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

Explorando Agrupamento de Requisições Conflitantes Durante Escalonamento em Replicação Máquina de Estados Paralela

Marcella Pantarotto

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

Orientador
Prof. Dr. Eduardo Adilio Pelinson Alchieri

Brasília
2022

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

Explorando Agrupamento de Requisições Conflitantes Durante Escalonamento em Replicação Máquina de Estados Paralela

Marcella Pantarotto

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

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

Prof. Dr. Marcos Fagundes Caetano
CIC/UnB

Prof. Dr. Lucas Rodrigues Costa
CIC/UnB

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

Brasília, 02 de maio de 2022

Dedicatória

Dedico este trabalho a todos que me apoiaram nesta jornada de dupla graduação: à minha família, aos meus amigos e aos meus professores.

Agradecimentos

Primeiramente a Deus, que me guia e protege ao longe de toda a minha eternidade.

À minha mãe e à minha tia Cláudia que ilimitadamente acreditam em meus potenciais e vibram a cada conquista obtida.

Ao Prof. Dr. Eduardo Alchieri por toda a dedicação no ensino e apoio que colaborou na elaboração deste e outros trabalhos.

Aos pesquisadores das pesquisas anteriores que vêm trabalhando no tema ao longo de anos, possibilitando a construção do conhecimento inicial deste projeto.

À Universidade de Brasília, ao Departamento de Ciência da Computação e às pessoas com quem compartilhei, ao longo desses anos, momentos inesquecíveis da minha formação acadêmica.

Aos meus amigos Ana Paula, Jaqueline, Alexandre e Gabriel que carinhosamente sempre compartilharam conhecimentos e experiências comigo durante o curso.

Ao meu namorado Victor Kelvin e ao meu amigo Vandré por todo o apoio dispensado na conclusão deste trabalho.

E, por fim, a todos que contribuíram na minha formação de forma direta ou indiretamente.

Muito obrigada!

Resumo

Propostas de escalonamento em Replicação Máquina de Estados Paralela (RME paralela) tiram proveito da semântica das requisições e permitem a execução paralela de parte delas. Isso leva a uma otimização no desempenho em relação à Replicação Máquina de Estados (RME) clássica. Apesar de ser uma abordagem popular na implementação de sistemas distribuídos tolerante a falhas, a RME clássica pode apresentar gargalos no processamento caso tenha que lidar com uma alta carga de requisições conflitantes. Especificamente no cenário de escalonamento tardio, um Grafo Acíclico Dirigido (DAG) é utilizado para gerenciar os conflitos do sistema. Os vértices representam as requisições e as arestas representam as dependências (conflitos) entre estas requisições. À medida que o número de requisições conflitantes aumenta, aumenta também o número de dependências e arestas no grafo, aumentando a necessidade de sincronização destas dependências e diminuindo o desempenho geral do sistema. A proposta deste trabalho é melhorar o desempenho em sistemas distribuídos com alta demanda de requisições conflitantes, através do agrupamento de requisições conflitantes para que sejam escalonadas juntas em um vértice do grafo. Esta abordagem propõe simplificar o DAG com um número menor de vértices e arestas e, consequentemente, diminuir a necessidade de sincronizações entre as dependências. Assim, o intuito do trabalho é comprovar que o tratamento de requisições conflitantes impacta diretamente no desempenho de sistemas distribuídos.

Palavras-chave: Replicação Máquina de Estados Paralela, paralelismo, concorrência, escalonamento tardio, requisições conflitantes, dependência

Abstract

Scheduling proposals in Parallel State Machine Replication (Parallel SMR) take advantage of the semantics of requests and allow parallel execution of some of them. This leads to a performance optimization over the classic State Machine Replication (SMR). Despite being a popular approach for implementing fault-tolerant distributed systems, classic SMR can present processing bottlenecks if it has to deal with a high load of conflicting requests. Specifically in the late scheduling scenario, a Directed Acyclic Graph (DAG) is used to manage system conflicts. The nodes represent the requests and the edges represent the dependencies (conflicts) between these requests. As the number of conflicting requests increases so does the number of dependencies and edges in the graph, thereby increasing the need to synchronize these dependencies and decreasing the system's overall performance. The proposal of this study is to improve performance in distributed systems with a high load of conflicting requests, by grouping conflicting requests so that they can be scheduled together in a node of the graph. This approach aims to simplify the DAG with a smaller number of nodes and edges, and, consequently, reduce the need for synchronizations between dependencies. Thus, the intention of this study is to prove that handling conflicting requests directly impacts the performance of distributed systems.

Keywords: Parallel State Machine Replication, parallelism, concurrency, late scheduling, conflicting requests, dependency

Sumário

1 Introdução **1**
1.1 Objetivos . . . 3
1.2 Organização do Texto . . . 3

2 Fundamentação Teórica e Trabalhos Relacionados **5**
2.1 Sistemas Distribuídos . . . 5
2.2 Tolerância a Falhas . . . 9
2.2.1 Classificação das Falhas . . . 10
2.2.2 Erros e Defeitos . . . 11
2.2.3 Implementação da Tolerância a Falhas . . . 13
2.2.4 A Problemática das Falhas Bizantinas . . . 14
2.3 Replicação Máquina de Estados . . . 15
2.3.1 Máquina de Estados Tolerante a Falhas . . . 15
2.3.2 BFT-SMaRt . . . 16
2.4 Replicação Máquina de Estados Paralela . . . 18
2.4.1 Escalonamento Estático . . . 21
2.4.2 Escalonamento Antecipado . . . 21
2.4.3 Escalonamento Tardio . . . 22
2.5 Trabalhos Relacionados . . . 23
2.6 Considerações Finais . . . 24

3 Agrupamento de Requisições Conflitantes **27**
3.1 Escalonador Tardio . . . 27
3.2 Proposta de Agrupar Requisições Conflitantes . . . 30
3.3 Considerações Finais . . . 33

4 Experimentos **34**
4.1 Ambiente Experimental . . . 34
4.2 Aplicação Utilizada . . . 35
4.3 Cargas de Trabalho . . . 36

4.4 Resultados 36

5 Conclusão **39**

Referências **40**

Lista de Figuras

2.1 Relação entre Falhas, Erros e Defeitos (Fonte: Avizienis et al. [1]). 10
2.2 As classes de falhas elementares (Fonte: Avizienis et al. [1]). 11
2.3 As classes de falhas combinadas numa representação matricial, com exemplos (Fonte: Avizienis et al. [1]). 12
2.4 As classes de falhas combinadas numa representação em árvore (Fonte: Avizienis et al. [1]). 12
2.5 Classes de protocolos que implementam RME paralela (Fonte: Alchieri et al. [2]). 20

3.1 Grafo de dependências do escalonador tardio. 29
3.2 Grafo de dependências com o agrupamento de comandos conflitantes. 32

4.1 Resultados dos testes 37

Lista de Abreviaturas e Siglas

API Interface de Programação de Aplicação.

BFT *Byzantine Fault Tolerant.*

COS *Confict-Ordered Set.*

DAG Grafo Acíclico Dirigido.

FIFO *First In, First Out.*

ME Máquina de Estados.

Parallel SMR *Parallel State Machine Replication.*

PGB Problema dos Generais Bizantinos.

RME Replicação Máquina de Estados.

RME paralela Replicação Máquina de Estados Paralela.

RME-TFB Replicação Máquina de Estados Tolerante a Falhas Bizantinas.

RMEs paralelas Replicação Máquinas de Estados Paralelas.

SMR *State Machine Replication.*

TFB Tolerância a Falhas Bizantinas.

URL Localizador Uniforme de Recursos.

Capítulo 1

Introdução

Tradicionalmente, a computação é realizada em apenas uma máquina. Contudo, há situações em que o poder computacional de uma única máquina não é o suficiente para o processamento. Um único computador pode ser muito lento para solucionar um problema de grande complexidade. Por esse motivo, surgiu a ideia de conectar múltiplos computadores em uma rede para resolver problemas de grande complexidade computacional que normalmente seriam resolvidos em supercomputadores.

> "Não importa o quão poderosos os computadores individuais se tornem, ainda há razões para aproveitar o poder de várias unidades computacionais, muitas vezes espalhadas por grandes áreas geográficas. Às vezes, isso é motivado pela necessidade de coletar dados de locais amplamente dispersos (por exemplo, páginas da Web de servidores ou sensores de clima ou tráfego). Outras vezes é motivado pela necessidade de realizar enormes cálculos que simplesmente não podem ser feitos por uma única CPU."
>
> (Google® - 2022) [3]

A ideia principal é dividir para conquistar, ou seja, pegar uma tarefa grande e complexa e dividi-la em pedaços menores para, então, distribuir entre diversos computadores. Assim, cada computador realizará uma tarefa pequena e se torna possível completar a tarefa original de grande complexidade computacional em um período menor de tempo, visto que todos os computadores trabalham de forma colaborativa.

Conceitualmente, computação distribuída é simples, porém escolher quais tarefas suportam uma execução distribuída deve ser feito com cautela. A caracterização de um padrão de execução sequencial pode dificultar a divisão e distribuição entre computadores, porque o resultado de uma computação é dependente e influenciada pelo resultado das demais. Assim sendo, as tarefas mais adequadas pra computação distribuída são aquelas que podem ser executadas de forma paralela, ou seja, operações que podem ser executadas de forma independente do resultado das demais máquinas. As tarefas são executadas por *threads*, também chamadas de fluxo de controle, fluxo de execução ou linha de execução.

Um sistema distribuído é composto por unidades independentes de software ou hardware, comumente chamados de nós, que trabalham juntas de forma coordenada em vários fluxos independentes de execução. Esses nós se comunicam entre si por meio de mensagens ou eventos, visando a colaboração na realização de uma tarefa. O objetivo de tal sistema é abstrair para o usuário final a complexidade de sua arquitetura, estrutura, gerenciamento de memória e o seu modo de operação.

Replicação Máquina de Estados (RME) é uma abordagem bastante abrangente e usada para a implementação de sistemas distribuídos tolerante a falhas. Esta abordagem consiste em replicar os servidores e coordenar suas ações de forma que todos eles apresentem a mesma evolução em seus estados. Para isso, os servidores (ou réplicas) precisam ordenar as requisições recebidas dos clientes e executá-las sequencialmente. Apesar de adequada, esta abordagem limita o desempenho para aquilo que uma única *thread* consegue executar.

Visando superar esta limitação, recentemente surgiram abordagens para permitir a execução paralela de requisições que não conflitam, chamadas de Replicação Máquinas de Estados Paralelas (RMEs paralelas). Requisições não conflitantes (ou independentes) são aquelas que acessam partes diferentes do estado da aplicação ou apenas fazem uma leitura das mesmas partes do estado (variáveis). Um aspecto fundamental das RMEs paralelas é como escalonar as requisições para serem executadas por um conjunto de *threads* (operárias) de forma que requisições conflitantes sejam executadas sequencialmente enquanto que as independentes possam ser executadas de forma paralela.

Dentre as propostas para RMEs paralelas, este trabalho considera o escalonamento tardio, em que as decisões de escalonamento são decididas dinamicamente após a emissão das requisições, i.e, nas réplicas existe uma *thread* paralelizadora responsável pelo escalonamento das requisições. Essa *thread* utiliza um grafo para gerenciar as dependências entre as requisições, em que cada vértice é uma requisição e as arestas definem as dependências. Claramente, aumentando a quantidade de requisições conflitantes, também se aumenta a quantidade de dependências/arestas neste grafo, fazendo-se com que o desempenho total do sistema diminua. Neste sentido, este trabalho propõe uma solução para o agrupamento de requisições conflitantes em um mesmo vértice do grafo, diminuindo as necessidades de sincronização e aumentando o desempenho do sistema. Note que este agrupamento deve ser determinístico entre as réplicas, para que as mesmas mantenham as propriedades da RME.

O desempenho é um fator de importância crescente na concepção de sistemas distribuídos e os pontos supracitados constituem um grande esforço computacional para lidar com as dependências e os conflitos entres requisições. O esforço para rastrear dependências impacta o desempenho, pois o escalonamento das requisições deve respeitar as dependências de modo a organizá-las em sequência para garantir a precisão do sistema e isso

implica em um gargalo no processamento. Dessa forma, surgiu a oportunidade de estudar possibilidades que aprimorassem o processamento de um sistema com um alto número de conflitos. A proposta desenvolvida neste estudo utilizou um projeto que opera em sistemas distribuídos, com escalonamento tardio e uma abordagem *lock-free* para manipular o grafo de dependências.

A *thread* responsável pelo escalonamento adicionava diretamente os comandos no grafo, um por nó, desconsiderando se estes possuíam ou não dependências. Dessa forma, comandos sem dependências não geravam conflitos e eram logo executados, porém os comandos com dependências exigem que suas dependências sejam adicionadas no grafo e que a execução seja feita de forma sequencial para resolver todos os conflitos. A direção da dependência implica na ordem de execução, se o comando A é dependência do comando B, então a execução de A deve preceder a execução de B.

Diante do exposto, a hipótese levantada foi que o agrupamento de comandos conflitantes em um único nó (vértice) do grafo pode melhorar o desempenho do sistema, ou seja, a quantidade de comandos executados por segundo. Como os resultados desses comandos influenciam nos demais, sendo escalonados e executados juntos permite que o sistema opere sem precisar rastrear dependências e mantenha o processamento em alta velocidade. Os comandos sem conflitos permaneceriam com o mesmo tratamento, pois a intenção é otimizar a forma de lidar com os conflitos.

1.1 Objetivos

O objetivo geral deste trabalho é propor uma solução para agrupar requisições dependentes em um mesmo vértice do grafo, considerando o escalonamento tardio, e analisar seus impactos no desempenho do sistema. Para isso, os seguintes objetivos específicos foram definidos:

- Estudar RMEs paralelas e os tipos de escalonamento.
- Propor uma solução que agrupe requisições dependentes no escalonamento tardio.
- Implementar esta solução e analisar seus impactos no desempenho do sistema.

1.2 Organização do Texto

O presente trabalho foi elaborado em cinco capítulos, além das referências bibliográficas apresentadas nas últimas páginas. Este é o primeiro e introduz o contexto do trabalho, bem como os objetivos estabelecidos para a pesquisa, levando em consideração o problema apresentado e a hipótese levantada como solução.

O Capítulo 2 expõe toda a fundamentação teórica que embasou esta pesquisa e os trabalhos relacionados ao tema desenvolvido que existem na comunidade acadêmica. Esse capítulo é subdividido em seis seções, a Seção 2.1 apresenta o contexto geral dos sistemas distribuídos. Exibe o seu funcionamento e os sete tipos principais de transparência que todo sistema distribuído possui. A Seção 2.2 aborda como um sistema distribuído lida com falhas e as maneiras de construir sistemas que sejam tolerantes a elas. Nessa seção, as falhas foram classificadas e, depois, diferenciadas de erros e defeitos.

A solução para lidar com as falhas é apresentada na Seção 2.3, referente a Replicação Máquina de Estados (RME). Nessa seção também é apresentada uma biblioteca robusta de Replicação Máquina de Estados Tolerante a Falhas Bizantinas (RME-TFB) chamada BFT-SMaRt, utilizada na arquitetura do projeto. A Seção 2.4 complementa a seção anterior, expandindo as réplicas ao paralelismo em Replicação Máquina de Estados Paralela (RME paralela). Três protocolos de escalonamento são apresentados: estático, antecipado e tardio, sendo o último o escolhido para a arquitetura do trabalho. A Seção 2.5 expõe trabalhos acadêmicos relacionados a este. São evidenciando principalmente os trabalhos de Escobar et al. (2019) e Pintor et al. (2020), por serem antecedentes deste. Por fim, a Seção 2.6 explana as conclusões finais sobre o capítulo.

O Capítulo 3 apresenta a proposta elaborada por esta pesquisa. A Seção 3.1 mostra o cenário do escalonador tardio, em que todos os comandos são escalonados assim que recebidos pela *thread* escalonadora. A Seção 3.2 expõe a proposta de agrupar comandos conflitantes. Enquanto um grupo de comandos com dependências são escalonados em um único nó, os comandos sem dependências permanecem sendo escalonados em nós diferentes. Em ambas as seções, há o pseudocódigo da *thread* escalonadora e das *threads* operárias e também o grafo de dependências gerado pela *thread* escalonadora em cada um dos cenários.

Para analisar a eficiência da hipótese levantada, uma comparação foi realizada entre o não agrupamento e o agrupamento de comandos conflitantes dentro do sistema de escalonamento tardio. Os resultados são evidenciados em gráficos e exibidos juntamente com uma análise no Capítulo 4. O ambiente experimental em que os experimentos foram realizados, bem como as cargas de trabalho utilizadas também foram detalhados nesse capítulo. O Capítulo 5 finaliza o estudo diante de todo o exposto, trazendo a conclusão da pesquisa com base nos resultados apresentados no capítulo anterior.

Capítulo 2

Fundamentação Teórica e Trabalhos Relacionados

Este capítulo aborda os principais tópicos necessários para a compreensão da fundamentação teórica deste trabalho. A Seção 2.1 introduz o significado de sistema distribuído e explica o seu funcionamento. A Seção 2.2 esclarece como é feita a tolerância de falhas dentro de sistemas distribuídos, incluindo os tipos mais recorrentes de falhas. A Seção 2.3 explica o que é Replicação Máquina de Estados (RME) e como é útil para um sistema distribuído. A Seção 2.4 avança neste assunto e explora Replicação Máquina de Estados Paralela (RME paralela), além de apresentar três abordagens de escalonamento. A Seção 2.5 discute sobre demais trabalhos existentes na comunidade acadêmica referentes ao mesmo tema. Por fim, a Seção 2.6 sintetiza tudo que foi abordado no capítulo com algumas pontuações finais.

2.1 Sistemas Distribuídos

Com a evolução da tecnologia e o barateamento de recursos, a utilização de computadores tem se tornado mais viável, ao ponto de crescer a exploração do acoplamento de diversos computadores para a realização de uma ou mais tarefas. Essa união de computadores, podendo ser desde computadores *mainframe* de alto desempenho a pequenos dispositivos em redes de sensores, exige uma descentralização de processamento chamada de sistema distribuído [4, 5]. Van Steen e Tanenbaum (2016) definem um sistema distribuído como "um conjunto de elementos computacionais autônomos que aparece para seus usuários como um sistema único e coerente" [4]. Essa definição é fundamentada no fato de que cada elemento computacional, seja um dispositivo de hardware ou um processo de software, possui a capacidade de operar de forma independente dos demais. Outro aspecto

importante dessa definição é que o usuário utilizando o sistema, seja ele uma pessoa ou aplicativo, não deve presumir a existência de tal distribuição espacial dos recursos [4, 6].

Dessa forma, o usuário não se preocupa com a localidade onde os dados estão armazenados e nem onde exatamente será feito o processamento de cada parte do processo, pois terá uma visão não distribuída de um sistema distribuído [6]. Cada elemento do sistema distribuído pode ser chamado de nó e, apesar de serem independentes, são programados para atuarem de forma colaborativa e alcançar os mesmos objetivos. Todo o conjunto de nós opera da mesma forma, não importa onde, quando e como a interação entre o usuário e o sistema ocorre [4]. Por serem independentes, não existe um relógio global que sincronize todos eles, cada um trabalha com um horário lógico local; portanto, é importante que haja comunicação entre todos os nós em cada etapa do processo [4, 7]. Isso se dá através de troca de mensagens organizadas em um meio de comunicação comum, em que cada nó processa o que foi recebido e envia informações sobre o trabalho que realizou [4, 6].

Há sempre um caminho de comunicação entre dois nós que viabiliza a troca de mensagens. Sendo assim, a organização dos nós possui uma simbólica importância, podendo ser a parte mais complexa do gerenciamento de um sistema distribuído [4]. A partir disso, o cálculo da complexidade de um algoritmo distribuído pode ser avaliado por duas métricas: tempo de envio e complexidade das mensagens. O primeiro mede o tempo necessário para a transmissão de uma mensagem dentro do sistema. O segundo possui diversas métricas, sendo a mais comum a quantidade total de mensagens transmitidas. Contudo, caso as mensagens tenham uma ordem igual ou superior a algumas centenas de bits, o número total de bits enviados pode ser uma medida de custo melhor [6].

Tradicionalmente, em sistemas não distribuídos, a atividade computacional é representada por processos executados de forma sequencial, em que um evento é executado atrás do outro seguindo uma ordem, sejam eles dependentes ou não. Já em sistemas distribuídos é possível a execução simultânea de diferentes processos sequenciais [6]. Se um evento depende de outro, ainda há a necessidade de uma execução sequencial para garantir que todas as dependências sejam cumpridas. No entanto, se os eventos são completamente independentes entre si, podem ser processados de forma concorrente ou até mesmo paralela em nós distintos.

Em cada nó do sistema distribuído é gerado uma sequência de eventos ordenados e cada evento possui uma série de atributos que os definem, como, por exemplo, o horário de execução e as dependências necessárias. Esses atributos são importantes para a depuração de um sistema distribuído e a ordenação dos nós. Dessa forma, os eventos podem ser ordenados de duas maneiras: (i) ordenando pares de eventos que ocorrem no mesmo nó, independente dos eventos que ocorrem em nós diferentes e (ii) ordenando a interação de eventos dependentes entre nós diferentes. Por exemplo, se dois nós estão se comunicando

através de mensagens, para que a comunicação se complete o evento de envio da mensagem deve preceder o de recebimento da mensagem [7].

Como dito anteriormente, o horário de execução de cada evento é uma informação essencial em um sistema distribuído. Através desse dado é possível fazer um rastreamento das atividades de um nó e, consequentemente, rastrear toda a movimentação ocorrida dentro de uma execução do sistema. Esses rastreamentos normalmente são sintetizados em um *log*, no qual múltiplas execuções são registradas de forma ordenada. Como cada nó opera sob um horário lógico local, a ordenação entre eles é importante para alimentar o relógio vetorial presente em cada nó com a subestimativa do tempo lógico dos demais nós [7].

Segundo Van Steen e Tanenbaum (2016), a construção de um sistema distribuído se torna compensatória se quatro propósitos são atingidos, sendo eles: (i) facilidade de acesso aos recursos, (ii) não estampar a distribuição espacial destes recursos, (iii) ser aberto e (iv) ser escalável [4]. Os autores afirmam que facilitar o compartilhamento de recursos possui um princípio econômico, mas também concordam que favorece e colabora na troca de informações. Exemplificam esse fator com o BitTorrent, sistema bem-sucedido e famoso de rede ponto a ponto de compartilhamento de arquivos de mídia, como áudio e vídeo [4]. O download de um arquivo não se faz por meio de uma cópia local do que está armazenado em um servidor, em vez disso, o download é feito por meio da cópia de diferentes partes de outros usuários que possuem e compartilham o mesmo arquivo, até que uma cópia completa do arquivo seja realizada.

Os quatro propósitos supracitados de Van Steen e Tanenbaum (2016) podem ser atingidos através da exploração da transparência de recursos e processos, aqui referenciados como objetos [4, 8]. Existem principalmente sete tipos diferentes de transparência, sendo eles:

1. **Transparência de Acesso:** Como por vezes os sistemas distribuídos podem estar espalhados por máquinas com sistemas operacionais diferentes, cada um com regras de acesso e nomenclaturas diferentes, procura-se encontrar um acordo para acessar e exibir os objetos com as mesmas operações, independentemente de serem locais ou remotos. Procura-se ocultar as diferenças entre as arquiteturas trabalhadas, na forma como representam os dados e como acessam os objetos, encontrando uma maneira consistente que não importa onde eles estejam realmente armazenados no sistema [4, 8].

2. **Transparência de Localização:** Permitir que os usuários acessem objetos sem conhecimento de sua localização física no sistema. Tais objetos podem ser escondidos através da nomenclatura, utilizando nomes lógicos. Um exemplo disso é o Localizador Uniforme de Recursos (URL) que não dá nenhuma pista sobre a localização

real do servidor Web do site. Outra forma de atingir o mesmo objetivo é usando identificadores, pois possibilita que um serviço dentro do sistema utilize a referência para localizar o objeto [4, 8].

3. **Transparência de Realocação:** Ocultar para o usuário que um objeto foi movido de um lugar para outro pelo sistema distribuído. Um exemplo disso é o URL, mencionado no item anterior, pois como não há indicação de sua localidade, não é perceptível se o objeto sempre se manteve no mesmo lugar ou foi transferido. Um site pode ser realocado entre servidores diferentes e continuará com o mesmo domínio [4].

4. **Transparência de Migração:** Diferentemente da transparência anterior, esta permite que o usuário transporte o objeto de um lugar para outro dentro do sistema, sem que algum dano seja causado à comunicação, às operações das aplicações que utilizam esses objetos e também aos resultados advindos destas operações [4, 8].

5. **Transparência de Replicação:** Esconder a existência de uma ou mais cópias do mesmo objeto, sem permitir a identificação de quantas são e nem liberar a visualização de uma instância específica de uma cópia. Para que isso ocorra, todas as réplicas devem ter o mesmo nome e deve-se garantir que todas as cópias tenham o mesmo conteúdo, ou seja, qualquer operação ou alteração que ocorra em uma, deve produzir igualmente os mesmos resultados nas outras [4, 8]. Esta transparência possibilita um aumento de disponibilidade e desempenho de um dado, pois uma cópia local facilita o acesso [4]. Também permite que uma réplica assuma o controle da operação caso outra venha a falhar [8].

6. **Transparência de Simultaneidade ou Concorrência:** Permitir o compartilhamento simultâneo dos objetos entre usuários independentes, sem existir interferência pelos múltiplos acessos e atualizações [4, 8]. Existem casos de compartilhamento cooperativo de recursos, mas há também o compartilhamento competitivo [4]. É necessário que haja procedimentos para garantir a consistência dos objetos. Isso é possível através de mecanismos de bloqueio que concedem o acesso exclusivo de algum recurso desejado para cada usuário [4, 8].

7. **Transparência de Falhas:** Esta é uma das transparências mais importantes, pois um sistema distribuído requer falhas imperceptíveis, para permitir que as aplicações continuem a funcionar sem qualquer impacto ao comportamento ou resultado [8]. Quando alguma parte do sistema para de funcionar corretamente, deve haver um mecanismo consecutivo de auto-recuperação desta parte. Mascarar algumas falhas são mais difíceis que outras, principalmente, quando é necessário reconhecer se um

processo está extremamente lento ou se encerrou. Como, por exemplo, os servidores Web que, dependendo de quão ocupados estão, é difícil distinguir se a rede está congestionada ou o servidor está inativo [4]. Devido à sua importância, esta transparência é melhor abordada na próxima seção, que explora como os sistemas distribuídos elaboram um controle de tolerância a falhas.

2.2 Tolerância a Falhas

Como um sistema distribuído é composto por vários nós é inevitável e previsível que, em algum momento, algum nó possa falhar. Como são nós independentes, pode acontecer que uma falha em um nó não afete o funcionamento de outro nó, porém pode ocorrer que atrapalhe o processamento dos demais e até mesmo o processo como um todo [5]. O vencedor do prêmio Turing, Leslei Lamport (1987) define sistemas distribuídos com base nessas possíveis falhas, definindo "um sistema distribuído como aquele no qual a falha de um computador que você nem sabia que existia pode tornar seu próprio computador inutilizável" [9]. A partir dessa definição, pode-se garantir que esses sistemas são fadados a apresentarem defeitos eventualmente e esses defeitos inesperados podem ser disruptivos e trazer consequências negativas ao usuário [4, 10, 11].

De acordo com os conceitos trazidos por Avizienis et al. (2004), mostrados na Figura 2.1, os defeitos de um sistema são originados a partir de falhas do sistema, que geram erros quando são ativadas [1]. Esses erros se propagam ao longo dos processamentos realizados pelo sistema e quando são percebidos pela interface do sistema, passam a ser defeitos do sistema. As falhas são peculiaridades de qualquer sistema complexo, mas em sistemas distribuídos são especialmente mais visíveis e devido a isso, torna-se essencial entender tais falhas e aprender a lidar com elas, para não permitir que se manifestem como defeitos do sistema [4, 10].

Essencialmente, um sistema distribuído deve ser construído de modo que consiga se recuperar automaticamente de falhas parciais sem que haja fortes impactos ao desempenho geral e também que, mesmo na ocorrência de falhas ou erros, seja capaz de continuar sua atuação de forma aceitável até que o reparo seja feito. Pode-se considerar que um sistema falhou quando este não consegue cumprir com o que foi proposto, ou seja, se é projetado e desenvolvido para fornecer um ou mais serviços e algum destes serviços não é entregue [5].

Dentro desse contexto, para que seja possível evitar certos defeitos do sistema é necessário que se construa um mecanismo de recuperação do sistema, que funcione a partir da detecção de erros que ocorreram com a ativação de alguma falha, ou seja, um mecanismo de tolerância a falhas.

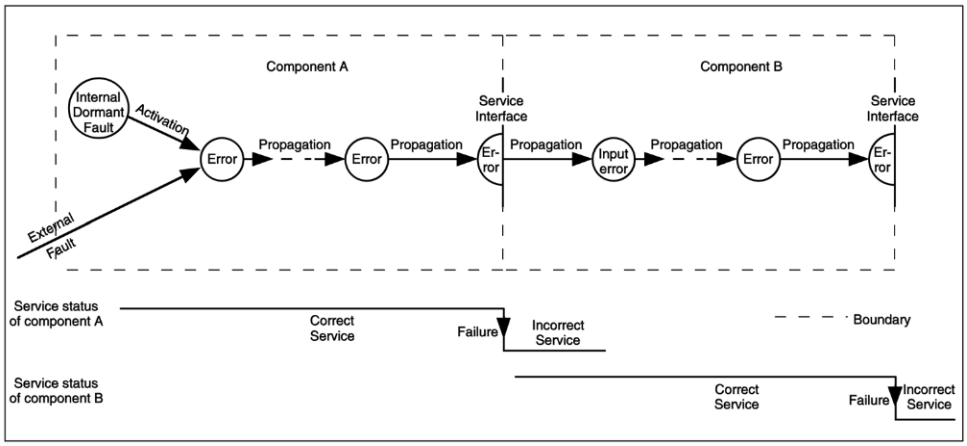


Figura 2.1: Relação entre Falhas, Erros e Defeitos (Fonte: Avizienis et al. [1]).

2.2.1 Classificação das Falhas

Normalmente, há três classificações para as falhas: transitórias, intermitentes ou permanentes. As transitórias são aquelas circunstanciais, que acontecem uma vez e depois desaparecem. As intermitentes são inconstantes, acontecem de forma espontânea e são bem difíceis de diagnosticar. Já a falha permanente é aquela que não deixará de existir até que o elemento defeituoso seja consertado ou substituído [5].

Outra classificação, proposta por Avizienis et al. (2004), é que existem substancialmente oito classes de falhas que podem ocorrer em sistemas distribuídos, estas classes se ramificam em 16 tipos, todos listados na Figura 2.2. Esses tipos de falhas podem ser combinados de acordo com critérios específicos, formando 31 combinações mais prováveis e que representam três grupos principais. São eles: falhas de desenvolvimento, falhas físicas e falhas de interação [1]. Nas Figuras 2.3 e 2.4 é possível visualizar a organização dessas combinações supracitadas, de acordo com os modelos de matriz e árvore, respectivamente. Ainda com base na terminologia proposta por Avizienis et al. (2004), cabe destacar as seguintes classes de falhas: falhas naturais, falhas humanas, falhas maliciosas e falhas de interação. O conhecimento acerca dessas falhas possibilita estabelecer um plano de tolerância a falhas personalizado para cada tipo de sistema de acordo com o contexto dado [1].

Sendo assim, as falhas naturais são aquelas relacionadas ao hardware e que envolvem fenômenos naturais, sem que haja a participação humana. Já as falhas humanas são as falhas causadas pela ação ou pela falta de ação humana, podendo ser combinada com outras classes, como, por exemplo, sendo maliciosa ou não maliciosa. Outra possível combinação que pode ser aplicada à classe das falhas humanas é a divisão entre falhas deliberativas ou não deliberativas, isto é, quando se tem ou não o conhecimento da pro-

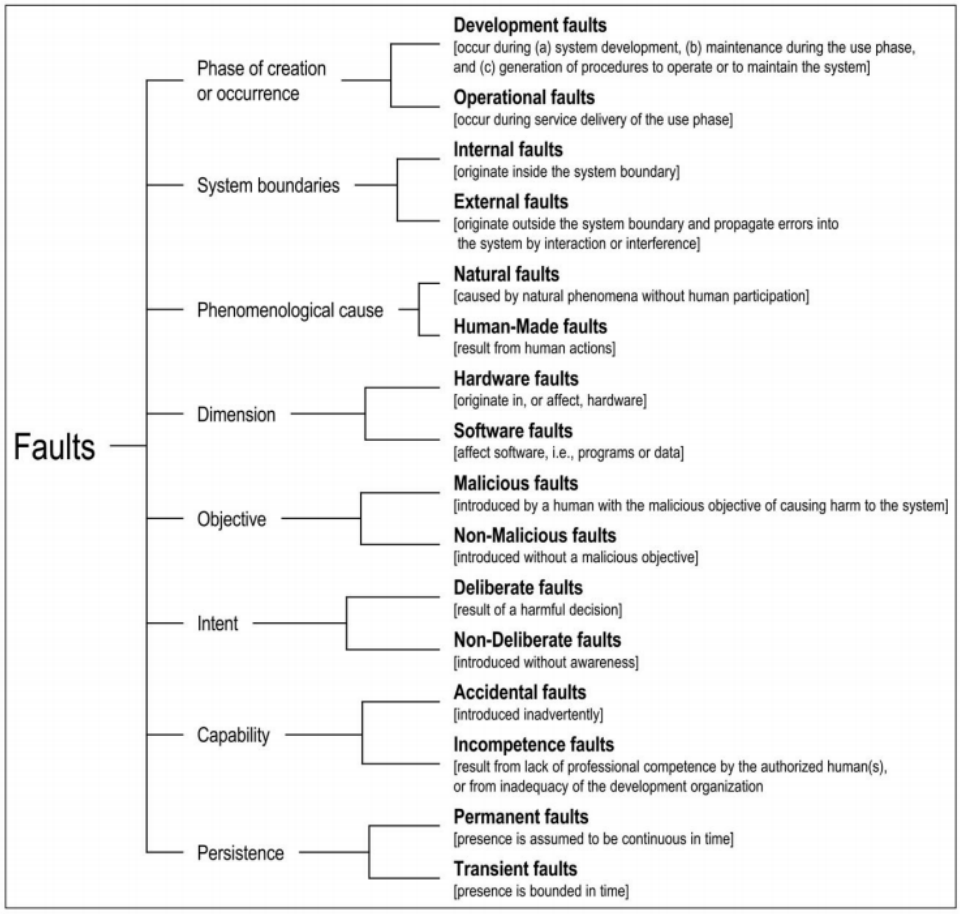


Figura 2.2: As classes de falhas elementares (Fonte: Avizienis et al. [1]).

dução da falha, respectivamente. Com relação às falhas maliciosas, é possível pontuar os três principais objetivos que motivam os agentes envolvidos nesta classe de falhas, são eles: (i) interromper o funcionamento do sistema, (ii) acessar informações confidenciais ou (iii) modificar indevidamente o sistema. Essa classe de falhas maliciosas se divide em 1) falhas lógicas, como cavalos de troia, vírus ou *trapdoors* e 2) tentativas de invasão, na qual há participação de agentes externos ao sistema. Por fim, as falhas de interação são as que ocorrem durante a fase de uso, portanto são falhas de operação [1].

2.2.2 Erros e Defeitos

Como dito anteriormente, as falhas podem gerar erros que se propagam ao longo do processamento até chegarem na interface do sistema, se transformando, assim, em um defeito. Tendo em vista que o conceito de um sistema defeituoso é definido por um sistema que não entrega o serviço que deveria entregar, ainda cabe pontuar as principais características acerca de erros e defeitos. Ainda de acordo com Avizienis et al. (2004) ,

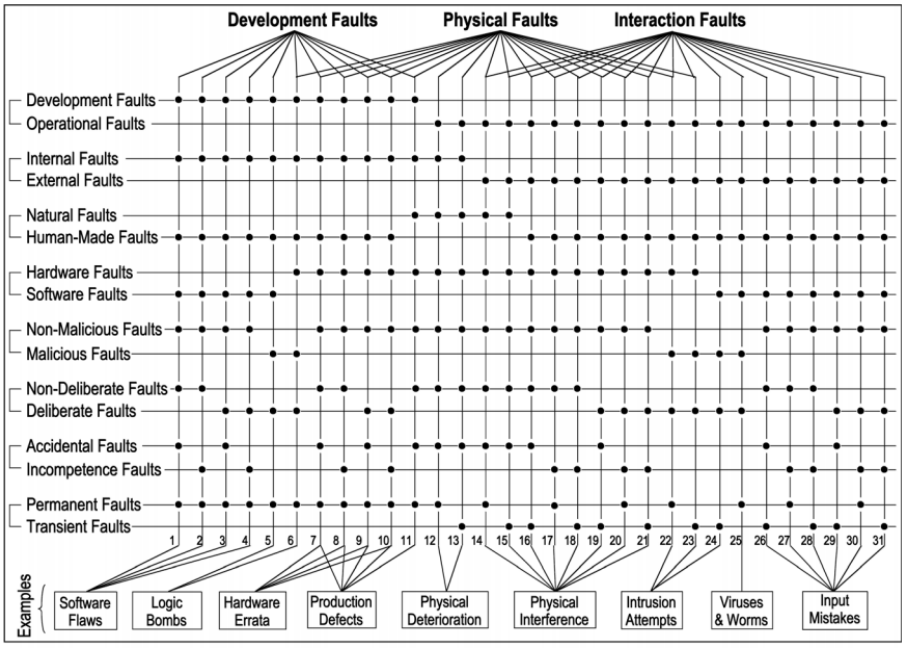


Figura 2.3: As classes de falhas combinadas numa representação matricial, com exemplos (Fonte: Avizienis et al. [1]).

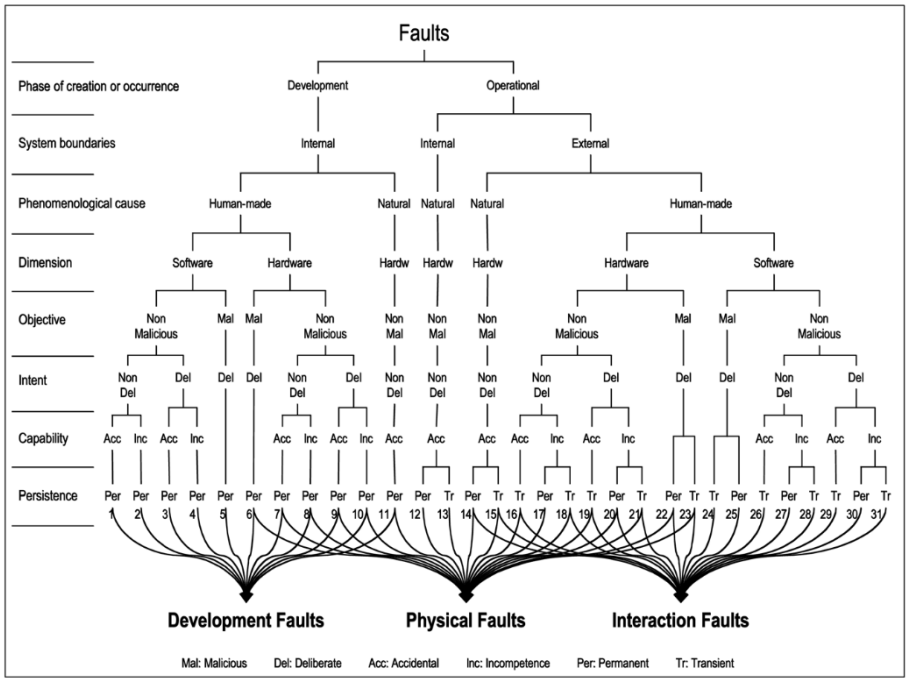


Figura 2.4: As classes de falhas combinadas numa representação em árvore (Fonte: Avizienis et al. [1]).

no que diz respeito aos erros, é possível classificá-los como: detectados ou não aparentes. Já os defeitos possuem quatro principais ângulos: domínio, detectabilidade, consistência

e severidade [1].

O domínio indica se o defeito está relacionado à interrupção, ao conteúdo e/ou ao *timing* do serviço prestado pelo sistema, sendo que o *timing* está relacionado ao atraso ou adiantamento do serviço. A detectabilidade do defeito é caracterizada pela existência ou não de mecanismos que verifiquem a existência deste defeito. A consistência diz respeito à percepção do defeito pelo usuário, podendo ser consistentes ou inconsistentes.

O defeito consistente é percebido pelos usuários de forma igual. Já o defeito considerado inconsistente, também comumente chamado de Defeito Bizantino, é originado a partir de uma **Falha Bizantina**. O caráter crítico dessa falha motivou este trabalho e um detalhamento mais aprofundado acerca disso é encontrado nas próximas seções.

Por fim, a severidade mede o impacto do defeito no sistema, que pode ser bastante subjetiva e específica para cada sistema. Devido a isso, a metodologia proposta para mensurar esse aspecto é a criação de uma escala específica para cada sistema. Os defeitos também podem surgir na fase de desenvolvimento, ou seja, antes do serviço ser lançado, porém esse caso não será abordado na solução proposta por este trabalho.

2.2.3 Implementação da Tolerância a Falhas

O trabalho de Avizienis et al. (2004) apresenta a tolerância a falhas como uma forma de evitar defeitos, sem que haja a necessidade de intervenção de um agente externo por meio da detecção automática de erros e da recuperação automática do sistema [1]. A detecção automática de erros pode ser feita enquanto o sistema estiver rodando, ou seja, uma detecção concorrente. Além disso, a detecção também pode ser preemptiva. Nesse caso, o sistema é suspenso para que haja uma varredura do mesmo à procura de erros latentes ou de falhas inativas [1].

A recuperação do sistema envolve o tratamento tanto de erros quanto de falhas. O tratamento de erros pode ser feito por meio da reversão de um sistema para um *checkpoint* previamente salvo. Outro mecanismo de tratamento de erro é criar um novo estado para o sistema com base em um estado sem erros. Por último, também há um estado chamado de compensação, na qual existe(m) erro(s), porém as redundâncias do sistema mascaram esse(s) erro(s). Em relação ao tratamento de falhas, é possível compreender quatro mecanismos: (i) identificação dos registros que causam erros, (ii) exclusão física ou lógica dos componentes falhos, (iii) realocação das tarefas que eram realizadas pelo componente excluído e (iv) verificação, atualização e armazenamento da nova configuração do sistema.

2.2.4 A Problemática das Falhas Bizantinas

O termo Falha Bizantina é derivado do Problema dos Generais Bizantinos (PGB) definido por Lamport, Shostak e Pease (1982) [12]. Nesse problema deve-se procurar uma solução que permita uma tomada de decisão distribuída sobre a ação dos generais: se devem atacar, aguardar ou recuar. Essa decisão deve ser tomada em um tempo finito. O grande fator dessa problemática é que existe a possibilidade de que algum dos generais sejam traidores e, portanto, que algumas mensagem sejam perdidas [12, 13].

É importante que os generais leais cheguem a um consenso sobre qual ação devem tomar. Utilizam um mecanismo de consenso, que é definido por um conjunto de passos dados por todos ou pela maioria para chegarem a um acordo sobre a ação. Se metade deles decidir atacar e metade decidir recuar, os exércitos que atacarem serão massacrados e os que recuarem, não serão numerosos o suficiente para levarem a guerra adiante [12, 13].

Os traidores enviam mensagens verdadeiras para não serem desmascarados e, em alguns momentos, enviam mensagem falsas para desestabilizar os exércitos. Trabalham de forma seletiva, escolhendo o momento e os generais para quem desejam enviar mensagens. Os traidores podem, inclusive, operar em conluio para tentar impedir que os generais leais atinjam os seus objetivos. Enquanto isso, os generais leais sempre obedecem o que eles acreditam ser a decisão da maioria [12, 13].

Trazendo o PGB para o contexto de sistemas distribuídos, existe a possibilidade que algum nó do sistema seja malicioso e não trabalhe de forma colaborativa com os demais, atrapalhando o processamento e seu resultado. A falha bizantina pode se apresentar através de um componente com comportamento arbitrário, podendo ser malicioso e ainda envolver colisão com outros componentes que também apresentam falhas [13]. Tanenbaum (2007) e Schneider (1990) concordam que as falhas bizantinas podem ser as mais disruptivas das falhas e também as mais difíceis de se lidar [5, 13]. Portanto, se basear em suposições sobre o comportamento dos componentes pode ser algo perigoso caso estas suposições não sejam satisfeitas, portanto, um sistema prevenido possui Tolerância a Falhas Bizantinas (TFB) [13].

A estratégia normalmente utilizada para a implementação de serviços de tolerância a falhas bizantinas em um sistema distribuído é utilizando Máquina de Estados (ME) e redundância [13, 5]. Produzindo Replicação Máquina de Estados (RME) para mascarar falhas, facilitar a cooperação, sem controle centralizado, e coordenar a interação entre o cliente e as réplicas de servidores. Além disso, a literatura também traz possibilidades de otimizações da RME por meio da paralelização desse mecanismo, ou seja, Replicação Máquina de Estados Paralela (RME paralela). As próximas seções detalham o funcionamento desses dois processos.

2.3 Replicação Máquina de Estados

Um sistema distribuído é composto por clientes e serviços, sendo o serviço uma composição de operações solicitadas pelos clientes. Para implementar um serviço, pode até ser mais simples operar apenas em um servidor (máquina), porém este serviço estará limitado a tolerância de falhas do processador que o executa e se, por ventura, este falhar, o serviço também falhará [13]. Portanto, para garantir que um serviço seja sempre entregue de modo correto, diversas réplicas do mesmo servidor são feitas e operam em processadores distintos dentro do sistema distribuído. Como supracitado na Seção 2.1, cada nó de um sistema distribuído é independente entre si, logo se cada nó for um processador há uma garantia que as falhas do servidor sejam independentes também. A interação entre o cliente e as réplicas é realizada através de certos protocolos de gerenciamento de réplicas [13].

Cada estado da ME é representado por uma variável que codifica seu estado. Um comando possui execução atômica e modifica a variável de estado, mudando o estado atual em que a máquina se encontra e/ou emitindo alguma saída. O comando é ativado através de um pedido do cliente à ME. Tal pedido informa qual ME será usada e também o comando a ser executado, juntamente com quaisquer informações que sejam necessárias para a sua execução [13].

A definição de uma ME está no fato que ela especifica uma computação determinística que lê um fluxo de requisições, processa cada requisição e ocasionalmente produz uma saída. Seu processamento é ordenado de acordo com o recebimento das requisições enviadas pelos clientes. A saída de uma ME é completamente determinada pela sequência de requisições que ela processa, independentemente do tempo e de qualquer outra atividade no sistema. A saída resultante de um processo requisitado pode ser para um atuador, para algum dispositivo periférico ou para clientes que aguardam respostas de requisições prévias [13].

2.3.1 Máquina de Estados Tolerante a Falhas

Um sistema composto por componentes distintos é tolerante a t falhas se ele satisfizer o que foi proposto e não mais que t componentes falhem durante um certo intervalo. Pode ser que o sistema continue a operar corretamente mesmo com mais de t falhas, porém não se pode garantir que ele esteja de fato correto. t é o número máximo de falhas que podem ocorrer com a garantia que o sistema continuará operando corretamente [13]. Uma versão de uma máquina de estados tolerante a t falhas pode ser implementada através da replicação dessa ME.

Executando Replicação Máquina de Estados (RME) em processadores distintos de um sistema distribuído em tal modo que todas as réplicas iniciam no mesmo estado inicial e executam as mesmas requisições, sempre na mesma ordem. Para que apenas um único processador seja afetado na ocorrência de uma falha. Então, combinando a saída de todas as t réplicas é possível obter o resultado para uma ME tolerante a t falhas [13]. Se falhas bizantinas são apresentadas nos processadores, deve-se implementar no mínimo $3t + 1$ réplicas, para que a maioria das saídas permaneçam corretas mesmo que tenham ocorridas t falhas [13].

2.3.2 BFT-SMaRt

A solução proposta neste trabalho foi desenvolvida em cima de RME e TFB, explicito no trabalho de Bessani et al. (2014) [14]. A BFT-SMaRt (do inglês *Byzantine Fault Tolerant (BFT) State Machine Replication (SMR)*) é uma implementação de uma biblioteca robusta de Replicação Máquina de Estados Tolerante a Falhas Bizantinas (RME-TFB), que tem como princípio a alta performance em execuções livres de falhas e uma auto-correção caso réplicas com falhas se apresentem mostrando algum comportamento arbitrário. Além da robustez, essa é a primeira biblioteca RME-TFB que suporta reconfigurações ao conjunto de réplicas e provê suporte transparente e eficiente para serviços duráveis [14]. A BFT-SMaRt foi construída com cinco princípios de design, sendo eles:

1. **Modelo de Falha Sintonizável:** Como diretriz, a BFT-SMaRt sabe lidar com falhas bizantinas não maliciosas. Sendo um modelo realista de sistema em que (1) as mensagens podem ser retardadas, caírem ou até mesmo serem corrompidas e (2) os processos podem executar anormalmente e se entregarem a qualquer ação falsa. Além disso, fornece assinaturas criptográficas para melhorar a tolerância às falhas bizantinas maliciosas e um protocolo simples de RME, para tolerar paradas (*crashes*) e mensagens corrompidas [14].

2. **Simplicidade:** Este princípio foi fundamental para evitar uma complexidade exacerbada no código, permitindo que a sua implementação e o resultado de sua execução se mantenham sempre corretos. A implementação de um algoritmo RME-TFB já é complexa por natureza, então certas técnicas promissoras em termos de performance ou eficiência de recursos não foram utilizadas [14].

3. **Modularidade:** O BFT-SMaRt implementa um protocolo RME modular que usa um consenso primitivo bem definido em seu núcleo. Além dos módulos para comunicação confiável ponto a ponto e de requisições de ordens e consenso de cliente, neste modelo também existem módulos de transferência de estados e reconfiguração independentes do protocolo de acordo [14].

4. **Interface de Programação de Aplicação (API) Simples e Extensível:** Toda a complexidade do RME-TFB está encapsulada dentro de uma simples API que pode ser usada para implementar serviços determinísticos. Seguindo o modelo de programação de RME, os clientes podem usar um comando para invocar um método que envia comandos a todas as réplicas e estas, por sua vez, implementam um método de execução para processar o comando. Se a aplicação exigir comportamentos que não existem dentro desse modelo básico, eles podem ser utilizados através de chamadas alternativas, *callbacks* ou *plugins* em ambos os lados, do cliente e do servidor [14].

5. **Conscientização Multi-Core:** Para melhorar o processamento de tarefas dispendiosas do protocolo, o BFT-SMaRt se utiliza de uma arquitetura onipresente de servidores multi-núcleo. Dessa forma, o rendimento do sistema foi feito para ser escalável com o número de *threads* suportado no hardware das réplicas, principalmente se forem utilizadas assinaturas, pois é necessário mais poder computacional para verificá-las [14].

No BFT-SMaRt é utilizado um modelo de RME-TFB que usa $n >= 3t + 1$ réplicas para tolerar t falhas bizantinas. Como o sistema permite a reconfiguração, é possível alterar os valores de n e t durante a execução, através das operações de *join* e *leave*. Também é possível configurar o sistema para que use apenas $n >= 2t + 1$ réplicas para tolerar t falhas *crash*. Independente da configuração, o sistema necessita de links confiáveis de ponto a ponto para garantir a comunicação entre processos [14]. Em adição a isso, possui três protocolos centrais, para lidar com: ordem total de *multicast*, transferência de estados e reconfiguração.

A Ordem Total de *Multicast* é alcançada usando um protocolo modular chamado Mod-SMaRt, que foi desenvolvido com uma metodologia dirigida a líder e implementa um RME-TFB utilizando um consenso subjacente primitivo. Durante uma execução normal, com ausência de falhas e com a presença de sincronia, a extensão permite que seus clientes enviem suas requisições para todas as réplicas e esperem uma resposta delas. A ordem total é alcançada através de uma sequência de instâncias de consenso, em que cada uma decide um conjunto de requisições de cliente. Cada instância é composta por três passos de comunicação, sendo que o primeiro passo requere que o líder envie uma mensagem de **propósito** às outras réplicas, em seguida dois passos de comunicação todos para todos, formados por **escrita** e **aceitação** de mensagens, são realizados. De tal forma, a mensagem de **propósito** contém o conjunto de requisições que devem ser decididas e as mensagens de **escrita** e **aceitação** contêm apenas o *hash* criptográfico de tal conjunto [14].

Transferência de Estados é importante para não afetar o processamento caso alguma falha venha a ocorrer no sistema. Uma réplica deve ser capaz de se corrigir e se reintegrar ao sistema, sem que haja necessidade de reiniciar todo o serviço replicado. Falhas correlacionadas podem derrubar até t réplicas de uma vez e é necessário utilizar memória estável para recuperar todo o sistema. As técnicas para recuperação de uma réplica da BFT-SMaRt são implementadas em uma camada bem definida entre o protocolo de replicação e a aplicação, sem influenciar o protocolo de consenso. São elas: 1) armazenar lotes de operações em um único disco enquanto estas operações estão sendo executadas pelo serviço, 2) registrar diferentes pontos da execução em réplicas distintas, para evitar a parada do sistema e 3) realizar a transferência do estado de forma colaborativa, com cada réplica enviando diferentes partes do estado para a réplica em recuperação [14]. Metodologia similar ao BitTorrent mencionado na seção 2.1.

Para garantir a Reconfiguração, o BFT-SMaRt permite que réplicas sejam adicionadas ou removidas do sistema durante a execução. Esse protocolo só pode ser iniciado por administradores do sistema executando um cliente chamado *View Manager* que trata falhas bizantinas. É um tipo especial de operação que é emitida e submetida ao algoritmo Mod-SMaRt, como as demais operações de cliente. Através dessas operações, o *View Manager* informa ao sistema sobre a réplica que deve ser adicionada ou retirada do sistema. Devido a ordem mantida no sistema entre as operações, toda réplica não defeituosa passará para o mesmo estado que o sistema atual se encontra. Através de uma assinatura, com chave particular, o cliente pode ter a garantia que a operação foi enviada por um administrador do sistema e esta operação não é enviada à aplicação como as operações comuns. Uma vez que a consistência da assinatura é validada, o estado atual do sistema é atualizado com reconfiguração requisitada.

2.4 Replicação Máquina de Estados Paralela

Tendo em vista esse contexto da RME, é possível notar uma oportunidade de otimização possibilitada por meio do uso paralelo da arquitetura de multi-processadores que os servidores modernos oferecem. No caso do mecanismo clássico de RME, todas as réplicas executam exatamente a mesma sequência de comandos. Já na implementação de Replicação Máquina de Estados Paralela (RME paralela) é possível distribuir, de forma concorrente, os comandos independentes para que o processamento geral se torne mais otimizado. Sendo assim, cabe introduzir o conceito de comandos independentes (não conflitantes): são comandos que acessam variáveis diferentes ou que são apenas de leitura. Enquanto, comandos dependentes (conflitantes) são comandos que acessam uma mesma variável e pelo menos um dos comandos altera o valor da variável em questão.

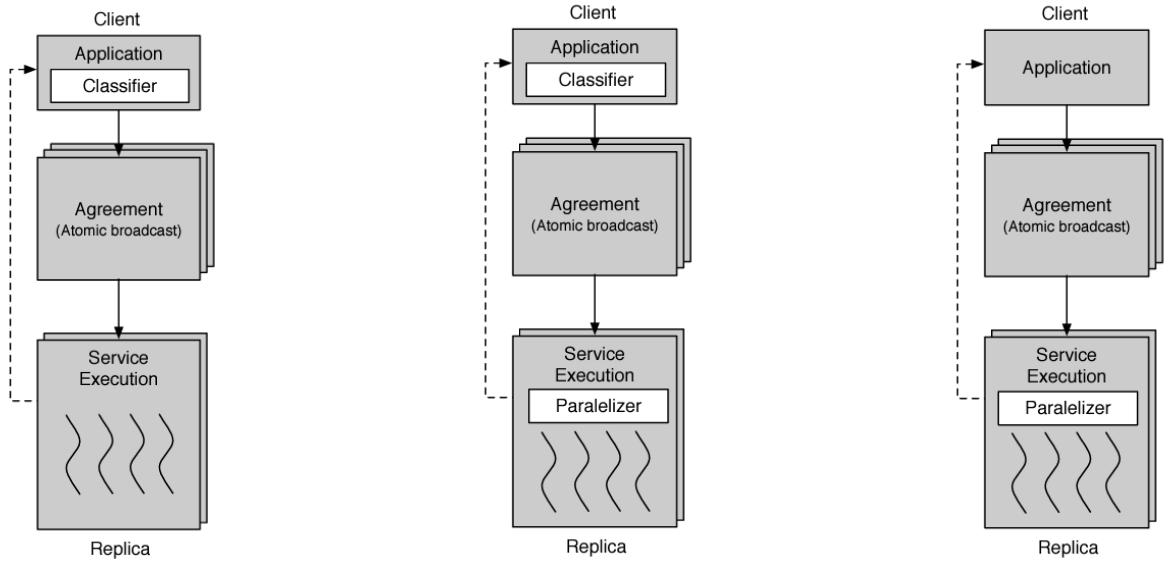
Na leitura acadêmica há diversas propostas de protocolos para permitir a concorrência durante a execução de RME. O fator comum entre elas é a exploração da possibilidade de executar comandos independentes de forma concorrente, pelo fato de não interferirem na operação executada por outros comandos, ou seja, não compartilham qualquer estado ou não modificam estado compartilhado. Comandos com conflitos (dependências) devem ser executados de forma sequencial e seguindo a mesma ordem em todas as réplicas, pois acessam um estado compartilhado e ao menos um modifica este estado. Em contrapartida, a divergência entre as propostas é a forma como executam as dependências entre os comandos [2]. Existem três classes de protocolos, sendo elas:

1. **Protocolos de Escalonamento Estático:** o escalonamento ocorre antes dos comandos serem emitidos, ou seja, antes de chegarem às réplicas e sem alteração posterior na ordem de execução [2].

2. **Protocolos de Escalonamento Antecipado:** os comandos são escalonados antes de serem emitidos, por meio da classificação dos comandos e pré-escolhendo as *threads* operárias que os executarão. Após sua emissão e já nas réplicas, o escalonamento é dinâmico para respeitar as restrições predefinidas [2].

3. **Protocolos de Escalonamento Tardio:** os comandos são escalonados para execução após sua emissão, ou seja, são escalonados dinamicamente nas réplicas. Além da exigência supracitada sobre comandos conflitantes, não há mais restrições ao escalonamento [2].

Todas as classes possuem, prototipicamente, uma estrutura de três camadas, como pode ser visto na Figura 2.5, sendo (1) a primeira do usuário, (2) a segunda dos servidores de aplicação e (3) a terceira das réplicas, que se responsabilizam pelo estado da aplicação. O usuário emite comandos aos servidores de aplicação, que realizam o processamento e os emitem às réplicas [2].



(a) Escalonador Estático

(b) Escalonador Antecipado

(c) Escalonador Tardio

Figura 2.5: Classes de protocolos que implementam RME paralela (Fonte: Alchieri et al. [2]).

Outro conceito importante a ser introduzido é da execução determinística: para o envio das mesmas requisições, sempre será gerada a mesma saída. Sua importância vem do pressuposto que RME deve realizar operações de forma determinística [2]. A paralelização em RMEs paralelas, faz com que nem sempre as mesmas requisições sejam executadas em todas as réplicas (não determinismo), pois comandos sem conflito são paralelizados por entre as réplicas. A partir disso, Alchieri et al. (2018) constatou em seu experimento que os escalonadores determinísticos são menos eficientes que os não determinísticos, apesar da RME exigir determinismo para garantir precisão nos resultados [2].

Escobar et al. (2019) evidencia a má utilização das arquiteturas de multiprocessadores que a RME clássica faz. Devido ao pretexto de seguir uma execução determinística, o processamento se fecha numa situação de monoprocessamento para executar os comandos sequencialmente. A consistência do processamento não é infringida com o multiprocessamento, em que comandos independentes são executados concorrentemente entre si, ainda que isso possa levar ao não determinismo [15].

As propostas que possuem o intuito de extrair o máximo de paralelismo, como o escalonamento tardio (Figura 2.5c) e antecipado (Figura 2.5b), contam com o auxílio de um paralelizador nas réplicas, também chamado de escalonador determinístico. A função desse escalonador é ordenar os comandos de acordo com suas dependências e distribuí-los entre as *threads*. Juntando aqueles que possuem dependências entre si e repartindo a

carga de trabalho entre as *threads* operárias para que todas operem de forma balanceada [2].

2.4.1 Escalonamento Estático

O nome estático vem do fato que o planejamento traçado para o processamento é escolhido e depois não sofre modificações. Isso acontece antes dos comandos serem emitidos. Portanto, não há decisões de escalonamento nas réplicas. Isso diminui as chances de um gargalo e aumenta a exatidão do processo [2].

Com esta abordagem não há necessidade de uma *thread* escalonadora (paralelizador), apenas *threads* operárias. Cada *thread* possui um identificador que serve para assinalar os comandos que deve executar. Isso é determinado pelo usuário no momento da emissão dos comandos, assim como todas as dependências que existem entre os comandos. A ideia é trabalhar com fluxos independentes de comandos que executam concorrentemente, não podendo haver dependência entre estes fluxos [2].

As réplicas possuem n *threads* operárias. Os usuários marcam os comandos que desejam emitir com o identificar de uma ou mais *threads* que os devem executar. A transmissão de ordem total, ou também chamada de atômica, entrega os comandos diretamente às *threads*, respeitando os identificadores previamente marcados. Dois comandos dependentes devem ser executados sequencialmente e assinalados a pelo menos uma *thread* em comum. Se o comando é assinalado a uma única *thread*, então pode ser executado paralelamente a outras *threads*. Todavia, se o comando for assinalado a duas ou mais *threads*, gerando um conflito, as *threads* devem sincronizar para ordenar a execução deste comando em todas elas [2].

2.4.2 Escalonamento Antecipado

O escalonamento de comandos é decidido de antemão nesta abordagem, removendo as decisões de escalonamento das réplicas. Os usuários devem apontar a qual classe cada comando pertence. O escalonador organiza a entrega dos comandos e também as *threads* que os processarão de acordo com as classes. Se existe conflito entre comandos da própria classe, estes devem ser serializados. Com isso em mente, classes conflitantes também devem ser serializadas para respeitar a dependência, enquanto classes independentes podem ser executadas de forma paralela [2].

Apesar dos usuários identificarem as classes dos comandos, o escalonamento final é dinâmico e feito pelo paralelizador nas réplicas. O benefício disso é permitir a redução de sobrecarga de escalonamento, não é necessário coordenar todas as *threads*, apenas o

subconjunto atribuído a cada classe. A probabilidade de gargalos durante o processamento diminui em razão da desnecessidade de um grafo de dependências [2].

As réplicas possuem $n + 1$ *threads*, sendo uma para o escalonador (paralelizador) e as outras n para as *threads* operárias. O escalonador entrega os comandos em ordem total e avisa uma ou mais *threads*. Cada thread operária possui uma fila individual de *inputs* e processa os comandos em ordem *First In, First Out (FIFO)*, ou seja, o primeiro a entrar é o primeiro a sair. Se um comando é escalonado em apenas uma *thread*, então não possui conflito e pode ser executado paralelamente com outros. Caso contrário, as *threads* devem sincronizar para processarem na ordem correta [2].

2.4.3 Escalonamento Tardio

A proposta do escalonador tardio é realizar as decisões de escalonamento dinamicamente nas réplicas, por meio do paralelizador ou *thread* escalonadora. Para tal, ele cria um Grafo Acíclico Dirigido (DAG) (sigla do inglês: *Directed Acyclic Graph*) para organizar as dependências. Cada vértice representa um comando e as arestas simbolizam as interdependências entre os comandos [2].

Utilizar um DAG garante que todos os comandos sejam executados, desde o primeiro inserido no grafo ao último. Isso é possível porquê depois de executar e remover um subconjunto de comandos do grafo, obrigatoriamente outros comandos são liberados para execução [2]. Para garantir a precisão do escalonamento, o paralelizador deve ter acesso exclusivo ao grafo enquanto insere comando, bem como as *threads* operárias enquanto selecionam e executam os comandos [2].

O paralelizador adiciona os vértices e as arestas no grafo quando entrega em ordem total os comandos às *threads* operárias. Estas, por sua vez, removem os vértices do grafo após a execução do comando e de suas dependências. Esse processo acontece de forma determinística. Todavia, Alchieri et al. (2018) notou que rastrear dependências entre comandos na abordagem de escalonamento tardio pode gerar gargalos no escalonamento. Isso acontece com cargas de trabalho consideradas "leves", aqueles em que o esforço computacional para escalonar e executar são semelhantes [2].

Alchieri et al. (2018) deduz que o momento em que as deliberações sobre o escalonamento são feitas está diretamente relacionado à eficiência de cada abordagem. Os autores notaram uma maior paralelização e processamento de dados no escalonador tardio [2]. Devido a isso, essa abordagem foi escolhida como base para o desenvolvimento do presente trabalho. Apesar disso, a proposta desenvolvida pode ser aplicada às outras abordagem de escalonamento também.

2.5 Trabalhos Relacionados

Seguindo a abordagem do escalonador tardio, o trabalho de Escobar et al. (2019) traz a representação das dependências entre os comandos por meio de um DAG, aonde os vértices são os comandos e as arestas são as dependências [15]. Uma aresta vai de um vértice *Vi* para um vértice *Vj* quando os comandos representados por *Vi* e *Vj* são dependentes e há necessidade de execução sequencial, em que a execução do comando *Vi* preceda a execução do comando *Vj*. Com base nessa estrutura de grafo é possível fazer inserções e remoções de vértices com base na demanda de comandos que chega em cada réplica. Em vista disso, é possível pontuar que o rastreamento de dependências pode se tornar uma operação custosa, podendo inclusive causar um gargalo no processamento [15]. Sendo assim, é necessário entender as técnicas mais adequadas para o rastreamento dessas dependências entre os comandos.

Com base nisso, cabe citar que foram criados protocolos para lidar com esse tipo de tarefa, em especial o protocolo CBASE proposto por Kotla et al. (2004) [16]. Esse é um protocolo que se baseia na existência de: (i) *threads* operárias, ou seja, que executam os comandos, e (ii) *threads* escalonadoras, que atribuem os comandos para as *threads* operárias. O CBASE é uma implementação básica do novo tipo de dado abstrato trazido pelo trabalho de Escobar et al. (2019), que é chamado de *Confict-Ordered Set (COS)* [15]. O *COS* é uma generalização das técnicas utilizadas em grafos de dependência: inserção, remoção e acesso aos vértices no grafo.

Ainda de acordo com Escobar et al. (2019), foi percebido que há uma baixa performance por parte das abordagens do tipo granularidade-grossa (e.g.: CBASE), na qual o grafo todo é considerado uma região crítica [15]. De acordo com isso, os autores propuseram duas abordagens do tipo granularidade-fina. A primeira abordagem de granularidade-fina se baseou em *locks* a nível de vértices. A segunda abordagem de granularidade-fina se baseou em um mecanismo *lock-free*, utilizando sincronizações *Nonblocking* relacionadas às operações atômicas: inserção de vértices e atribuição de vértices para a execução. Além disso, também foram utilizadas sincronizações *Lazy* na segunda abordagem, aonde os vértices são removidos apenas de forma lógica inicialmente e somente quando há uma operação de inserção que são removidos fisicamente. A conclusão dos autores após a realização dos experimentos relacionados a essas soluções propostas foi que as abordagens de granularidade-fina também estão sujeitas a um gargalo no processamento, porém, foi possível perceber que na implementação *lock-free* houve uma otimização em relação à quantidade de *threads* operárias [15].

Em um trabalho recente, Pintor et al. (2020) [17] buscou expandir o trabalho de Escobar et al. (2019) [15]. A contribuição proposta pelos autores se dividiu em duas partes: a primeira diz respeito a construção de uma carga de trabalho mais apropriada

para os experimentos da abordagem proposta por Escobar et al. (2019), já a segunda parte é a sugestão de uma implementação análoga a abordagem proposta por Escobar et al. (2019), porém com menor complexidade de código devido ao uso de bibliotecas padrão da linguagem Java.

A carga de trabalho explorada por Pintor et al. (2020) foi desenvolvida com base em um dicionário chave-valor, aonde é possível realizar apenas dois comandos: (i) incrementar o valor associado a uma determinada chave e (ii) ler o valor associado a uma determinada chave. Sendo assim, os comandos conflitantes serão apenas aqueles que atuarem sobre uma mesma chave e pelo menos um deles realiza uma operação de incremento. Nesse contexto, foi implementado um gerador aleatório de comandos com base em uma distribuição Gaussiana [17]. Dessa forma, foi possível aumentar o grau de paralelismo da carga de trabalho, deixando-a mais próxima de situações reais, como, por exemplo, um sistema distribuído de contagem de votos.

A implementação alternativa sugerida por Pintor et al. (2020) baseou-se na abordagem do tipo granularidade-fina e se deu de forma análoga à implementação *lock-free* proposta previamente. Nessa implementação análoga, foram utilizadas as seguintes estruturas de dados concorrentes disponibilizadas pela biblioteca padrão do Java: *CompletableFuture* e *ForkJoinPool*. Essas estruturas foram utilizadas com o intuito de abstrair o gerenciamento e a representação da estrutura do grafo de dependências e, assim, diminuir a complexidade da implementação. Ao final dos experimentos realizados pelos autores, foi concluído que a implementação alternativa se mostrou equivalente à implementação proposta previamente [17].

A proposta apresentada neste trabalho é uma continuação do trabalho realizado por Pintor et al. (2020) [17], que por sua vez é uma continuação de Escobar et al. (2019) [15]. É um projeto desenvolvido em um protótipo de RMEs paralelas, com escalonamento tardio e uma abordagem *escalonador-operários*. O gerenciamento de dependências é feito com o auxílio de um DAG e o rastreio de dependências se dá através de uma abordagem de granularidade-fina com mecanismos *lock-free*. A proposta é melhor detalhada no Capítulo 3, mas de forma resumida: propõe otimizar o desempenho do sistema através de uma nova abordagem no tratamento das dependências do sistemas durante o escalonamento.

2.6 Considerações Finais

O poder de supercomputadores em lidar com tarefas de grande complexidade computacional é conhecido. No entanto, vem crescendo a noção de que se pode atingir o mesmo resultado, por um custo muito menor, com o acoplamento de diversos computadores menores em uma rede distribuída. Cada unidade dentro dessa rede é chamada de nó e age de

forma independente em relação aos demais nós. Todos os nós trabalham de forma colaborativa para atingirem os mesmos objetivos e são gerenciados por um sistema distribuído.

Para que um sistema distribuído seja compensatório deve ter: (i) facilidade de acesso aos recursos, (ii) não estampar a distribuição espacial destes recursos, (iii) ser aberto e (iv) ser escalável. Isso é possível através de alguns princípios de transparência detalhados na Seção 2.1. Dentre as listadas, a transparência a falhas é primordial para o funcionamento correto do sistema, sendo essencial a implementação de um método de tolerância a falhas.

Falhas geram erros que ao se propagarem ao longo do processamento e ao atingirem a interface do sistema, se transformam em defeitos. Um sistema defeituoso é aquele que não cumpre com o proposto e uma forma de evitar isto é usando Replicação Máquina de Estados (RME), em que todas as réplicas executam exatamente a mesma sequência de comandos. Outra forma, mais eficiente, é utilizando Replicação Máquina de Estados Paralela (RME paralela), pois é possível distribuir comandos independentes para serem processados de forma paralela.

Existem várias propostas para permitir o paralelismo durante a execução de RME. Todas exploram possibilidades de executar comandos independentes de forma paralela e diferem na forma como lidam com comandos dependentes. Os comandos são escalonados, para depois serem executados por *n threads* operárias. O escalonamento pode contar ou não com uma *thread* escalonadora. Alchieri et al. (2008) levanta em seu estudo que o momento em que as decisões sobre o escalonamento são feitas está diretamente relacionado à eficiência de cada abordagem. Os autores também notaram que a abordagem de escalonamento tardio apresentou uma maior paralelização entre os dados e um maior processamento também [2].

Dentro da abordagem de escalonamento tardio, a *thread* escalonadora é responsável pelo escalonamento dinâmico nas réplicas. Para rastrear e organizar as interdependências entre os comandos, essa *thread* constrói um Grafo Acíclico Dirigido (DAG), em que os vértices representam os comandos e as arestas direcionadas representam as direções das dependências. As *threads* operárias devem respeitar a ordem de execução das dependências para garantir a consistência e precisão do sistema. Dessa forma, se uma aresta vai de um vértice *Vi* para um vértice *Vj*, quando os comandos representados por *Vi* e *Vj* são dependentes, há necessidade de uma execução sequencial entre eles, de forma que a execução do comando *Vi* preceda a execução do comando *Vj*.

Escobar et al. (2019) propôs algumas abordagens de granularidade para filtrar dependências entre comandos. A abordagem que apresentou melhor otimização em relação à quantidade de *threads* foi uma de granularidade-fina com um mecanismo *lock-free* [15]. Tal mecanismo foi aplicado com sincronizações *Nonblocking* às operações atômicas de inserção dos vértices no grafo e de atribuição deles para a execução. Também foram utilizadas

sincronizações *Lazy* para otimizar o processamento, na qual um vértice só é removido fisicamente do grafo no momento da inserção de um novo comando [15]. Pintor et al. (2020) construiu uma carga de trabalho mais apropriada para os experimentos e apresentou uma implementação alternativa análoga à de Escobar et al. (2019), usando bibliotecas padrão Java que abstraem ao programador a responsabilidade de implementar gerenciar o grafo de dependências [17].

Com base nas referências apresentadas neste capítulo, foi possível desenvolver uma proposta de melhoria no modo de tratamento de comandos conflitantes. A proposta é melhor explicada no Capítulo 3, mas se baseia em um sistema distribuído consistindo de processos interligados que se comunicam por meio de troca de mensagens, com uma tolerância a falhas bizantinas.

Para construir a estrutura deste trabalho e prevenir falhas disruptivas, utilizou-se a biblioteca BFT-SMaRt, descrita na Seção 2.3.1. Essa biblioteca possui um sistema de tolerância às falhas especializada para lidar com falhas bizantinas e garante alta performance e auto-correção das falhas. Para o ampliar a paralelização das requisições, seguiu-se a abordagem de escalonamento tardio, citada na Seção 2.4. Com o uso de um grafo de dependências e uma implementação *lock-free*, citada na Seção 2.5. Ainda vale ressaltar que a proposta desenvolvida neste trabalho também pode ser aplicada às outras abordagens de escalonamento: estático e antecipado.

Capítulo 3

Agrupamento de Requisições Conflitantes

Este capítulo apresenta o projeto proposto, incluindo os algoritmos utilizados e explicando seu funcionamento. O capítulo está organizado em três seções. A Seção 3.1 aborda o cenário preexistente do escalonador tardio usado como base de escalonamento. A Seção 3.2 detalha a proposta de melhoria apresentada por este projeto, ou seja, a proposta de organizar o escalonamento por meio do agrupamento de comandos conflitantes. Por fim, a Seção 3.3 finaliza e conclui as ideias do capítulo.

3.1 Escalonador Tardio

Em busca de uma maior paralelização dos comandos, o escalonador tardio possui uma *thread* escalonadora (chamada de paralelizador) nas réplicas. Responsável pelo escalonamento dinâmico, essa *thread* organiza os comandos de acordo com as suas dependências em um Grafo Acíclico Dirigido (DAG), como mencionado na Seção 2.4.3. Apesar do DAG trazer benefícios para a abordagem, há cenários em que o desempenho cai drasticamente. Isso acontece quando existem muitas dependências, o que aumenta a necessidade de sincronização e como a sincronização acontece nas réplicas, o custo é maior.

Diferentemente do protocolo CBASE proposto por Kotla et al. (2004), mencionado na Seção 2.5, que foi desenvolvido utilizando apenas um *lock* para todo o grafo, a proposta apresentada por Escobar et al. (2019), e seguida por este trabalho, foi elaborada de forma a não utilizar *locks*, ou seja, *lock-free*. Para que isso seja possível, foi necessário utilizar um semáforo com um número certo de permissões para garantir a sincronia de execução entre as *threads*. Nesse cenário, a *thread* escalonadora retira uma permissão do semáforo para inserir um comando no grafo e as *threads* operárias devolvem a permissão após a execução do comando.

Existem dois algoritmos para o funcionamento do escalonador tardio, os Algoritmos 1 e 2 respectivamente. O primeiro é da *thread* escalonadora, também chamada de paralelizador, que organiza os comandos e as dependências, adicionando vértices e arestas no grafo. O segundo é das *threads* operárias que executam os comandos e após isso removem vértices e arestas do grafo. Para evitar que o grafo fique muito grande, a *thread* escalonadora sempre adquire uma permissão do semáforo para inserir uma requisição no grafo e as *threads* operárias devolvem essa permissão ao semáforo após executarem e removerem uma requisição do grafo.

A linha 5 do Algoritmo 1 representa a aquisição de uma permissão do semáforo pela *thread* escalonadora. Caso não exista uma permissão, a *thread* é bloqueada para escalonamento até que uma seja liberada ou ela seja interrompida. Para isso, uma *thread* operária invoca o método da linha 8 do Algoritmo 2 para que uma permissão seja liberada e a *thread* escalonadora volte a executar.

Algoritmo 1 Escalonador tardio

1: **variáveis:**
2: $grafo \leftarrow \emptyset$ // grafo de dependências
3: $semáforo$: com n permissões
4: **na entrega(requisição):**
5: $semáforo.adquirir()$; // pega uma permissão do semáforo
6: $grafo.inserir(requisição)$; // insere a requisição e as dependências

Algoritmo 2 *Threads* operárias para escalonamento tardio

1: **variáveis:**
2: $grafo \leftarrow \emptyset$ // grafo de dependências
3: $semáforo$: com n permissões
4: **enquanto(verdadeiro):**
5: $requisição \leftarrow grafo.próximo()$; // pega a próxima requisição disponível
6: $executar(requisição)$; // executa a requisição
7: $grafo.remover(requisição)$; // remove a requisição e suas dependências do grafo
8: $semáforo.liberar()$; // devolve a permissão do semáforo

Com o Algoritmo 1 (linha 6), todos os comandos são escalonados assim que recebidos, sem distinguir se o comando possui ou não dependências/conflitos. A Figura 3.1 é uma visualização do grafo criado pelo paralelizador, cada vértice é um comando e as arestas direcionadas mostram a direção das dependências entre os comandos. Existem 10 comandos inseridos nesse grafo, sendo eles de leitura (não gera conflito) e escrita (gera

conflito). Os comandos de escrita geram conflito com todos os comandos que são inseridos posteriormente a sua inserção e os comandos de leitura não possuem dependência entre si, apenas com os comandos de escrita. Note que neste exemplo os comandos de leitura podem se interpretados como comandos que apenas leem o estado da aplicação enquanto que os comandos de escrita modificam o estado, e por isso precisam de acesso exclusivo, i.e., conflitam com todos os outros comandos. Os comandos de leitura são representados com a letra **L** na frente, enquanto os comandos de escrita são representados com a letra **E** na frente. Nesse cenário, os comandos foram entregues na seguinte ordem: $E1, L2, E3, L4, L5, L6, E7, E8, L9, L10$.

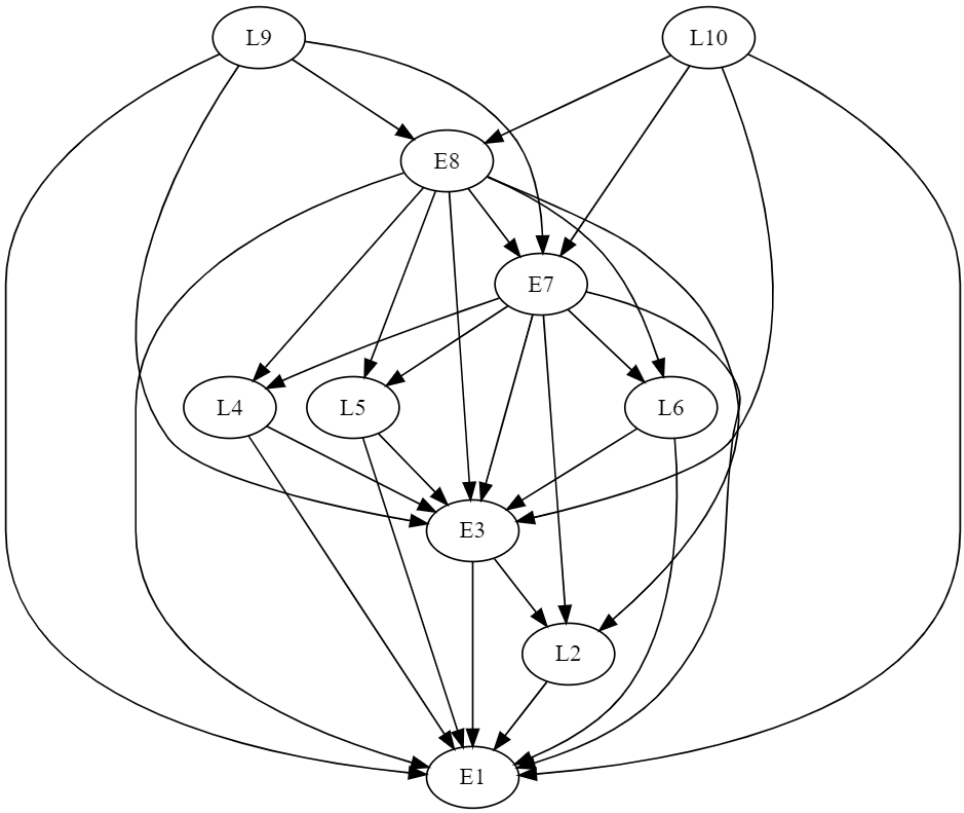


Figura 3.1: Grafo de dependências do escalonador tardio.

Como dito anteriormente, não há distinção entre os comandos conflitantes e não conflitantes, todos são escalonados assim que chegam. Para comandos não conflitantes o escalonamento é direto, porém para comandos com conflitos é necessário procurar todas as dependências e executá-las na ordem de entrega. Após a execução, na linha 7 do Algoritmo 2, os comandos são removidos do grafo. Primeiramente são removidos logicamente do grafo e, posteriormente, durante a inserção de um novo comando, são excluídos fisicamente. Esse procedimento possui a finalidade de poupar um pouco o processamento.

3.2 Proposta de Agrupar Requisições Conflitantes

Uma abordagem de agrupamento de requisições ou comandos conflitantes foi proposta com a intenção de otimizar o processamento do sistema. Nessa proposta, os comandos de leitura que são não conflitantes são escalonados assim que recebidos, como supracitado na Seção 3.1. Já os de escrita são agrupados em uma lista encadeada, porque geram conflitos por causa das dependências. Essa lista é escalonada após atingir o seu tamanho máximo. Embora essa proposta tenha sido construída em cima do escalonamento tardio, também pode ser desenvolvida em outros tipos de escalonamento.

Essa proposta propõe otimizar o processamento porque permite que o sistema funcione em velocidade máxima por um período maior de tempo. Enquanto comandos sem conflitos (leitura) são executados, não há necessidade de sincronização e, portanto, há uma maior agilidade no processamento. Com isso em mente, a busca em réplica pelas dependências desacelera o processamento com a chegada de comandos conflitantes (escrita).

A implementação necessária para o desenvolvimento desta proposta é apresentada nos Algoritmos 3 e 4. O Algoritmo 3 representa a *thread* escalonadora que durante a entrega verifica primeiramente se o comando é de leitura. Caso seja, uma permissão do semáforo é retirada e o comando é adicionado no grafo. Caso seja um comando de escrita, o comando é adicionado a uma lista encadeada ao invés de ser escalonado. Quando essa lista atingir o seu tamanho máximo, então é retirada uma permissão do semáforo para cada requisição da lista e, então, a lista inteira é escalonada em um único vértice do grafo.

O Algoritmo 4 representa o comportamento das *threads* operárias. Estas verificam a quantidade de requisições presente em cada vértice recebido. Caso exista apenas um, significa que é um comando sem conflito (leitura), então o comando é executado, removido do grafo e, por fim, uma permissão é devolvida ao semáforo. Caso exista uma lista de comandos no vértice, uma iteração é realizada na lista: cada comando é executado e após a execução, uma permissão é devolvida ao semáforo. Ao final da iteração, todos os comandos da lista foram executados e suas permissões do semáforo foram liberadas, então o vértice é removido do grafo.

Algoritmo 3 Escalonador tardio com agrupamento de comandos conflitantes

```
1:  variáveis:
2:      grafo ← ∅                                          // grafo de dependências
3:      lista ← ∅                                          // lista encadeada
4:      semáforo: com n permissões
5:      tamanho_max                                        // tamanho máximo para a lista encadeada
6:  na entrega(requisição):
7:      se(requisição.éLeitura()):
8:          semáforo.adquirir();                           // pega uma permissão do semáforo
9:          grafo.inserir(requisição);                     // insere a requisição e as dependências
10:     senão:
11:         lista.adicionar(requisição);                   // adiciona requisição à lista
12:         se(lista.tamanho() == tamanho_max):
13:             semáforo.adquirir(tamanho_max);            // pega as permissões do semáforo
14:             grafo.inserir(lista);                      // insere um conjunto de requisições no grafo
15:         lista.esvaziar();                              // esvaziar a lista para ser usada novamente
```

Algoritmo 4 *Threads* operárias com agrupamento de comandos conflitantes

```
1:  variáveis:
2:      grafo ← ∅                                          // grafo de dependências
3:      semáforo: com n permissões
4:  enquanto(verdadeiro):
5:      no ← grafo.próximo();                              // se não houver próximo, o grafo é bloqueado
6:      se(no.requisições == 1):
7:          executar(no.requisições[0]);                   // executa a requisição do nó
8:          grafo.remover(no);                             // remove nó do grafo
9:          semáforo.liberar();                            // devolve permissão do semáforo
10:     senão:
11:         para cada requisição em no.requisições:        // para cada requisição do nó
12:             executar(requisição);                      // executa a requisição
13:             semáforo.liberar();                        // devolve permissão do semáforo
14:         grafo.remover(no);                             // remove requisição do grafo
```

A Figura 3.2 é uma visualização do grafo montado pela *thread* escalonadora com esta proposta, seguindo o exemplo apresentado na Figura 3.1. Diferentemente da Figura 3.1, nesse grafo os vértices não representam apenas um comando. Os comandos de leitura, que não geram conflitos entre si, ainda são representados individualmente em seu próprio

vértice. Porém, os comandos conflitantes de escrita são agrupados em uma lista e esta lista é adicionada ao vértice. No caso, o tamanho da lista é igual a dois, porém este número pode ser alterado.

Diante desse cenário, ao comparar as Figuras 3.1 e 3.2, pode-se perceber que há uma diminuição no número de vértices e, consequentemente, no número de arestas também. Enquanto na Figura 3.2 existem 10 vértices, um para cada comando entregue pela *thread* escalonadora, na Figura 3.1 há 8 vértices: 6 representando cada comando de leitura e outros 2 vértices para representar os comandos de escrita que, por sua vez, incluem uma lista com 2 comandos de escrita.

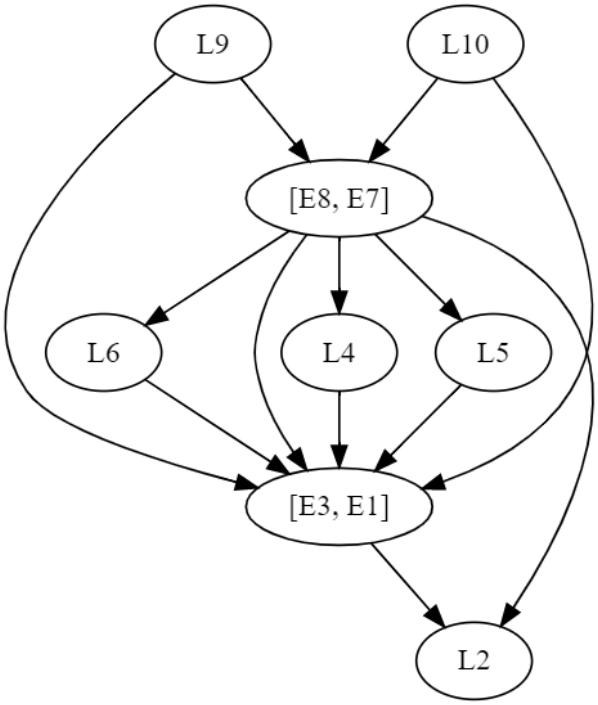


Figura 3.2: Grafo de dependências com o agrupamento de comandos conflitantes.

A linguagem de programação utilizada neste projeto foi Java. Essa decisão é devida às bibliotecas típicas existentes na linguagem que permitem, ao desenvolver, realizar programação concorrente e controlar a utilização de *threads*. Especificamente, as estruturas de dados concorrentes: *CompletableFuture* e *ForkJoinPool* [17, 18]. A primeira auxilia a simplificar o código, pois é responsável pela gerência do grafo, tornando desnecessário que o desenvolvedor precise implementá-lo. Com uma estrutura lógica, cada nó do grafo que possui dependência é uma instância de *CompletableFuture* e a direção das arestas informa qual a sequência que a execução linear deve seguir [17, 18].

A segunda estrutura de dados concorrente é uma versão de implementação da interface *ExecutorService* [17, 18]. Segundo Pintor et al. (2020), a classe *ForkJoinPool* "[...]

implementa um contexto de execução baseado em um conjunto de *threads* com furto de trabalho" [17]. As *threads* executam as tarefas que lhes foram atribuídas ou se apossam de alguma tarefa de outra *thread*. Assim, com esse cenário, há uma maximização do uso do tempo da CPU pela *thread* [17].

3.3 Considerações Finais

A proposta apresentada neste trabalho é uma continuação do trabalho realizado por Pintor et al. (2020) [17], que por sua vez é uma continuação de Escobar et al. (2019) [15]. É um projeto desenvolvido em um protótipo de Replicação Máquinas de Estados Paralelas (RMEs paralelas). Com uma abordagem *escalonador-operários*, em que há uma *thread* paralelizadora responsável pelo escalonamento dinâmico em réplicas e *n threads* operárias responsáveis pela execução dos comandos escalonados.

A abordagem de escalonamento preferida por Escobar et al. (2019) foi o escalonamento tardio [15]. Apesar de ser uma abordagem elegante para lidar com RMEs paralelas, possui a desvantagem de escalonar os comandos assim que chegam, sem verificar se existem ou não conflitos. Pode ser considerada uma desvantagem porque esta abordagem acaba aumentado as sincronizações necessárias para a execução de uma mesma carga de trabalho.

Diante do exposto, surgiu a oportunidade de implementar uma proposta que permitisse um processamento em velocidade máxima por um período maior de tempo. Portanto, agrupar comandos conflitantes em uma lista encadeada e escalonar esta lista posteriormente propõe uma otimização no processamento. Os comandos sem conflitos continuam sendo escalonados assim que chegam, porque não carecem de sincronização e, consequentemente, não impactam a velocidade do processamento.

Esta proposta não alterou a essencialidade do trabalho de Escobar et al. (2019) [15]. Dessa forma, as provas de corretude e precisão que o autor apresentou ainda são válidas. A propriedade de linearizabilidade[1] e o determinismo são mentidos, pois a intenção é apenas de agrupar comandos. A proposta *lock-free* de Pintor et al. (2020) [17] também não foi alterada.

[1] `https://stringfixer.com/pt/Atomic_operations`

Capítulo 4

Experimentos

Este capítulo expõe os experimentos realizados sobre a proposta implementada. Primeiramente, a Seção 4.1 apresentada o ambiente em que os experimentos foram realizados. A Seção 4.2 apresenta a aplicação utilizada para os experimentos. Em seguida, a Seção 4.3 detalha sobre a cargas de trabalho utilizadas na execução de cada experimento. Por fim, a Seção 4.4 demonstra os resultados dos experimentos em forma de gráfico e também há uma análise destes resultados.

4.1 Ambiente Experimental

O ambiente utilizado para a realização dos experimentos foi um ambiente em rede especializado para testes em redes e sistemas distribuídos chamado Emulab [1] [19]. Sua instalação primária é gerenciada pelo Flux Research Group, parte da Escola de Informática da Universidade de Utah. Esse banco de ensaio em rede pode ser utilizado de forma gratuita e permite o estudo, desenvolvimento, depuração e avaliação de sistemas distribuídos [20].

Seu nome Emulab advém do princípio de funcionar como um laboratório para emulação de experimentos. Em cada experimento é possível especificar uma topologia de rede arbitrária. O ambiente oferecido é controlável, previsível e repetível. Permite a implementação, execução e controle de um aplicativo sendo executado em locais diferentes ao redor do mundo. O Emulab unifica os nós espalhados geograficamente em uma interface de usuário comum e os integra em uma estrutura comum. Essa estrutura fornece abstrações, serviços e *namespaces* comuns a todos, como alocação e nomenclatura de nós e links. Ao mapear as abstrações em mecanismos específicos de domínio e nomes internos, o Emulab mascara muito da heterogeneidade dos diferentes recursos ao usuário final.

Apensar do Emulab possibilitar a execução de aplicações distribuídas, as soluções propostas neste trabalho afetam apenas a execução interna de um nó. Por isso, escolhemos

[1] `https://www.emulab.net/`

apenas uma máquina e adicionamos uma fila de comandos ordenados para escalonamento e execução. A máquina escolhida foi a **d430**, com a seguinte especificação de hardware [21]:

- Servidores Dell Poweredge R430 1U
- 2 processadores "Haswell" de 8 núcleos E5-2630 de 2,4 GHz e 64 bits, velocidade de barramento de 8,0 GT/s, cache de 20 MB e suporte a VT (VT-x, EPT e VT-d)
- 64 GB 2133 MT/s DDR4 RAM (8 x módulos de 8 GB)
- 4 NICs Broadcom NetXtreme BCM5720 GbE integradas à placa-mãe (apenas uma em uso)
- 1 NICs PCI-Express Intel I350 1GbE de porta dupla ou 1 porta quádrupla
- 1 NICs PCI-Express Intel X710 10GbE de porta dupla ou 1 porta quádrupla
- 1 SSD Intel SATA de 200 GB, 2 discos SATA de 1 TB e 7200 rpm

Utilizamos o sistema operacional Ubuntu LTS (GNU/Linux 4.15.0-47-generix x86_64) em sua versão 18.04.1. Bem como foi instalado o ambiente de execução do Java, juntamente com sua JVM, nas seguintes especificações:

- OpenJDK Runtime Enviroment (build 11.0.3+7-Ubuntu-1ubuntu218.04.1)
- OpenJDK 64-Bit Server VM (build 11.0.3+7-Ubuntu-1ubuntu218.04.1, mixed mode, sharing)

4.2 Aplicação Utilizada

A aplicação escolhida foi uma lista encadeada de inteiros com operações de leitura e escrita, a primeira $contains(i)$ verifica se i está na lista e a segunda $add(i)$ inclui i na lista. Observe que no modelo de concorrência desta aplicação, as requisições *contains* não conflitam umas com as outras, mas conflitam com as requisições *add*, que conflitam com todas as requisições. A lista foi inicializada com 10.000 entradas, variando de 0 a 9999. O parâmetro inteiro i usado nas operações é sempre uma posição escolhida aleatoriamente na lista. Também foi configurado o tamanho máximo do grafo de dependências com 150 entradas. Nos experimentos, medimos a vazão (ou *throughput*) do sistema. Uma fase de aquecimento precedeu cada experimento.

4.3 Cargas de Trabalho

Dentro do ambiente experimental é utilizando os algoritmos apresentados na Seção 3.2, diversas cargas de trabalho foram utilizadas para testar a proposta deste trabalho. Todos os testes realizados foram executados com e sem a abordagem de agrupar comandos conflitantes. Além disso, o sistema foi submetido a uma alta carga de trabalho para testar a capacidade de esforço computacional. Dessa forma, foram enviados 10.000 comandos por experimento. Vale ainda ressaltar que os experimentos foram executados por 5 minutos.

Como a ideia era testar a eficiência no tratamento de comandos conflitantes, quatro cenários foram testados com quantidades diferentes de comandos de escrita. Esse parâmetro é baseado na quantidade de comandos passados na lista de entrada, mencionado anteriormente. Dessa forma, os testes foram realizados com $0\%, 10\%, 20\%$, e 30% de comandos de escritas. A progressão serve para analisar se há algum gargalo no processamento e analisar as diferenças entre as abordagens.

Lembrando que esta é uma abordagem *escalonador-operários*, em que há uma *thread* escalonadora responsável pela construção do grafo de dependências e n *threads* operárias responsáveis pela execução e, posteriormente, remoção dos comandos do grafo. Testamos o desempenho do sistema com diferentes quantidades de *threads* operárias. O valor de n recebeu como *input* os seguintes os números: 1, 2, 4, 6, 8, 10, 12, 16, 24 e 32.

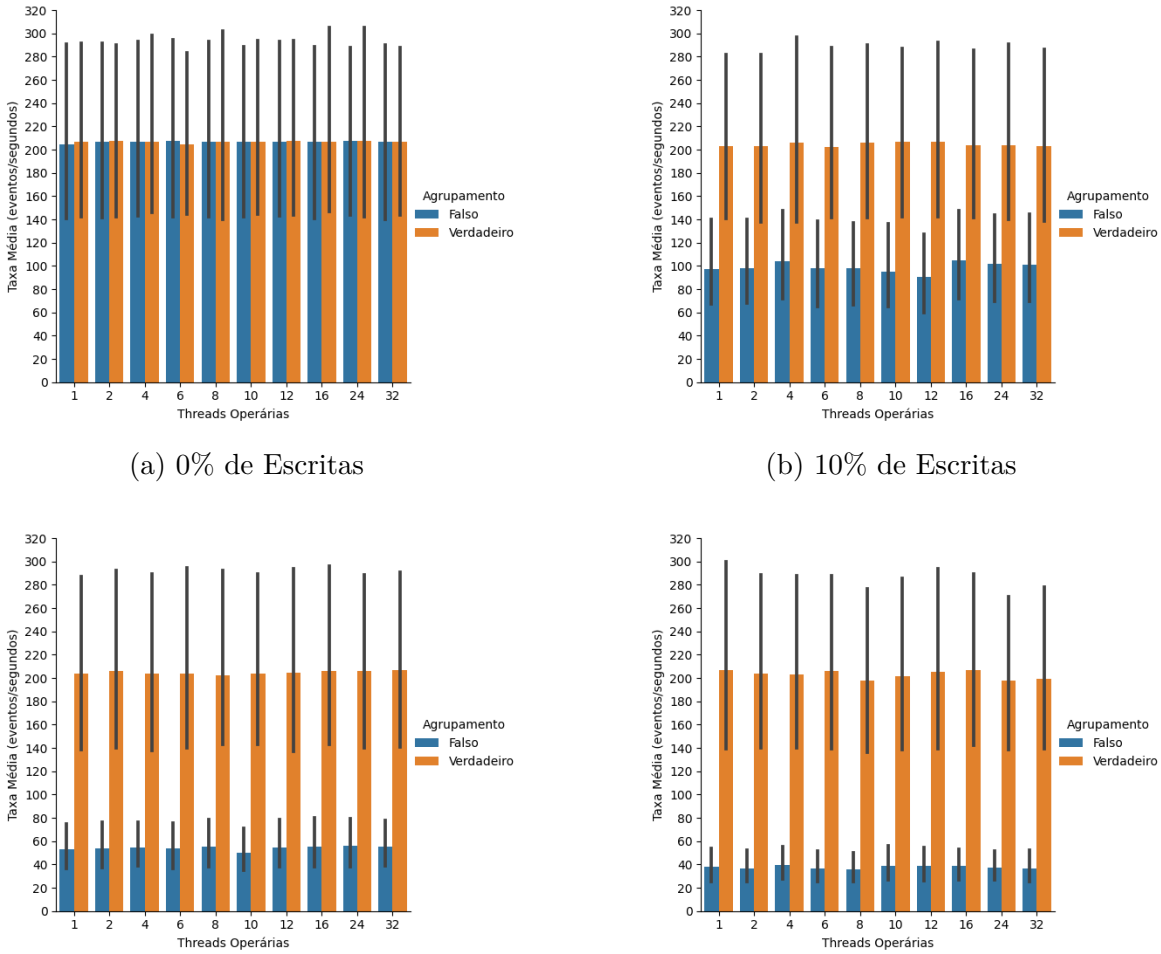
Diante disso, foram realizados um total de 80 experimentos, com variação na quantidade de comandos de escrita e de *threads* operárias. Para cada definição dos parâmetros, foi efetuado um teste sem a proposta de agrupamento e, em seguida, com a proposta levantada por este estudo.

4.4 Resultados

Os resultados de todos os experimentos podem ser vistos na Figura 4.1. A proposta sem agrupamento é identificada com agrupamento igual a falso e com barras na cor azul. Já a proposta com agrupamento de comandos conflitantes é identificada com agrupamento igual a verdadeiro e barras na cor laranja. Como explícito nos gráficos, o eixo X retrata o valor de n para as n *threads* operárias do sistema e o eixo Y retrata a quantidade média de comandos (eventos) executados por segundo.

O gráfico da Figura 4.1a representa 0% de escritas e o resultado foi similar em ambas as abordagens. Isso é justificável pelo fato de não existirem comandos conflitantes nesse experimento, portanto as duas abordagens apresentam o mesmo comportamento. Como todos os comandos são de leitura, são escalonados diretamente no grafo em nós individuais. Com isso em mente, os testes seguintes (Figuras 4.1b, 4.1c e 4.1d) mostram uma diferença

entre as abordagens que está diretamente relacionada com a quantidade de comandos de escrita (conflitantes). Quanto maior a quantidade de escritas, menor é o desempenho da abordagem sem o agrupamento de comandos conflitantes, enquanto a abordagem com agrupamento manteve o desempenho durante os testes.



(a) 0% de Escritas

(b) 10% de Escritas

(c) 20% de Escritas

(d) 30% de Escritas

Figura 4.1: Resultados dos testes

Isso é compreensível pela necessidade de suspender a execução das *threads* operárias toda vez que um comando conflitante aparece, até que todas as suas dependências sejam resolvidas. O tempo necessário para o rastreamento das dependências pode variar em cada caso, ainda assim, nota-se que quanto maior o número de comandos conflitantes (escrita), maior será a quantidade de vezes que a execução será suspensa, impactando diretamente na quantidade de comandos executados durante o experimento.

O rendimento da média de quantos comandos foram executados por segundo não foi afetado pela variação na quantidade de *threads* operárias. Observando a Figura 4.1, é

possível perceber que a quantia média de comandos se manteve com comportamento similar em todos os testes.

A Figura 4.1a é referente aos testes com 0% de escritas, ou seja, 100% de comandos de leitura. Ambas as abordagens tiveram um comportamento homogêneo e executaram em média um pouco mais de 200 comandos por segundo. Esse comportamento se manteve um pouco similar para a proposta de agrupamento dos comandos conflitantes nos testes seguintes. Percebe-se que apenas na Figura 4.1d, com 30% de comandos de escrita e 70% de comandos de leitura, a execução foi um pouco inferior a 200 comandos por segundos em alguns momentos.

Já a proposta sem o agrupamento de comandos conflitantes teve uma queda gradual no rendimento da média de comandos executados. Variou entre um pouco mais e um pouco menos de 100 comandos por segundo na Figura 4.1b, que representa 10% de escritas e 90% de leitura. Na Figura 4.1b caiu para aproximadamente 50 comandos por segundo com 20% de escritas e 80% de leitura. E na Figura 4.1d seu rendimento foi ainda mais baixo, com 30% de escritas foi possível executar um pouco menos de 40 comandos por segundo.

Ao analisar todos os gráficos da Figura 4.1 é perceptível que a abordagem de agrupamento de comandos conflitantes manteve um desempenho consistente em todos os experimentos. Isso demonstra a eficiência do método, pois o rendimento do sistema se manteve mais ou menos estável apesar do aumento de comandos conflitantes.

Capítulo 5

Conclusão

A computação distribuída tornou-se parte do uso cotidiano na Era Digital. A internet, juntamente com seus serviços, não seria possível se não fossem pelas arquiteturas de sistemas distribuídos. Cada pesquisa do Google® envolve computação distribuída com instâncias de fornecedores em todo o mundo trabalhando juntas para gerar resultados de pesquisa correspondentes. Em vista disso, este trabalho desenvolveu uma proposta que melhorasse a eficiência e o desempenho de sistemas distribuídos.

Com este estudo foi possível expandir o conhecimento sobre sistemas distribuídos e, mais especificamente, sobre Replicação Máquina de Estados Paralela (RME paralela) e os tipos de escalonamento. RME paralela supera as limitações de uma Replicação Máquina de Estados (RME) clássica, pois escalonam requisições para serem executadas por um conjunto de *thread* (operárias) de forma que requisições conflitantes são executadas sequencialmente enquanto que as independentes são executadas de forma paralela. A abordagem de escalonamento tardio para RMEs paralelas utiliza um grafo para gerenciar as dependências, em que vértices representam requisições e arestas representam dependências entre requisições, como apresentado no Capítulo 3.

O problema identificado foi que ao aumentar a quantidade de requisições conflitantes, também aumentava a quantidade de arestas (dependências) no grafo e o desempenho do sistema diminuía. Diante disso, este trabalho levantou a hipótese de agrupar requisições conflitantes em um mesmo vértice do grafo, diminuindo a quantidade de arestas e sincronizações, para aumentar o desempenho do sistema. Com a proposta implementada foi possível produzir um grafo de dependências simplificado, apresentado na Figura 3.2, e os resultados apontados pela Figura 4.1. Através de avaliações empíricas, foi possível provar que o tratamento de requisições conflitantes, ou também chamadas de dependentes, está diretamente relacionado ao desempenho do sistema. Assim, analisando os impactos da proposta no sistema, pode-se afirmar que o agrupamento de comandos conflitantes é uma abordagem eficiente para melhorar o desempenho de sistemas distribuídos.

Referências

[1] Avizienis, Algirdas e Laprie, J C e Randell, Brian e Landwehr, Carl: *Basic concepts and taxonomy of dependable and secure computing.* IEEE transactions on dependable and secure computing, 1(1):11–33, 2004. ix, 9, 10, 11, 12, 13

[2] Alchieri, Eduardo e Dotti, Fernando e Marandi, Parisa e Mendizabal, Odorico e Pedone, Fernando: *Boosting state machine replication with concurrent execution.* Em *2018 Eighth Latin-American Symposium on Dependable Computing (LADC)*, páginas 77–86. IEEE, 2018. ix, 19, 20, 21, 22, 25

[3] Google: *Research areas: Distributed systems and parallel computing.* `https://research.google/research-areas/distributed-systems-and-parallel-computing/`. (Accessed on 15/04/2022). 1

[4] Van Steen, Maarten e Tanenbaum, Andrew S: *A brief introduction to distributed systems.* Computing, 98(10):967–1009, 2016. 5, 6, 7, 8, 9

[5] Tanenbaum, Andrew S e Van Steen, Maarten: *Distributed systems: principles and paradigmas.* Person Prentice Hall, Upper Saddle River, NJ, 2007. 5, 9, 10, 14

[6] Lamport, Leslie e Lynch, Nancy: *Distributed computing: Models and methods.* Em *Formal models and semantics*, páginas 1157–1199. Elsevier, 1990. 6

[7] Beschastnikh, Ivan e Liu, Perry e Xing, Albert e Wang, Patty e Brun, Yuriy e Ernst, Michael D.: *Visualizing distributed system executions.* ACM Transactions on Software Engineering and Methodology (TOSEM), 29(2):1–38, 2020. 6, 7

[8] Anthony, Richard John: *Systems Programming: Designing and Developing Distributed Applications.* Elsevier Inc, Waltham, Massachusetts, EUA, 2016. 7, 8

[9] Lamport, Leslei: *Distribution in 'my writings', item 75, mensagem de email enviado ao quadro de avisos da DEC SRC as 12:23:29 PDT em 28 de maio de 1987.* `https://lamport.azurewebsites.net/pubs/distributed-system.txt`, 1987. Acessado em: 2021-06-27. 9

[10] Aurell, Erik: *The physics of distributed information systems.* Em *Journal of Physics: Conference Series*, volume 473, página 012017. IOP Publishing, 2013. 9

[11] Cristian, Flavin: *Understanding fault-tolerant distributed systems.* Communications of the ACM, 34(2):56–78, 1993. 9

[12] Lamport, Leslie e Shostak, R. e PEase M.: *The byzantine generals problem.* Em *Transactions on Programming Languages and Systems (TOPLAS)*, página 382–401. ACM, 1982. 14

[13] Schneider, Fred B: *Implementing fault-tolerant services using the state machine approach: A tutorial.* ACM Computing Surveys (CSUR), 22(4):299–319, 1990. 14, 15, 16

[14] Bessani, Alysson e Sousa, João e Alchieri, Eduardo E.P.: *State machine replication for the masses with bft-smart.* Em *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, páginas 355–362. IEEE, 2014. 16, 17, 18

[15] Escobar, Ian Aragon e Alchieri, Eduardo e Dotti, Fernando Luís e Pedone, Fernando: *Boosting concurrency in parallel state machine replication.* Em *Proceedings of the 20th International Middleware Conference*, páginas 228–240, 2019. 20, 23, 24, 25, 26, 33

[16] Kotla, Ramakrishna e Dahlin, Michael: *High throughput byzantine fault tolerance.* Em *International Conference on Dependable Systems and Networks, 2004*, páginas 575–584. IEEE, 2004. 23

[17] Pintor, Erick e Dotti, Fernando Luís: *Explorando concorrência em aplicações chave-valor com replicação de máquinas de estado.* `https://github.com/erickpintor/parallel-SMR/blob/master/Artigo.pdf`. (Accessed on 07/28/2021). 23, 24, 26, 32, 33

[18] Mircosystems, Sun: *Java documentation*, 1998. 32

[19] Eide, Eric e Ricci, Robert e Van der Merwe, Jacobus (Kobus) e Stoller, Leigh e Webb, Kirk e Duerig, Jon e Wong, Gary e Downie, Keith e Hibler, Mike e Johnson, David: *Emulab.* `https://www.emulab.net/portal/frontpage.php`. (Accessed on 14/04/2022). 34

[20] Eide, Eric e Ricci, Robert e Van der Merwe, Jacobus (Kobus) e Stoller, Leigh e Webb, Kirk e Duerig, Jon e Wong, Gary e Downie, Keith e Hibler, Mike e Johnson, David: *The emulab manual.* `http://docs.emulab.net/`. (Accessed on 14/04/2022). 34

[21] Eide, Eric e Ricci, Robert e Van der Merwe, Jacobus (Kobus) e Stoller, Leigh e Webb, Kirk e Duerig, Jon e Wong, Gary e Downie, Keith e Hibler, Mike e Johnson, David: *Características da máquina d430 do emulab.* `https://gitlab.flux.utah.edu/emulab/emulab-devel/-/wikis/Utah-Cluster/d430s`. (Accessed on 14/04/2022). 35