chamada para a função genérica descrita no Algoritmo 4.4. Essa função recebe o conjunto de respostas de uma determinada instância e aplica algum tipo de tratamento sobre o seu conteúdo, de acordo com a semântica da aplicação utilizada. O retorno dessa função é enviado como resposta para o grupo acima na árvore, ou para o cliente, finalizando o processamento da requisição.

1: **procedure** fnReplies(*repContent*)
2: $result = \emptyset$
3: **for each** replies $r \in repContent$ **do**
4: do something with r *{faça algo com cada resposta recebida}*
5: **return** *result*

Algoritmo 4.4 – Função genéria que atua sobre respostas

Este modelo de execução pode ser utilizado em situações em que é permitido o tratamento comum dos retornos antes de serem repassados. Observe que essa condição depende da semântica da aplicação

4.2.4 Aplicabilidade

A seguir, consideramos diferentes cenários de aplicações distribuídas em multi-partições utilizando os dois modelos descritos anteriormente.

- Cenário 1: Considere uma base de dados chave-valor particionada entre múltiplos grupos conforme apresentado na Figura 4.5. Cada grupo é responsável por um escopo limitado de chaves (p. ex. $g_1 = \{0..999\}$ e $g_2 = \{1000..1999\}$ e $g_3 = \{2000..2999\}$).

  Uma consulta de um intervalo de chaves é realizada e a mesma excede os limites de uma partição (p. ex. quais valores das chaves entre 1500 e 2500).

  - Concatenado: Cada partição envolvida na consulta retorna os valores para as chaves solicitadas que estão em sua base. O retorno de cada partição é conca-

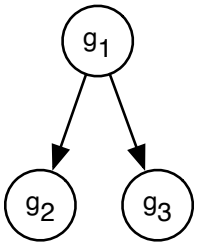

Figura 4.5 – Uma base chave-valor é particionada entre três grupos utilizando uma topologia de árvore do ByzCast

tenado em todos os níveis até o cliente. Ou seja, os valores de g_2 e g_3 são concatenados em g_1 e encaminhados para o cliente. Dessa forma, cabe ao cliente realizar o processamento do retorno com os resultados de múltiplas partições.

- Função: Entretanto, deseja-se oferecer um retorno transparente ao cliente. Ou seja, não basta somente repassar os resultados de forma concatenada, mas eles devem ser agregados conforme a aplicação. Uma variação da função genérica apresentada anteriormente no Algoritmo 4.4 é descrita no Algoritmo 4.5. Nesse exemplo, o grupo g_1 ao invés de simplesmente responder duas respostas concatenadas, o grupo aplica uma função de tratamento sobre todas as respostas. Dessa forma ao receber as respostas dos níveis localizados abaixo (g_2 e g_3), aplica-se a função que agrega os valores e retorna ao cliente.

```
1: procedure fnReplies(repContent)
2:     allReplies = ∅
3:     for each key k ∈ repContent do
4:         allReplies.add(k.getValue())
5:     return allReplies
```

Algoritmo 4.5 – Função que agrega respostas

- Cenário 2: Considere mesma aplicação chave-valor descrita no cenário anterior, mas com uma variação no comportamento do cliente: Agora o objetivo é obter quais valores dentro de um escopo de chaves (p. ex. $\{1500..2500\}$) tem seu valor duplicado.

  - Concatenado: Assim como no cenário anterior, cada partição envolvida na consulta retorna os valores para as chaves solicitadas e que estão registradas em sua base. O retorno de cada partição é concatenado em todos os níveis até o cliente. Nesse exemplo, para obtermos os valores duplicados em um escopo que envolve múltiplas partições, necessitamos que cada grupo destinatário, no caso g_2 e g_3, retorne seus valores de cada chave para que então g_1 concatene os resultados e encaminhe para o cliente. Cabe então ao cliente realizar a busca de valores duplicados a partir da resposta obtida.

- Função: Uma modificação no comportamento da função de tratamento de retorno é apresentada no Algoritmo 4.6. Note agora que partes do processamento que anteriormente seriam realizados pelo cliente podem agora ser aplicados em grupos intermediários da árvore ByzCast. Nesse exemplo g_1, ao receber os valores das chaves em seus destinos (g_2 e g_3), agora pode aplicar uma função de tratamento para encontrar os valores duplicados. É natural considerarmos que além do ganho de funcionalidade fornecido pelo modelo, também pode-se considerar o ganho em relação a redução no volume de mensagens que retornam ao cliente. Nesse exemplo em específico ao considerarmos o modelo concatenado, é necessário que todo o conjunto de chaves solicitadas retornassem ao cliente para que o mesmo verificasse os valores duplicados.

```
1: procedure fnReplies(repContent)
2:   duplicates = ∅
3:   seen = ∅
4:   for each key k ∈ repContent do
5:     if k.getValue() ∈ seen then
6:       duplicates.add(k)
7:     else
8:       seen.add(k.getValue())
9:   return duplicates
```

Algoritmo 4.6 – Função que busca valores duplicados

- Cenário 3: Podemos estender a ideia para transladar semântica da aplicação nestes tratamentos. Por exemplo, considere agora que uma base de dados chave-valor é utilizada para o armazenamento de movimentações bancárias de usuários em um banco. Considere ainda que a instituição financeira tem presença em diversos países, então partições estão distribuídas geograficamente (por exemplo, por continente) e que as movimentações de cada cliente são armazenadas em uma partição que esteja mais próxima do mesmo. Um caso de uso simples de uma requisição à nível administrativo do banco pode ser considerado: qual a quantidade de usuários com saldo negativo considerando toda a abrangência da instituição bancária.

  - Concatenado: Cada partição envolvida na consulta retorna a quantidade de usuários com saldo negativo em sua base. O retorno de cada partição é então concatenado em todos os níveis até o retorno ao cliente. Cabe então ao cliente realizar o processamento do retorno com os diferentes resultados obtidos das partições consultadas.
  - Função: O retorno em cada caso seria a soma dos retornos dos descendentes, e da própria partição realizando o tratamento. Considerando o Algoritmo 4.7,

como cada partição envolvida retorna o seu total de usuários com saldo negativo. Perceba que devido ao funcionamento recursivo do algoritmo, agora ao invés do retorno concatenado das respostas de cada uma das partições até o cliente, agora o grupo g_k calcula antes de responder para o nó acima o total de contas com saldo negativo, considerando tanto seus usuários locais (que estão armazenados em sua partição) quanto a(s) resposta(s) de seus grupos filhos contendo a soma de usuários com saldo negativo.

Note que a função que realiza a contabilização é genérica o suficiente para ser empregada em outras aplicações que utilizem de uma semântica similar.

```
1: procedure fnReplies(repContent)
2:     total = 0
3:     for each value v ∈ repContent.getValue() do    {como cada partição retorna seu total}
4:         total = total + v                          {basta somar os valores de cada partição}
5:     return total                                   {e retornar para o cliente}
```

Algoritmo 4.7 – Função que contabiliza respostas

- Cenário 4: Considere uma aplicação que implementa um serviço de rede social que utiliza uma base de dados chave-valor. Partições são distribuídas geograficamente com o objetivo de aproximar as informações aos seus utilizadores, reduzindo a latência e aumentando a performance. Cada partição distribuída armazena um mapa para *users*, *tweets* e *user_connections*. Para gerar a linha de tempo desse usuário uma requisição é naturalmente encaminhada para a partição mais próxima desse usuário. A partição então deve retornar os *tweets* de todas as conexões (*user_connections*) desse usuário contidas tanto na partição do cliente como em todas as partições descendentes.

  - Concatenado: Cada partição envolvida na consulta retorna os valores para as chaves que se encontram em sua base. O retorno de cada partição é concatenado em todos os níveis até o cliente. Cabe ao cliente realizar o processamento do retorno com os resultados de múltiplas partições.
  - Função: Com a possibilidade da aplicação de uma função de tratamento sobre as respostas obtidas, algumas funcionalidades adicionais são elencadas: (i) O retorno ao cliente pode ser ordenado de tal forma que as publicações respeitem uma ordem de publicações. (ii) De posse de todas as repostas das partições, uma função de tratamento pode agregar a partir do conteúdo de todas as respostas obtidas, outros *tweets* que estejam em sua partição local e que sejam relacionados aos interesses deste cliente, funcionando como um sistema de recomendação. (iii) Ainda considerando o item anterior, como resultado da função aplicada as respostas, uma partição pode gerar uma nova requisição para

grupos abaixo da árvore baseado nesse novo conhecimento obtido a partir das respostas em múltiplas partições, funcionando então como um sistema de recomendações que trabalha de forma recursiva sobre um ramo inteiro da árvore ByzCast.

- Cenário 5: Considere uma base de dados utilizada para receber e analisar informações fiscais de cidadãos de um país como o Brasil. Além desta base ser replicada, pois o serviço necessita ser tolerante a falhas, ela também é particionada tendo em vista que, além do país possuir um contingente populacional elevado, ele também ocupa uma grande área territorial com proporções continentais. Nesse exemplo, o estado ser replicado e particionado se aplica pois segmenta o elevado número de requisições dos contribuintes para a partição mais próxima do mesmo, tornando assim o sistema ágil (devido a menor latência fornecida pela partição estar mais próxima do contribuinte) e disponível (devido ao estado da aplicação de qualquer partição estar replicado).

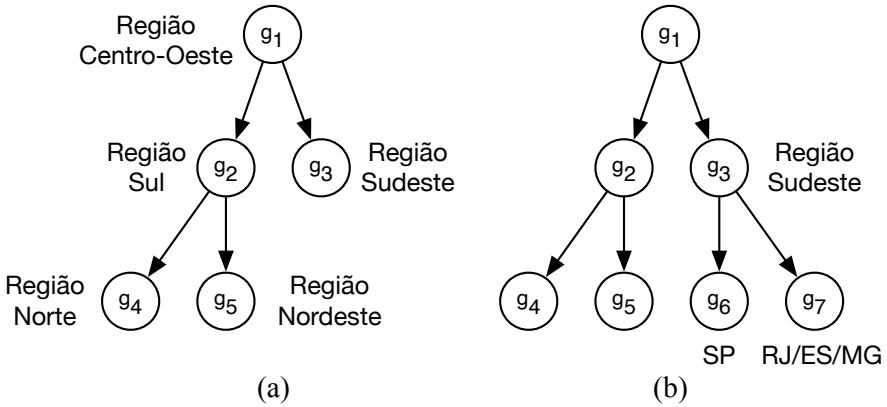

Figura 4.6 – Uma topologia de árvore no ByzCast onde o estado da aplicação é (a) particionado por região ou (b) particionado por região e número de usuários

A Figura 4.6 (a) exibe uma topologia utilizando regiões como critério de particionamento em uma árvore ByzCast e (b) expande essa topologia para atender regiões onde existe uma demanda maior de processamento devido a um maior número de requisições. Note que ao contrário de quando o estado da aplicação é único (seria nesse exemplo se uma única base de dados com todas informações de todos os contribuintes estivessem armazenadas em um único local), ao utilizarmos uma topologia em árvore fornecida pelo ByzCast, é possível prover uma ilusão de base única considerando que uma requisição ao ser encaminhada por g_1 consegue atingir qualquer partição da árvore dependendo da semântica da aplicação utilizada.

Considere então uma consulta administrativa com o objetivo de realizar algum tipo de uma análise fiscal de um contribuinte. A consulta enviada tem como objetivo obter uma listagem de contribuintes das regiões Centro-Oeste (g_1) e Sul (g_2) que possuem algum tipo de pendência administrativa junto ao fisco.

- Concatenado: Nesse exemplo, a requisição ingressa por g_1 e é encaminhada para g_2. Cada partição envolvida na consulta responde seus contribuintes encontrados na partição local conforme o critério enviado para o nível acima. Nesse caso, g_2 responde para g_1 que concatena as informações recebidas de g_2 com as suas e responde para o cliente. Cabe então o cliente realizar algum tipo de processamento nesse retorno com informações concatenadas.
- Função: Utilizando algum tipo de função aplicada sobre as respostas das partições é possível agregar mais informações antes de responder ao próximo nível acima. Algumas possibilidades utilizadas em cenários anteriores também se aplicam nesse contexto: somatório, ordenação, categorização ou uma nova computação sobre os resultados locais e recebidos, gerando uma nova requisição que consegue atingir todo o ramo restante da árvore. Note que uma função aplicada as respostas recebidas não necessita atuar somente no contexto de uma topologia em árvore ByzCast. Perceba que o modelo de uma função de tratamento de respostas apresentado no Algoritmo 4.4 é genérico o suficiente para ser plugado em outros sistemas. Quanto ao exemplo de contribuintes com pendências junto ao fisco, o Algoritmo 4.8 descreve uma função que busca maiores informações através de uma chamada à recurso externo e agrega esse retorno junto à resposta enviada ao nível acima (ou ao cliente).

```
1: procedure fnReplies(repContent)
2:   malhaFina = ∅
3:   for each cpfs cpf ∈ repContent.getValue() do
4:     if upcall_buscaBancoCentral(cpf) then        {verifica Banco Central}
5:       malhaFina.add(cpf)
6:     if upcall_buscaINSS(cpf) then                {verifica INSS}
7:       malhaFina.add(cpf)
8:     ...
9:   return malhaFina
```

Algoritmo 4.8 – Função que verifica pendências fiscais

Note que parte da lógica da aplicação que anteriormente era considerada uma competência do cliente, com a utilização desse modelo, essa competência pode ser movida para dentro da RME particionada se a semântica da aplicação possibilitar.

- Cenário 6: Considere uma aplicação utilizada para agregar e divulgar notícias na Internet. Considerando o cenário atual quanto a divulgação de notícias fraudulentas e de violações de privacidade, pode-se aplicar uma função de validação sobre as informações contidas no retorno da consulta nas partições envolvidas.

- Concatenado: Cada partição envolvida na consulta retorna as notícias com base na consulta enviada. O retorno de cada partição é concatenado em todos os níveis até o cliente. Cabe ao cliente realizar o processamento do retorno com os resultados de múltiplas partições.
- Função: Além de agrupar ou reordenar as notícias recebidas, uma função pode ser aplicada sobre as respostas, aplicando algum tipo de processamento externo sobre as mesmas.

```
1: procedure fnReplies(repContent)
2:   notFakeNews = ∅
3:   for each news n ∈ repContent.getValue() do
4:     if upcall_validaNoticia(n) then
5:       notFakeNews.add(n)
6:   return notFakeNews
```

Algoritmo 4.9 – Função que valida notícias

Perceba que funções aplicadas às respostas podem ser úteis inclusive na geração de gatilhos. Como por exemplo em sistemas de grandes varejistas com múltiplos centros de distribuição que analisa os motivos por abandono de carrinho, uma função de retorno pode detectar, baseado em respostas que consultam o percentual de abandono de um mesmo produto nos níveis abaixo na árvore (que representam outras regiões desse mesmo comércio online), os motivos desse cliente não ter realizado a compra, como por exemplo um prazo de entrega diferente entre as regiões e assim acionar um gatilho para redistribuição de estoque entre os centros de distribuição.

Note que outros cenários são possíveis de serem elencados para descrever a aplicabilidade dos modelos que utilizam o estado da aplicação particionado considerando uma árvore de sobreposição hierárquica do ByzCast.

4.3 Avaliação de desempenho

Nesta seção, apresentamos um protótipo que implementa o algoritmo assíncrono do ByzCast. Além disso, discutimos os resultados da nossa avaliação experimental.

4.3.1 Implementação

Tanto a versão síncrona quanto a versão assíncrona do ByzCast foram implementadas utilizando a biblioteca BFT-SMART como base. O BFT-SMART é uma biblioteca amplamente utilizada para implementar algoritmos tolerantes a falhas bizantinas por meio da

técnica de Replicação Máquinas de Estados [6]. Além disso, ela é comumente empregada em projetos acadêmicos e em sistemas *blockchain* [12, 61].

Na versão assíncrona do ByzCast, cada grupo corresponde a uma RME tolerante a falhas bizantinas. Cada grupo é composto por quatro réplicas, que podem tolerar a falha de uma única réplica. Além disso, cada réplica se conecta a todos os grupos disponíveis no próximo nível da árvore.

Duas topologias diferentes de árvore foram avaliadas neste estudo: uma árvore com três níveis, conforme apresentado na Figura 4.7, e uma árvore com cinco níveis, conforme apresentado na Figura 4.9. Diferentemente do que foi apresentado em [15], que utiliza grupos auxiliares para ordenar as mensagens globais, no ByzCast assíncrono cada grupo pode desempenhar o papel de tanto um nó destinatário quanto um nó intermediário na árvore.

Cada cliente encaminha suas mensagens para todas as réplicas do grupo de menor ancestral comum da mensagem. Adicionalmente, todos os clientes operam em um laço fechado, enviando uma nova mensagem somente após receber a resposta da mensagem anterior. Tanto a versão síncrona quanto assíncrona do ByzCast foram implementadas em Java e seu código-fonte está disponível publicamente[3].

4.3.2 Ambiente e configuração

Experimentos foram executados em uma rede local de computadores (LAN) com o objetivo de obter um ambiente controlado. Neste ambiente, cada réplica utilizou um servidor composto por dois processadores Intel Xeon L5420 que somados totalizam 8 núcleos executando à 2.5GHz, 8GB de memória RAM, discos SATA de estado sólido (SSD) e uma rede ethernet operando à 1Gbps. Clientes executam em um servidor com processador AMD Opteron 2122 com velocidade de 2Ghz, 4GB de memória RAM, discos SATA de estado sólido (SSD) e uma interface de rede ethernet de 1Gpbs. Todos os servidores foram configurados com o sistema operacional GNU/Linux Ubuntu 18.04 LTS 64bits. O tempo total entre o envio e o recebimento de uma requisição (RTT) entre os servidores é de aproximadamente de 0,1ms.

O estado de cada réplica no ByzCast é mantido na memória principal e as réplicas só respondem aos clientes após a execução do comando em todos os níveis nos quais a mensagem foi endereçada. Para avaliar o desempenho do sistema, foram realizados experimentos com mensagens de tamanho fixo de 64 bytes. Cada rodada te testes teve uma duração de 120 segundos. Diversas topologias de árvore com até 5 níveis de profundidade e quantidades variadas de grupos foram avaliadas, sendo 5 o número máximo de grupos por topologia analisado devido à limitação na quantidade de servidores do clus-

[3] https://github.com/tarcisiocjr/byzcast

ter de servidores. Cada grupo é composto por 4 processos, que executam em servidores distintos no cluster.

4.3.3 Vazão e latência das variantes apresentadas

Este estudo inicia analisando os resultados obtidos a partir da avaliação de duas implementações do ByzCast: a versão síncrona, inicialmente apresentada em [15], e a versão assíncrona, apresentada no Algoritmo 4.2. Ambas as versões foram submetidas às mesmas condições de carga de trabalho, topologia e quantidade de grupos nos testes realizados.

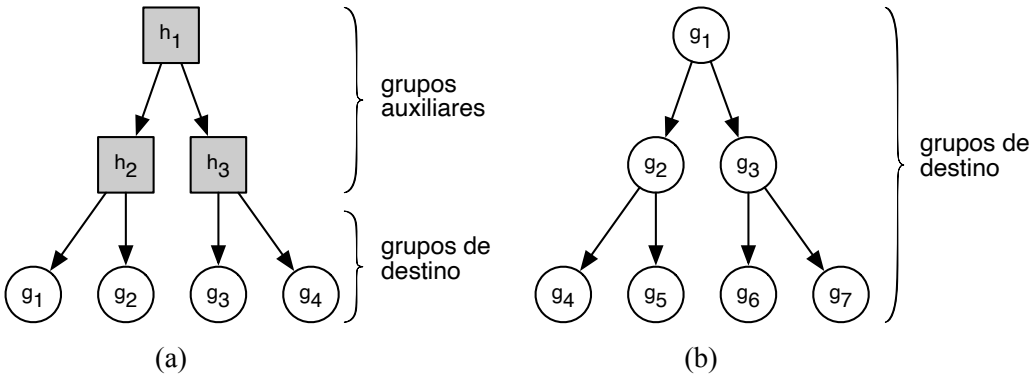

Figura 4.7 – Uma mesma topologia de árvore com três níveis do ByzCast é apresentada, sendo (a) versão síncrona que utiliza três grupos auxiliares (h_x) para ordenação e quatro grupos possíveis de serem destinatários. (b) versão assíncrona sendo que cada nó da árvore é um grupo destino possível de uma mensagem.

Um dos fatores limitadores da versão síncrona do ByzCast é a sua característica relacionada ao funcionamento de grupos auxiliares que foram projetados com a função de ordenar e encaminhar mensagens para grupos posicionados abaixo na árvore. Na Figura 4.7 (a), pode-se observar a topologia utilizada na versão síncrona do protocolo. A presença desses grupos auxiliares pode se tornar um fator limitador e ineficiente na prática devido a necessidade de empregar recursos computacionais em nós que são utilizados somente para a ordenação de mensagens. Por exemplo, considere o caso onde mensagens são destinadas a grupos folha da árvore, mas em ramos distintos. Para encaminhar uma mensagem endereçada para $msg.dst = g_2, g_3$ é necessária uma infraestrutura de 12 réplicas (h_1, h_2, h_3) somente para suportar a funcionalidade de ordenação e encaminhamento das mensagens dos grupos auxiliares.

A versão assíncrona do protocolo ByzCast propõe uma mudança significativa em relação à funcionalidade de cada nó em uma árvore ByzCast. Conforme ilustrado na Figura 4.7 (b), a nova versão remove a noção de grupos auxiliares e permite que cada nó atue como intermediário ou destinatário de uma mensagem. Com a utilização do algo-

ritmo que implementa a versão assíncrona, o estado pode ser particionado sem a sobreposição de partições pela quantidade de nós da árvore ByzCast.

Os resultados apresentados a seguir têm como objetivo avaliar a vazão e a latência dos cenários propostos anteriormente em 4.2, em relação ao tratamento e retorno de uma mensagem em uma RME particionada utilizando multicast atômico.

Análise do Modelo B - Concatenação de Respostas

Os resultados obtidos no experimento a seguir analisam a vazão e a latência considerando uma árvore com até 5 níveis conforme Figura 4.8. Nessa topologia, cada nível da árvore foi avaliada desde o grupo de ingresso da mensagem até o destino da mesma. Por exemplo, para avaliar a vazão e a latência considerando três níveis de profundidade, um cliente define o último nó da árvore ($msg.dst = g_5$) como destino da mensagem e submete a mensagem ao grupo auxiliar h_3, para os experimentos da versão síncrona, ou ao grupo g_3, para os experimentos da versão assíncrona.

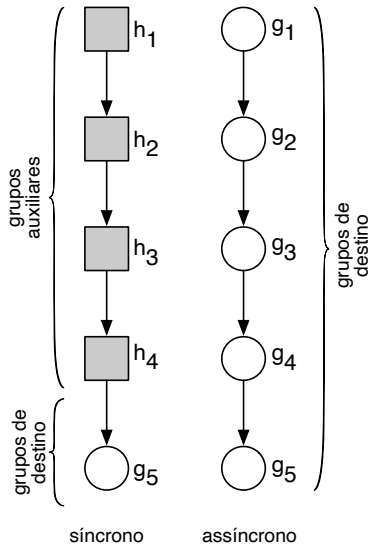

Figura 4.8 – Topologia em árvore ByzCast com até 5 níveis

Na versão síncrona do ByzCast, cada nó concatena as respostas antes de responder ao grupo imediatamente acima na árvore (ou ao cliente). Esse funcionamento onde requisições são concatenadas e que é definido neste trabalho como um Modelo na Seção 4.2.2, é a forma clássica de operação de uma RME aplicada no contexto de multipartições. Devido a essa característica de tratamento das respostas, esses resultados refletem a única comparação direta realizada entre as versões síncrona e assíncrona.

Os resultados apresentados na Figura 4.9 foram obtidos ao avaliarmos o comportamento do ByzCast em diferentes níveis de profundidade da árvore, sem a utilização da técnica de agrupamento de mensagens (*batch*). A aplicação utilizada foi uma base de dados chave-valor particionada pela quantidade de grupos destinatários possíveis da

árvore. Uma mensagem enviada pelo cliente refere-se a uma operação de escrita com uma chave e valor gerados de forma aleatórias. O tamanho de cada mensagem enviada tem 64 bytes.

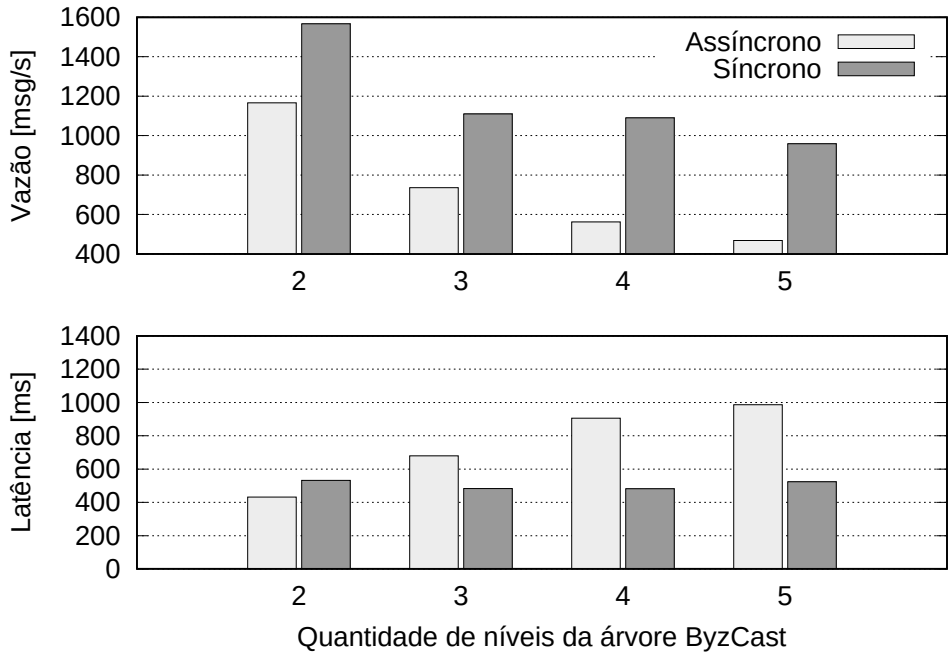

Figura 4.9 – Vazão e latência sem *batch* de mensagens em uma árvore ByzCast com até 5 níveis

Os resultados indicam que a versão síncrona do ByzCast apresentou desempenho superior em todos os cenários avaliados, com uma redução de 15% na taxa de vazão entre o terceiro e o quinto nível da árvore. A versão assíncrona apresentou uma diferença significativa na taxa de vazão em todos os níveis avaliados em relação à versão síncrona, tendo a vazão reduzida pela metade à medida que a altura da árvore avaliada aumentava.

Além disso, a latência na versão síncrona manteve-se estável com cerca de $400ms$ em todos os níveis, enquanto na versão assíncrona foi acrescentado em média $200ms$ na taxa de latência a cada nível avaliado da árvore.

Dessa forma, optou-se por implementar uma otimização na versão assíncrona do algoritmo utilizando a técnica de agrupamento de requisições (*batch*). Essa técnica tem o potencial de melhorar a eficiência e reduzir a sobrecarga de comunicação em sistemas distribuídos.

A análise dos resultados apresentados na Figura 4.10 revelou que, mesmo com a implementação da técnica de agrupamento de mensagens no algoritmo assíncrono, ele ainda teve um desempenho inferior em comparação com a versão síncrona em todos os níveis avaliados da árvore ByzCast. A topologia em árvore usada neste experimento é a mesma da Figura 4.7.

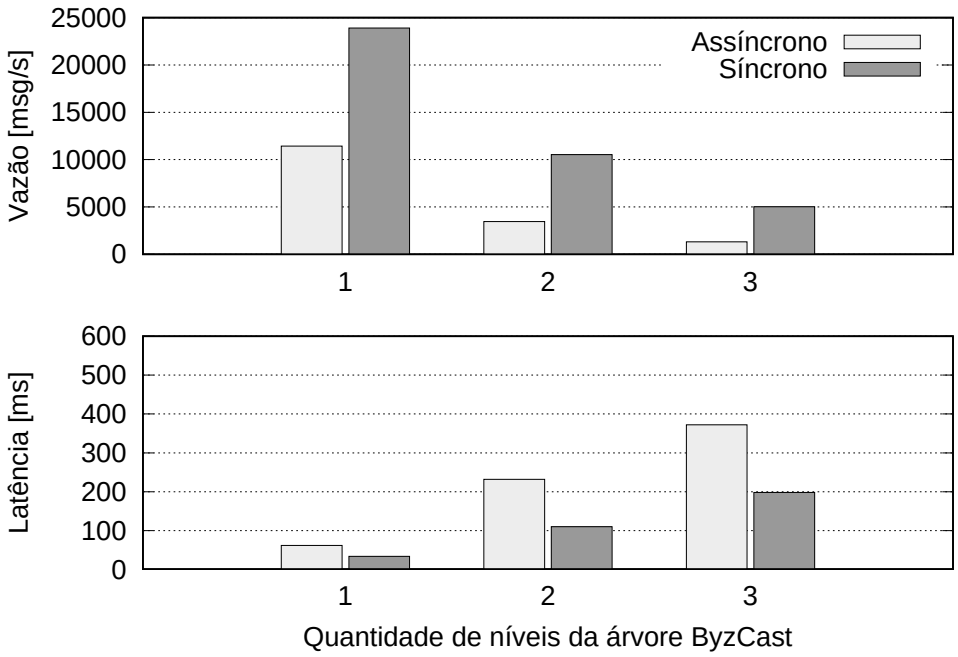

Figura 4.10 – Vazão e latência com *batch* de mensagens em uma árvore ByzCast com até 3 níveis nas versões síncrona e assíncrona

É importante destacar que, ao analisarmos tanto a vazão quanto a latência em uma árvore de um único nível (as duas barras à esquerda da Figura 4.10), nenhuma mensagem específica desse experimento estava sendo repassada para outros níveis da árvore.

Os motivos ainda não estão totalmente compreendidos, mas uma possível explicação é que a introdução de novas estruturas de controle necessárias para o funcionamento assíncrono do algoritmo pode ter aumentado significativamente o custo computacional de processamento de cada mensagem, o que prejudicou o desempenho geral da versão assíncrona. Além disso, observou-se que, em média, cada requisição na versão assíncrona leva aproximadamente $\sim 4ms$ a mais para ser respondida ao cliente em comparação com a versão síncrona. Esse resultado levanta questões sobre a otimização do algoritmo assíncrono e será discutido com mais detalhes na seção de trabalhos futuros.

Consideramos esse modelo de execução onde as respostas são concatenadas como ponto de referência para o comparativo descrito a seguir.

Análise do Modelo C - Função Aplicada às Respostas

Os experimentos conduzidos no Modelo C tiveram como objetivo avaliar a aplicação de uma função às respostas antes das mesmas serem encaminhadas ao grupo acima na árvore (ou ao cliente), agregando uma funcionalidade adicional que, até o presente momento, não foi abordada na literatura.

Em virtude da impossibilidade de aplicar tal função às respostas em cada nível da árvore na versão síncrona do ByzCast, utilizou-se como referência os resultados obtidos no modelo concatenado e assíncrono apresentado anteriormente. Assim, nesta análise buscou-se mensurar o impacto na vazão e latência ao aplicarmos uma função sobre as respostas em cada nível.

Este experimento tem como objetivo avaliar a vazão e latência em cada nível da árvore. Uma mensagem enviada pelo cliente se traduz em uma operação que verifica valores duplicados em uma base de dados chave-valor particionada. Antes do início dos experimentos, a base de dados é pré-populada de forma determinística em todos os grupos da árvore, garantindo que todos os experimentos sejam realizados com uma população constante e homogênea.

Os resultados apresentados comparam a operação que verifica valores duplicados em dois modelos distintos. Utilizado como referência, no modelo concatenado a resposta com os valores duplicados de cada grupo destino da mensagem é concatena até o seu retorno ao cliente. Por outro lado, no modelo onde uma função é aplicada sobre a resposta, além da verificação de valores duplicados localmente, cada um dos valores duplicados recebidos anteriormente são verificados se são valores duplos na base local, antes de responder para o nível acima. Como resposta o cliente recebe os valores duplicados vistos apenas uma vez entre os grupos destinatários da mensagem.

Os resultados apresentados na Figura 4.11, que utiliza uma topologia em árvore de sobreposição conforme Figura 4.7 (b), confirmam o impacto na vazão e latência ao aplicarmos uma função sobre as respostas. Note que o tempo de processamento realizado pelo cliente na versão que concatena os resultados não é considerado no cálculo dos resultados. Nesses experimentos, o processamento adicional causado pela função aplicada sobre as respostas impactou tanto na vazão quanto na latência com uma topologia de árvore de 2 e 3 níveis.

Analisando os resultados apresentados em cada nível, tanto o modelo que concatena as respostas quando o modelo que aplica uma função sobre as respostas apresentaram o mesmo desempenho considerando um único nível da topologia. Ou seja, no cenário onde as mensagens encaminhadas pelo cliente são submetidas para g_1 e possuem $msg.dst = \{g_1\}$ também como destino, valores de vazão e latência em ambos modelos são iguais. Por mais que esse comportamento seja esperado tendo em vista que ambos modelos só se aplicam quando mais de um grupo destinatário é envolvido, esse resultado define um ponto inicial para os próximos comparativos. Como referência, os resultados apresentados em ambos modelos nesse cenário para vazão e latência foram respectivamente de $\sim$11K msg/s e $\sim$60ms.

Considerando uma topologia em árvore com dois níveis, um cliente agora submete suas requisições para g_1 tendo como $msg.dst = \{g_2, g_3\}$. O tempo de processamento adicional em g_1 causado pela função aplicada nas respostas recebidas de g_2 e g_3 é de

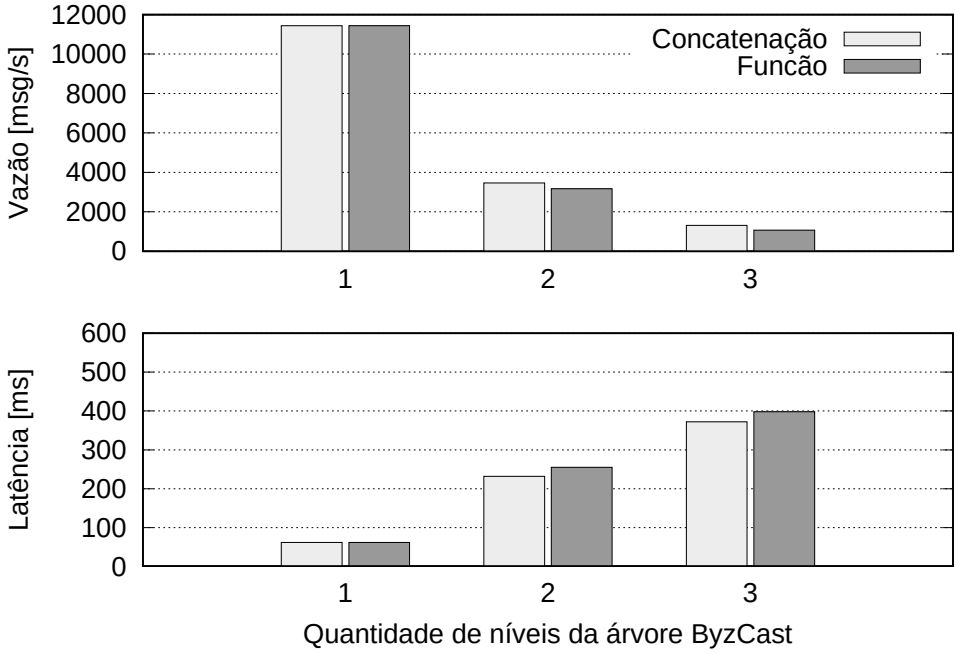

Figura 4.11 – Vazão e latência com *batch* de mensagens em uma árvore ByzCast com até 3 níveis.

∼8%. Enquanto a latência média no modelo que concatenada foi de ∼230ms, a função aplicada foi de ∼250ms. Quanto a vazão, observou-se uma redução de ∼3500 msg/s no modelo concatenada para ∼3100 msg/s no modelo que aplica a função.

A topologia com três níveis considera mensagens com $msg.dst = \{g_5, g_6\}$ de destino. Requisições remetidas pelos clientes são submetidas em g_1, que encaminha para g_2 e g_3. Do nível intermediário da árvore, g_2 remete para g_5 e o grupo g_3 remete para g_6. A resposta segue o caminho inverso até o recebimento da resposta pelo cliente. Enquanto a vazão na versão concatenada foi de ∼1300 msg/s, ao aplicarmos a função em cada um dos níveis intermedíarios, g_2, g_3 e g_1, a vazão observada caiu para ∼ 1100 msg/s, ou ∼15%. Compartamento siminar quanto a latência, subindo de ∼370 no modelo concatenado para ∼400 ms no modelo que aplica a função nas respostas recebidas.

É importante destacar que a queda de vazão e latência é um aspecto comparativo e não conclusivo, visto que os cenários apresentados oferecem um valor agregado diferente ao cliente. Além disso, é importante ressaltar que o uso de uma função sobre as respostas pode agregar valor ao sistema, permitindo que o cliente obtenha informações mais relevantes ou processadas de acordo com suas necessidades. No entanto, é preciso considerar o custo computacional que essa funcionalidade extra pode vir a gerar, podendo afetar o desempenho geral do sistema em determinados tipos de aplicações.

4.4 Discussão

Multicast atômico é uma abstração fundamental para suportar aplicações distribuídas, especialmente em ambientes que necessitem de garantias de alta disponibilidade e escalabilidade. Nesse contexto, ByzCast é um protocolo de multicast atômico tolerante a falhas bizantinas parcialmente genuíno que tem como requisito para o seu funcionamento ser implementado considerando uma topologia de árvore de sobreposição hierárquica. Devido a estas características, ByzCast foi candidato ideal para a definição e avaliação dos modelos de execução propostos. Os modelos apresentados nesse capítulo possibilitam que aplicações distribuídas possam ser projetadas considerando uma nova perspectiva de funcionalidade de acordo com cada modelo.

Ao aplicarmos uma função de tratamento sobre as respostas, move-se a lógica da aplicação do cliente para dentro da RME e dependendo do tipo de aplicação, isso pode ser relevante. Embora a construção do algoritmo assíncrono não tenha fornecido ganhos em termos de vazão e latência, o desenvolvimento do mesmo possibilitou a extinção de grupos auxiliares da árvore ByzCast. Além disso, o mesmo algoritmo viabilizou a implementação dos modelos propostos, permitindo que se apresentassem seus ganhos considerando suas funcionalidades.

5. TRABALHOS RELACIONADOS

Neste capítulo, revisamos algumas abordagens existentes utilizadas no contexto de escalabilidade da técnica de Replicação Máquina de Estados. Primeiramente, nos concentramos nas abordagens existentes para RME Paralela e, em seguida, consideramos as técnicas utilizadas no contexto de RME Particionada.

5.1 RME Paralela

Como observado por Schneider [60], comandos independentes podem ser executados concorrentemente em uma RME. Trabalhos anteriores mostraram que muitas cargas de trabalho são dominadas por comandos independentes, justificando estratégias para a execução paralela de comandos [8, 11, 38, 47, 48, 49]. Organizamos as principais abordagens de RME paralelo em quatro classes, descritas a seguir.

Uma abordagem que chamamos de *(*i) escalonamento tardio trata desses aspectos exclusivamente no lado da réplica. Uma ordem total de comandos é entregue nas réplicas que então, ao invés de executá-los sequencialmente, detectam conflitos entre comandos pendentes para escalonar os independentes em paralelo. Em CBASE [38], as réplicas são aperfeiçoadas através de um escalonador determinístico. Os comandos decididos são adicionados em um grafo de dependência acíclico dirigido. Durante a inserção do comando no grafo, dependências com comandos inseridos anteriormente são detectadas e incluídas junto ao grafo, como arestas. Comandos sem dependências são processados por um conjunto de threads. A execução de um comando leva à exclusão de seu nó do grafo, levando a remoção de suas dependências.

O grafo de dependência introduz contenção quando vários núcleos são utilizados. Portanto, em [51] os autores propõem mecanismos mais eficientes para detectar dependências, porém ao preço de falsos positivos, introduzindo um trade-off entre sobrecarga no escalonamento e concorrência. O mesmo problema é abordado de uma perspectiva diferente em [25]. Aqui são propostos algoritmos e estruturas para um grafo de dependência livre de bloqueios, reduzindo consideravelmente a contenção.

Em [3], uma técnica chamada *(*ii) escalonamento antecipado é proposta para evitar a contenção em uma estrutura de dados sincronizada (por exemplo, GAD). A ideia é que as decisões de escalonamento nas réplicas sejam as mais rápidas possíveis, sendo a classificação dos comandos realizada por classes pelos clientes. As réplicas têm um mapeamento conhecido de classes para threads, derivado através de uma definição de conflitos entre classes de comando e a carga de trabalho esperada em cada classe. Nas réplicas, de acordo com o mapeamento já decidido, as informações de classe são utiliza-

das para encaminhar o respectivo comando para a fila de entrada de uma ou mais threads. O modelo de execução e o mapeamento de threads garantem a execução sequencial de comandos conflitantes. Essa técnica pode levar a um aumento de vazão, mas o desempenho pode ser penalizado se a carga de trabalho se desviar do mapeamento esperado de classes e threads.

P-SMR [48] também evita um paralelizador ou um escalonador central. Isso é obtido através do mapeamento de comandos para diferentes grupos multicast nos clientes. Os comandos não conflitantes são propagados por meio de diferentes grupos de multicast que ordenam parcialmente os comandos entre as réplicas. Os comandos são entregues por múltiplas *threads* de trabalho de acordo com o grupo multicast. Com base nas informações de comando que são específicas da aplicação, essa abordagem impõe uma escolha de grupo de destino no lado do cliente. Comandos não conflitantes podem ser enviados para grupos distintos, enquanto comandos conflitantes são enviados para o(s) mesmo(s) grupo(s). No lado da réplica, cada *thread* de trabalho é associada a um grupo multicast e processa os comandos à medida que eles chegam.

Ao invés de lidar com conflitos antes da execução de comandos, como relatado anteriormente, técnicas *(iii) otimistas* introduzem uma abordagem a posteriori. Em Eve [37], réplicas executam otimistamente lotes de comandos em paralelo a medida que eles chegam, e após a execução é verificado se a consistência foi violada através do acordo entre as réplicas. Em caso de violação de consistência, as réplicas desfazem a execução e reexecutam os comandos de forma sequencial. Embora se espere que a reexecução seja rara, ela impacta no desempenho. Em Storyboard [35], um mecanismo prevê uma mesma sequência ordenada de bloqueios entre as réplicas. Comandos podem ser executados em paralelo quando as previsões estão corretas. Caso contrário, as réplicas interrompem o processamento dos comandos e usam a etapa de acordo para recalcular a sequência de execução dos comandos. Em [49], P-SMR [48] é estendido com execução otimista com objetivo de aumentar a execução concorrente entre comandos. Ao invés de assumir de forma conservadora que dois comandos conflitam quando não há informação suficiente nos clientes, é assumido otimistamente que os comandos não conflitam. Se ocorrer um conflito (detectado durante a execução), os comandos envolvidos são reexecutados de forma sequencial considerando os conflitos entre os comandos.

Algumas técnicas implementam mecanismos de (iv) coordenação em tempo de execução (*runtime coordination*) com objetivo de garantir a execução determinística nas réplicas. Rex [30] é um sistema que utiliza a estratégia de "executar-concordar-prosseguir". Nesse sistema um único servidor denominado primário recebe as solicitações e as processa em paralelo em diferentes *threads*. Enquanto executa, o primário armazena um rastro de dependências entre as solicitações com base nas variáveis compartilhadas acessadas por cada *thread*. Periodicamente é proposto um corte consistente nesse rastro em acordo com todas as réplicas. Com isso, outras réplicas recebem esses rastros e repro-

duzem a execução respeitando a ordem parcial dos comandos, seguindo a causalidade das operações de *lock* e *unlock*. O não determinismo devido à concorrência é resolvido seguindo as decisões geradas pelo servidor primário. No entanto, operações de sincronização de rastros entre réplicas podem resultar em alto consumo de rede e sobrecarga de processamento.

CRANE [19] utiliza outra estratégia para resolver o não-determinismo durante a execução de comandos. Ele mantém as réplicas sincronizadas usando vários mecanismos em tempo de execução. Através da interceptação de chamadas de socket da interface de rede, consegue-se implementar o acordo (usando uma implementação do algoritmo de consenso Paxos) sobre sequências de chamadas entre as demais réplicas. A sincronização de *threads* utiliza *multi-threading* determinístico [55]. Além disso, CRANE introduz uma técnica de *time bubbling* para impor tempos lógicos e determinísticos para rajadas de solicitações. Entretanto, a sobrecarga no tempo de execução é significativa pois além da necessidade de acordo em cada evento de socket, o sistema DMT incorre em uma sobrecarga de 12,7%.

Entre as quatro classes descritas acima, a arquitetura apresentada anteriormente está mais próxima das abordagens de escalonamento tardio (i) pois decisões de escalonamento são realizadas do lado do servidor e antes da execução. Mesmo considerando o esforço considerável na literatura com objetivo de explorar o paralelismo na técnica de Replicação Máquina de Estados, até onde se tem conhecimento, PePaxos foi a primeira abordagem apresentada a utilizar as informações de conflitos gerada pelo consenso generalizado com objetivo de aumentar a concorrência na execução de comandos.

5.2 RME Particionada

A busca por escalabilidade na técnica de RME frequentemente envolve o particionamento do estado, sendo que lidar eficientemente com operações que envolvem múltiplas partições é um dos principais desafios enfrentados. Multicast atômico é uma abstração fundamental comumente utilizada na construção de protocolos que particionam o estado em uma RME.

Diversos algoritmos de multicast foram propostos na literatura e comumente são divididos em pelo menos três categorias [21]: (i) baseados em *timestamp* [27, 58, 17, 59]; (ii) baseados em *rodadas* [59]; e (iii) baseados em *topologia* [50, 1, 5, 7].

(i) Denominado algoritmo *Skeen* [9], foi um dos primeiros protocolos de multicast atômico genuíno, mas desenvolvido para executar em cenários livre de falhas. Nesse protocolo, cada processo implementa um relógio lógico [40] atribuindo timestamps nas mensagens. Para decidir sobre um timestamp final de uma determinada mensagem, cada processo no conjunto de destinatários define localmente um timestamp e dissemina esse

timestamp escolhido entre os demais processos até que, de forma determinística, um único timestamp entre todos que foram propostos seja aceito por todos os processos. Esse protocolo é escalável pois apenas os destinatários da mensagem estão envolvidos. Em [27] é descrito o primeiro protocolo tolerante a falhas que estende o algoritmo original de *Skeen* [9]. Nesse trabalho, cada grupo atua como um processo independente do protocolo de *Skeen* [9] e utiliza o consenso para decidir entre propostas (de *timestamp*) com o objetivo de avançar o relógio lógico. FastCast [17] utiliza um modelo de execução otimista que consegue entregar mensagens destinadas para múltiplos grupos em quatro passos de comunicação.

Outra variação do protocolo de Skeen conhecida é apresentada em Tempo [24]. Definido como um protocolo *leaderless*, nessa abordagem cada comando possui uma réplica que se encarrega pela sua execução. A réplica coordenadora se comunica com as réplicas de destino para replicar o comando e concordar sobre seu *timestamp*. Dessa forma, as réplicas trocam informações sobre cada *timestamp* atribuído e só executam o comando quando todos os comandos com *timestamp* menores forem executados.

(ii) Em algoritmos baseados em rodadas, processos executam uma sequência ilimitada de rodadas e concordam com as mensagens entregues ao final de cada uma destas rodadas. Um algoritmo não genuíno de multicast atômico baseado em rodadas é apresentado em [59]. Diferentemente do ByzCast, esse algoritmo pode penalizar mensagens locais devido à lentidão de mensagens globais.

(iii) Algoritmos baseados em topologia propagam e ordenam mensagens e garantem as propriedades do multicast atômico a partir da utilização dessa topologia. Um algoritmo de multicast atômico dessa categoria é proposto em [22], onde o conjunto de grupos de destino de uma determinada mensagem são encadeadas. Dessa forma, o primeiro grupo executa o consenso para decidir sobre a entrega da mensagem e repassa para o próximo grupo, e assim por diante. A complexidade de tempo desse algoritmo depende da quantidade de grupos destinatários dessa mensagem.

Multi-Ring Paxos [50], Spread [1, 5], e Ridge [7] são protocolos de multicast atômico não-genuínos baseados em anel que para entregar uma mensagem é necessária a comunicação de processos fora dos grupos de destino da mensagem.

Baseado no Multi-Ring Paxos [50], em S-SMR[8] o serviço é particionado entre partições que se utilizam da primitiva de multicast atômico para ordenar de forma consistente os comandos entre todas as partições envolvidas. O S-SMR assegura a linearizabilidade do sistema por meio de uma propriedade conhecida como *execução atômica*. Durante a operação, todas as partições envolvidas na execução se comunicam entre si, garantindo que o cliente receba sua resposta somente após a execução da operação em todas as partições envolvidas.

6. CONCLUSÃO

Replicação Máquina de Estados é uma técnica bem estabelecida para fornecer um serviço disponível. Ao replicar um serviço entre vários servidores, clientes têm a garantia de que mesmo na ocorrência de falha de uma réplica, o serviço ainda estará disponível. Ocorre que independente do número de réplicas que o serviço possui, o desempenho do sistema permanece o mesmo devido ao seu modelo básico de funcionamento.

Para aumentarmos a escalabilidade da técnica RME, diferentes estratégias foram propostas e esta tese apresenta contribuições em duas linhas distintas. A primeira explora o aumento do paralelismo na execução de comandos independentes e a segunda através do particionamento do estado da técnica de RME aliado a utilização de um protocolo de multicast atômico.

Nas seções seguintes faremos um breve resumo das contribuições desta tese e apresentaremos algumas direções para trabalhos futuros.

6.1 Contribuições

Nesta tese apresentamos três contribuições relacionadas a escalabilidade da técnica de RME: (i) o aumento do paralelismo na execução de comandos independentes; (ii) uma versão assíncrona de funcionamento do ByzCast; e (iii) a definição de modelos de execução em uma RME particionada.

O aumento do paralelismo na execução de comandos independentes foi resultado do uso de informações de conflito fornecido pelo consenso generalizado. Por meio da avaliação experimental realizada, foi possível constatar que a abordagem adotada proporcionou ganhos consideráveis de desempenho à medida que a independência entre comandos e o custo computacional aumentaram.

Uma versão assíncrona de funcionamento do ByzCast foi implementada e comparada a sua versão síncrona. Embora a construção do algoritmo assíncrono não tenha apresentado melhorias em termos de vazão e latência, o algoritmo permitiu a eliminação dos grupos auxiliares da árvore ByzCast.

A definição de modelos de execução em uma RME particionada utilizando uma topologia de uma árvore de sobreposição do ByzCast, possibilitou que aplicações distribuídas múltiplas partições sejam projetadas considerando diferentes perspectivas de funcionalidade de acordo com cada modelo. Ao aplicarmos uma função de tratamento sobre as respostas, parte da lógica da aplicação do cliente é deslocada para dentro da RME e dependendo do tipo de aplicação isso pode ser relevante.

6.2 Trabalhos Futuros

PePaxos apresenta uma abordagem para escalonar a execução paralela de comandos independentes em ambientes de Replicação Máquina de Estados por meio do uso de informações de conflito geradas pelo consenso generalizado. O processo para a representação de conflitos é realizado por meio da construção de um grafo de dependência, em que uma instância independente é representada por um grafo que contém uma única instância (a própria instância). A identificação de Componentes Fortemente Conexos (CFCs) em um grafo de dependência é realizada por meio do algoritmo de *Tarjan*, que possui natureza sequencial. No entanto, essa característica sequencial do algoritmo pode indicar um ponto de contenção e a investigação de ganhos adicionais através da paralelização da geração de grafos de dependência uma possibilidade a ser estudada.

ByzCast é um protocolo de multicast atômico parcialmente genuíno construído sobre múltiplas instâncias de um protocolo de difusão atômica que foi utilizado neste trabalho para contemplar tanto multicast atômico como a definição de diferentes modelos de execução. Por mais que o algoritmo assíncrono do protocolo apresentado tenha possibilitado a eliminação dos grupos auxiliares em uma árvore ByzCast, o algoritmo não forneceu qualquer ganho em termos de vazão ou latência frente a versão síncrona. Estudar os custos computacionais das novas estruturas utilizadas no algoritmo assíncrono podem indicar os motivos dessa queda de desempenho. Diferente da versão onde os controles necessários são simplificados devido a sua natureza síncrona, o uso de agrupamento de mensagens utilizado era fornecido pela própria plataforma de difusão atômica utilizada, o BFT-SMART. Na versão assíncrona, mesmo utilizando a mesma plataforma de difusão atômica, além de uma implementação própria do agrupamento de mensagens para o uso assíncrono, também foi necessário o emprego de controles adicionais para o tratamento posterior das respostas enviadas. Além disso, BFT-SMART é um protocolo de difusão atômica utilizado no contexto de multicast atômico. Um estudo pode avaliar a real capacidade do protocolo BFT-SMART de gerenciar uma grande quantidade de sessões assíncronas criadas em cada nível da árvore ByzCast.

Os modelos de execução apresentados utilizam-se da topologia em árvore de sobreposição do protocolo ByzCast e podem aprimorar as funcionalidades de uma RME particionada. Ao invés de concatenar as respostas recebidas em cada nível da árvore, uma função pode ser aplicada sobre o conjunto completo de respostas em cada nível, transferindo parte da lógica da aplicação para dentro da RME. Embora os modelos apresentados neste trabalho tenham abordado o tratamento das respostas em cada nível da árvore, estudos complementares podem avaliar a existência de novos modelos que considerem o processamento das requisições à medida que estas descem pela árvore.

REFERÊNCIAS BIBLIOGRÁFICAS

[1] Agarwal, D. A.; Moser, L. E.; Melliar-Smith, P. M.; Budhia, R. K. “The totem multiple-ring ordering and topology maintenance protocol”, *ACM Transactions on Computer Systems*, vol. 16–2, May 1998, pp. 93–132.

[2] Alchieri, E.; Dotti, F.; Mendizabal, O. M.; Pedone, F. “Reconfiguring parallel state machine replication”. In: Proceedings of the International Symposium on Reliable Distributed Systems, 2017, pp. 104–113.

[3] Alchieri, E.; Dotti, F.; Pedone, F. “Early scheduling in parallel state machine replica”. In: Proceedings of the ACM Symposium on Cloud Computing, 2018, pp. 82–94.

[4] Arun, B.; Peluso, S.; Palmieri, R.; Losa, G.; Ravindran, B. “Speeding up consensus by chasing fast decisions”. In: Proceedings of the International Conference on Dependable Systems and Networks, 2017, pp. 49–60.

[5] Babay, A.; Amir, Y. “Fast total ordering for modern data centers”. In: Proceedings of the International Conference on Distributed Computing Systems, 2016, pp. 669–679.

[6] Bessani, A.; Sousa, J.; Alchieri, E. E. “State Machine Replication for the Masses with BFT-SMART”. In: Proceedings of the International Conference on Dependable Systems and Networks, 2014, pp. 355–362.

[7] Bezerra, C. E.; Cason, D.; Pedone, F. “Ridge: high-throughput, low-latency atomic multicast”. In: Proceedings of the Symposium on Reliable Distributed Systems, 2015, pp. 256–265.

[8] Bezerra, C. E.; Pedone, F.; Renesse, R. V. “Scalable state-machine replication”. In: Proceedings of the International Conference on Dependable Systems and Networks, 2014, pp. 331–342.

[9] Birman, K. P.; Joseph, T. A. “Reliable communication in the presence of failures”, *ACM Transactions on Computer Systems*, vol. 5–1, Jan 1987, pp. 47–76.

[10] Burrows, M. “The chubby lock service for loosely-coupled distributed systems”. In: Proceedings of the Symposium on Operating Systems Design and Implementation, 2006, pp. 335–350.

[11] Böger, D. S.; Fraga, J. S.; Alchieri, E. “Reconfigurable scalable state machine replication”. In: Proceedings of the Latin-American Symposium on Dependable Computing, 2016, pp. 1–8.

[12] Cachin, C.; Vukolic, M. “Blockchain consensus protocols in the wild”. In: Proceedings of the International Symposium on Distributed Computing, 2017, pp. 1–16.

[13] Castro, M.; Liskov, B. “Practical byzantine fault tolerance and proactive recovery”, *ACM Transactions on Computer Systems*, vol. 20–4, Nov 2002, pp. 398–461.

[14] Ceolin Junior, T.; Dotti, F.; Pedone, F. “Parallel state machine replication from generalized consensus”. In: Proceedings of the International Symposium on Reliable Distributed Systems, 2020, pp. 133–142.

[15] Coelho, P.; Ceolin Junior, T.; Bessani, A.; Dotti, F.; Pedone, F. “Byzantine fault-tolerant atomic multicast”. In: Proceedings of the International Conference on Dependable Systems and Networks, 2018, pp. 39–50.

[16] Coelho, P.; Pedone, F. “Geographic state machine replication”. In: Proceedings of the Symposium on Reliable Distributed Systems, 2018, pp. 221–230.

[17] Coelho, P. R.; Schiper, N.; Pedone, F. “Fast atomic multicast”. In: Proceedings of the International Conference on Dependable Systems and Networks, 2017, pp. 37–48.

[18] Corbett, J. C.; Dean, J.; Epstein, M.; Fikes, A.; Frost, C.; Furman, J. J.; Ghemawat, S.; Gubarev, A.; Heiser, C.; Hochschild, P.; Hsieh, W.; Kanthak, S.; Kogan, E.; Li, H.; Lloyd, A.; Melnik, S.; Mwaura, D.; Nagle, D.; Quinlan, S.; Rao, R.; Rolig, L.; Saito, Y.; Szymaniak, M.; Taylor, C.; Wang, R.; Woodford, D. “Spanner: Google’s globally distributed database”, *ACM Transactions on Computer Systems*, vol. 31, Aug 2013, pp. 1–22.

[19] Cui, H.; Gu, R.; Liu, C.; Chen, T.; Yang, J. “Paxos made transparent”. In: Proceedings of the Symposium on Operating Systems Principles, 2015, pp. 105–120.

[20] DeCandia, G.; Hastorun, D.; Jampani, M.; Kakulapati, G.; Lakshman, A.; Pilchin, A.; Sivasubramanian, S.; Vosshall, P.; Vogels, W. “Dynamo: Amazon’s highly available key-value store”. In: Proceedings of the Symposium on Operating Systems Principles, 2007, pp. 205–220.

[21] Défago, X.; Schiper, A.; Urbán, P. “Total order broadcast and multicast algorithms: Taxonomy and survey”, *ACM Computing Surveys*, vol. 36–4, Dec 2004, pp. 372–421.

[22] Delporte-Gallet, C.; Fauconnier, H. “Fault-tolerant genuine atomic multicast to multiple groups”. In: Proceedings of the International Conference on Principles of Distributed Systems, 2000, pp. 107–122.

[23] Dwork, C.; Lynch, N.; Stockmeyer, L. “Consensus in the presence of partial synchrony”, *Journal of the ACM*, vol. 35, apr 1984, pp. 103–118.

[24] Enes, V.; Baquero, C.; Gotsman, A.; Sutra, P. “Efficient replication via timestamp stability”. In: Proceedings of the Sixteenth European Conference on Computer Systems, 2021, pp. 178–193.

[25] Escobar, I. A.; Alchieri, E.; Dotti, F. L.; Pedone, F. “Boosting concurrency in parallel state machine replication”. In: Proceedings of the International Middleware Conference, 2019, pp. 228–240.

[26] Fischer, M. J.; Lynch, N. A.; Paterson, M. S. “Impossibility of distributed consensus with one faulty process”, *Journal of the ACM*, vol. 32–2, Apr 1985, pp. 374–382.

[27] Fritzke, U.; Ingels, P.; Mostéfaoui, A.; Raynal, M. “Fault-tolerant total order multicast to asynchronous groups”. In: Proceedings of the Symposium on Reliable Distributed Systems, 1998, pp. 228.

[28] Glendenning, L.; Beschastnikh, I.; Krishnamurthy, A.; Anderson, T. “Scalable consistency in scatter”. In: Proceedings of the ACM Symposium on Operating Systems Principles, 2011, pp. 15–28.

[29] Guerraoui, R.; Schiper, A. “Total order multicast to multiple groups”. In: Proceedings of International Conference on Distributed Computing Systems, 1997, pp. 578–585.

[30] Guo, Z.; Hong, C.; Yang, M.; Zhou, D.; Zhou, L.; Zhuang, L. “Rex: Replication at the speed of multi-core”. In: Proceedings of the Conference on Computer Systems, 2014, pp. 1–14.

[31] Hadzilacos, V.; Toueg, S. “A modular approach to fault-tolerant broadcasts and related problems”, Technical Report, Cornell University, 1994, 83p.

[32] Herlihy, M. P.; Wing, J. M. “Linearizability: A correctness condition for concurrent objects”, *ACM Transactions on Programming Languages and Systems*, vol. 12–3, jul 1990, pp. 463–492.

[33] Hunt, P.; Konar, M.; Junqueira, F. P.; Reed, B. “ZooKeeper: Wait-free coordination for internet-scale systems”. In: Proceedings of the USENIX Annual Technical Conference, 2010, pp. 1–11.

[34] Junqueira, F. P.; Reed, B. C.; Serafini, M. “Zab: High-performance broadcast for primary-backup systems”. In: Proceedings of the International Conference on Dependable Systems and Networks, 2011, pp. 245–256.

[35] Kapitza, R.; Schunter, M.; Cachin, C.; Stengel, K.; Distler, T. “Storyboard: Optimistic deterministic multithreading”. In: Proceedings of the Workshop on Hot Topics in System Dependability, 2010, pp. 1–6.

[36] Kapritsos, M.; Junqueira, F. P. “Scalable agreement: toward ordering as a service”. In: Proceedings of the International Conference on Hot Topics in System Dependability, 2010, pp. 1–8.

[37] Kapritsos, M.; Wang, Y.; Quema, V.; Clement, A.; Alvisi, L.; Dahlin, M. “All about eve: Execute-Verify replication for Multi-Core servers”. In: Proceedings of the USENIX Symposium on Operating Systems Design and Implementation, 2012, pp. 237–250.

[38] Kotla, R.; Dahlin, M. “High throughput byzantine fault tolerance”. In: Proceedings of the International Conference on Dependable Systems and Networks, 2004, pp. 575–584.

[39] Lakshman, A.; Malik, P. “Cassandra: A decentralized structured storage system”, *ACM SIGOPS Operating Systems Review*, vol. 44, Apr 2010, pp. 35–40.

[40] Lamport, L. “Time, clocks, and the ordering of events in a distributed system”, *Communications of the ACM*, vol. 21–7, Jul 1978, pp. 558–565.

[41] Lamport, L. “Using time instead of timeout for fault-tolerant distributed systems”, *ACM Transactions on Programming Languages and Systems*, vol. 6–2, Apr 1984, pp. 254–280.

[42] Lamport, L. “The part-time parliament”, *ACM Transactions on Computer Systems*, vol. 16–2, May 1998, pp. 133–169.

[43] Lamport, L. “Paxos made simple”, *ACM SIGACT News*, vol. 32, Dec 2001, pp. 51–58.

[44] Lamport, L. “Generalized consensus and paxos”, Technical Report, Microsoft Research, 2005, 60p.

[45] Lamport, L. “Fast paxos”, *Distributed Computing*, vol. 19–2, Oct 2006, pp. 79–103.

[46] Lamport, L.; Shostak, R.; Pease, M. “The Byzantine generals problem”, *ACM Transactions on Programming Languages and Systems*, vol. 4–3, Jul 1982, pp. 382–401.

[47] Le, L. H.; Bezerra, C. E.; Pedone, F. “Dynamic scalable state machine replication”. In: Proceedings of the International Conference on Dependable Systems and Networks, 2016, pp. 13–24.

[48] Marandi, P. J.; Bezerra, C. E.; Pedone, F. “Rethinking state machine replication for parallelism”. In: Proceedings of the International Conference on Distributed Computing Systems, 2014, pp. 368–377.

[49] Marandi, P. J.; Pedone, F. “Optimistic parallel state-machine replication”. In: Proceedings of the International Symposium on Reliable Distributed Systems, 2014, pp. 57–66.

[50] Marandi, P. J.; Primi, M.; Pedone, F. “Multi-ring paxos”. In: Proceedings of the International Conference on Dependable Systems and Networks, 2012, pp. 1–12.

[51] Mendizabal, O.; de Moura, R.; Dotti, F.; Pedone, F. “Efficient and deterministic scheduling for parallel state machine replication”. In: Proceedings of the International Parallel & Distributed Processing Symposium, 2017, pp. 748–757.

[52] Moraru, I.; Andersen, D. G.; Kaminsky, M. “There is more consensus in egalitarian parliaments”. In: Proceedings of the ACM Symposium on Operating Systems Principles, 2013, pp. 358–372.

[53] Mostéfaoui, A.; Raynal, M. “Solving consensus using chandra-toueg’s unreliable failure detectors: A general quorum-based approach”. In: Proceedings of the International Symposium on Distributed Computing, 1999, pp. 49–63.

[54] Mu, S.; Nelson, L.; Lloyd, W.; Li, J. “Consolidating concurrency control and consensus for commits under conflicts”. In: Proceedings of the USENIX Symposium on Operating Systems Design and Implementation, 2016, pp. 517–532.

[55] Olszewski, M.; Ansel, J.; Amarasinghe, S. “Kendo: Efficient deterministic multithreading in software”, *SIGARCH Computer Architecture News*, vol. 37–1, Mar 2009, pp. 97–108.

[56] Ongaro, D.; Ousterhout, J. “In search of an understandable consensus algorithm”. In: Proceedings of the USENIX Annual Technical Conference, 2014, pp. 305–320.

[57] Pedone, F.; Schiper, A. “Handling message semantics with generic broadcast protocols”, *Distributed Computing Journal*, vol. 15–2, Apr 2002, pp. 97–107.

[58] Rodrigues, L.; Guerraoui, R.; Schiper, A. “Scalable atomic multicast”. In: Proceedings of the International Conference on Computer Communications and Networks, 1998, pp. 840–847.

[59] Schiper, N.; Pedone, F. “On the Inherent Cost of Atomic Broadcast and Multicast in Wide Area Networks”. In: Proceedings of the International Conference on Distributed Computing and Networking, 2008, pp. 147–157.

[60] Schneider, F. B. “Implementing fault-tolerant services using the state machine approach: A tutorial”, *ACM Computing Surveys*, vol. 22–4, Dec 1990, pp. 299–319.

[61] Sousa, J.; Bessani, A.; Vukolic, M. “A byzantine fault-tolerant ordering service for the hyperledger fabric blockchain platform”. In: Proceedings of the International Conference on Dependable Systems and Networks, 2018, pp. 51–58.

[62] Tarjan, R. “Depth-first search and linear graph algorithms”. In: Proceedings of the Annual Symposium on Switching and Automata Theory, 1971, pp. 114–121.

Pontifícia Universidade Católica do Rio Grande do Sul
Pró-Reitoria de Pesquisa e Pós-Graduação
Av. Ipiranga, 6681 – Prédio 1 – Térreo
Porto Alegre – RS – Brasil
Fone: (51) 3320-3513
E-mail: propesq@pucrs.br
Site: www.pucrs.br