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

Replicação Máquina de Estados Paralela e Reconfigurável

Alex Lobo de Oliveira Rodrigues

Monografia apresentada como requisito parcial
para conclusão do Curso de Computação — Licenciatura

Orientador
Prof. Dr. Eduardo Adilio Pelinson Alchieri

Brasília
2016

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

Replicação Máquina de Estados Paralela e Reconfigurável

Alex Lobo de Oliveira Rodrigues

Monografia apresentada como requisito parcial
para conclusão do Curso de Computação — Licenciatura

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

Prof. Dr. André Costa Drummond
CIC/UnB

Prof. Dr. Priscila America Solis Mendez
CIC/UnB

Prof. Dr. Pedro Antonio Dourado Rezende
Coordenador do Curso de Computação — Licenciatura

Brasília, 23 de dezembro de 2016

Dedicatória

Eu dedico esse trabalho a minha família.

Agradecimentos

Agradeço primeiramente a Deus, por ter permitido que eu chegasse até aqui, a minha família e amigos pelo apoio e ao professor Eduardo Alchieri pelo seu trabalho como orientador.

Resumo

A Replicação Máquina de Estados (RME) é uma abordagem muito utilizada na implementação de sistemas tolerantes a falhas. Esta técnica consiste em replicar os servidores e fazer com que os mesmos executem deterministicamente, e na mesma ordem, o mesmo conjunto de requisições. Para isso, as requisições devem ser ordenadas e executadas sequencialmente segundo esta ordem em todas as réplicas. Visando melhorar o desempenho do sistema em arquiteturas com múltiplos núcleos, RMEs paralelas tiram proveito da semântica das requisições e permitem a execução paralela de algumas delas, de acordo com um grau de paralelismo pré-definido. Porém, algumas requisições continuam precisando de execução sequencial e impactam negativamente o desempenho do sistema, visto que sincronizações adicionais são necessárias, de acordo com o grau de paralelismo. Este trabalho propõe um protocolo para RME paralela e com grau de paralelismo reconfigurável de acordo com o *workload* atual, visando tirar proveito em situação favoráveis e impactar o mínimo possível em situações desfavoráveis. Experimentos mostram os ganhos advindos com as reconfigurações e ajudam a elucidar o funcionamento deste tipo de sistema.

Palavras-chave: Replicação Máquina de Estados, concorrência, execuções paralelas, protocolo adaptativo

Abstract

State Machine Replication (SMR) is an approach widely used to implementing fault-tolerant systems. In this approach servers are replicated and client requests are deterministically executed in the same order by all replicas. Consequently, client requests must be ordered and sequentially executed by every replica. To improve system performance in multicore systems, parallel SMR allows parallel execution of requests, according to the degree of parallelism definied at startup. However, some requests still need sequential execution, impacting the system performance once additional synchronization is needed, according to the degree of parallelism. This work proposes a protocol for a parallel SMR with the degree of parallelism reconfigurable according to the current workload, with the aim of improve the performance when the workload is favorable and, otherwise, do not impact it. Experiments show the gains due to reconfigurations and shed some light on the behaviour of this kind of system.

Keywords: State Machine Replication, concurrency, parallel executions, adaptive protocol

Sumário

Lista de Figuras

Capítulo 1

Introdução

Inicialmente, as máquinas que posteriormente ganharam nome de computadores eram muito grandes, ocupavam grandes espaços, eram pesados e, além disso, funcionavam de forma independente. Não havia ainda nenhuma forma de comunicação entre as máquinas, muito menos um meio de compartilhar recursos entre elas.

A grande quantidade de melhorias que aconteceram na tecnologia com o passar do tempo permitiu que os computadores se tornassem mais acessíveis, portáteis e mais fortes (*hardwares* mais poderosos), além disso, houve um enorme avanço nas tecnologias de rede, isso permitiu que os computadores pudessem se conectar a grandes distâncias com taxas de transmissão de dados cada vez maiores. Com tudo isso, foi possível montar sistemas compostos por mais de um computador, trazendo também o interesse de desenvolvimento nessa área.

Compartilhar recursos foi a primeira e mais importante motivação para o desenvolvimento de sistemas distribuídos. Aplicações feitas para sistemas centralizados diferem-se bastante das produzidas para sistemas descentralizados, trazendo grandes desafios no desenvolvimento de aplicações para sistemas distribuídos. As empresas que buscaram soluções com aplicações em sistemas distribuídos de maneira interna, hoje procura expandir isso para a rede externa, ou seja, a Internet. A partir desse cenário, percebe-se a presença de uma demanda cada vez maior por novas tecnologias e arquiteturas de redes. Para atender esta demanda as aplicações necessitam seguir um modelo distribuído que leve em consideração à qualidade do serviço. Essas aplicações também devem fornecer segurança no uso de seus serviços e, portanto, torna-se fundamental a aplicação ser tolerante a falhas, ou seja, a garantia que o sistema não irá ser comprometido na presença falhas. O desafio atual das pesquisas é conseguir soluções satisfatórias para tolerância a falhas.

Com toda essa evolução veio também desafios para a Ciência da Computação, dentre eles os relacionados com sistemas distribuídos, tanto em ambientes estáticos quanto dinâmicos. A principal característica dos sistemas distribuídos dinâmicos em relação aos

estáticos é a possibilidade de principalmente processos entrarem e saírem do sistema a qualquer momento. Desta forma, as topologias e composições de processos podem mudar arbitrariamente durante uma execução da aplicação [1], diferentemente dos processos de um sistema estático. Essas mudanças podem ser chamadas de reconfiguração de sistemas.

A Replicação Máquina de Estados (RME) [26, 18], é uma abordagem muito utilizada na implementação de sistemas distribuídos tolerantes a falhas [18, 26, 8]. Basicamente, esta técnica consiste em replicar os servidores e fazer com que os mesmos executem deterministicamente, e na mesma ordem, o mesmo conjunto de operações requisitadas por clientes, fornecendo um serviço de replicação com consistência forte (*linearizability*) [14].

Para manter o determinismo da execução, as operações são ordenadas e executadas sequencialmente segundo esta ordem em todas as réplicas, o que limita o desempenho do sistema principalmente quando consideramos servidores atuais que possuem um *hardware* com múltiplos núcleos, pois apenas um deles seria utilizado para a execução das operações. Com o objetivo de contornar esta limitação, recentemente surgiram abordagens que, tirando proveito da semântica das operações, empregam protocolos que suportam a execução paralela de algumas operações [16, 22, 21, 31, 2].

A ideia básica destas abordagens, chamadas de RME paralelas, é classificar as requisições em dependentes (ou conflitantes) e independentes (ou não conflitantes), de modo que as requisições independentes são executadas em paralelo nas réplicas. Já requisições dependentes devem ser executadas sequencialmente, o que exige alguma sincronização nas réplicas pois nenhuma outra operação pode ser executada paralelamente. Duas requisições são independentes quando acessam diferentes variáveis ou quando apenas leem o valor de uma mesma variável. Por outro lado, duas requisições são dependentes quando acessam pelo menos uma mesma variável e pelo menos uma das requisições altera o valor desta variável.

A quantidade de requisições independentes executadas em paralelo é configurada de acordo com um grau de paralelismo (número de *threads* de execução) definido na inicialização do sistema. Um número elevado aumenta o desempenho em *workloads* com requisições predominantemente independentes, mas impacta negativamente caso a predominância seja de operações dependentes devido a necessidade de sincronizações adicionais. Por outro lado, um número baixo não tira proveito de *workloads* favoráveis (com predominância de requisições independentes) [21, 2]. Desta forma, apesar de ser uma técnica muito promissora, configurar o grau de paralelismo é um problema complexo mesmo conhecendo-se o *workload* a priori, o que na grande maioria das vezes é impossível. Além disso, o *workload* geralmente sofre variações durante a execução do sistema [19].

1.1 Objetivos

Com o objetivo de contornar as limitações anteriormente comentadas, o objetivo geral deste trabalho é o projeto e implementação de um protocolo para RME paralela, que possibilite alterações no grau de paralelismo de acordo com o *workload* atual.

Baseado neste objetivo mais geral, uma série de objetivos específicos são definidos:

1. Fazer uma revisão teórica sobre sistemas distribuídos e temas relacionados;
2. Proposta de um protocolo para RME paralela que melhora os protocolos anteriormente propostos, notadamente [21, 22], por permitir a definição de grupos intermediários de dependências, além dos grupos de dependentes e independentes, aumentando o paralelismo na execução.
3. Proposta de extensão do protocolo anterior para suportar reconfiguração no grau de paralelismo (número de *threads* de execução), possibilitando a adaptação ao *workload* atual, sendo que a principal vantagem desta técnica é que nenhuma informação preliminar se faz necessária;
4. Definir e implementar algumas políticas de reconfiguração;
5. Apresentação e análise de uma série de experimentos realizados com uma implementação dos protocolos propostos, possibilitando uma melhor compreensão a respeito do funcionamento de uma RME paralela bem como dos ganhos advindos com a sua reconfiguração.

1.2 Organização do Texto

A organização deste texto reflete as diversas etapas cumpridas para alcançar os objetivos específicos já listados.

O Capítulo 2 discute conceitos envolvendo sistemas distribuídos e apresenta alguns desafios que surgem a partir de suas características.

O Capítulo 3 apresenta conceitos envolvendo tolerância a falhas em sistemas distribuídos como: falhas, erros, defeitos, dependabilidade e técnicas para se alcançar a dependabilidade. A segunda parte do capítulo ficou responsável por dar uma visão geral de problemas como, consenso e difusão confiável. Por fim uma breve introdução sobre replicação de dados.

O Capítulo 4 discute os conceitos envolvendo a abordagem replicação máquina de estados (RME), apresenta a diferença entre RME paralela e escalável e por fim fala sobre a biblioteca BFT-SMaRt.

O Capítulo 5 apresenta os protocolos para RME paralela, em seguida são mostradas as adaptações efetuadas nesses protocolos para que fosse possível alterar o grau de paralelismo do sistema, além disso, define políticas de reconfiguração e apresenta exemplos de implementação das mesmas.

O Capítulo 6 apresenta os experimentos realizados utilizando as adaptações e o novo protocolo proposto e os resultados obtidos.

Por último, o Capítulo 7 conclui o trabalho, apresentando uma visão geral dos estudos realizados e abordando os possíveis trabalhos futuros.

Capítulo 2

Sistemas Distribuídos

Um sistema distribuído é definido como um conjunto de computadores independentes, que através da troca de mensagens pode processar uma aplicação em diferentes localidades. Os clientes dessas aplicações muitas vezes desconhecem que estão acessando recursos de um sistema descentralizado. Com a viabilidade na construção de sistemas de computação aliado ao enorme avanço nas tecnologias de rede e a grande necessidade de se compartilhar recursos, torna-se essencial o desenvolvimento de sistemas distribuídos.

> "Um sistema no qual componentes de *hardware* ou de *software* localizados em uma rede de computadores se comunicam e coordenam suas ações apenas por passagem de mensagens."
>
> Coulouris

Existem algumas definições para sistemas distribuídos, cada uma trás aspectos importantes, mas nenhuma ainda foi considerada completa, da definição de Couloris levantamos as seguintes caracaterísticas:

- Concorrência: Quando se fala em recurso compartilhados, acaba sendo comum surgir a necessidade de controlar o acesso aos recursos, esse controle acaba gerando situações de concorrência, onde os clientes competem pelo acesso;
- Falta de um relógio global: A única forma de comunicação entre os componentes do sistema é por meio de troca de mensagens, através dessas mensagens eles coordenam suas ações;
- Falhas de componentes independentes: Em sistemas distribuídos onde os serviços são fornecidos de maneira descentralizada, acaba sendo normal algum componente do sistema falhar, essa falha terá efeito local, ou seja, o componente que falhar não influenciará no funcionamento dos outros componentes que podem continuar fornecendo o serviço de forma correta.

Um dos maiores motivos para a construção de sistemas distribuídos advém do compartilhamento de recursos. Os usuários utilizam serviços como: e-mails, redes sociais, armazenamento na nuvem, etc. Tudo isso está tão atrelado no cotidiano dos usuários que muitas das vezes nem percebem que estão utilizando um serviço fornecido por um sistema distribuído.

Um servidor de impressão é um exemplo comum de sistemas distribuídos, o servidor é responsável por gerenciar todos os pedidos de impressão de um determinado setor, possibilita o monitoramento de problemas relacionados as impressoras e/ou identificar a origem dos pedidos que estão gerando problemas.

2.1 Desafios

As características de um sistema distribuído traz consigo alguns desafios para a implementação desse tipo de sistema, tais desafios devem ser enfrentados e superados para se construir um sistema de qualidade, nessa seção serão apresentados alguns desafios, muitos desses desafios já foram resolvidos mas precisam ser conhecidos [15].

2.1.1 Heterogeneidade

Um sistema distribuído pode possuir componentes distintos, seja em *hardware*, sistemas operacionais ou até em linguagens de programação, além disso, diferentes tipos de redes podem estar envolvidas. E para que isso seja possível, desenvolveram-se protocolos, ou padrões, que devem ser usados por pelos componentes.

Um dos meios de tornar isso possível é utilizando o chamado *Middleware* (Figura 2.1), que é uma camada de software usada para tratar essa heterogeneidade. Além de solucionar esse problema ele ainda fornece um modelo computacional para ser seguido por desenvolvedores de serviços e aplicativos distribuídos. Na Figura 2.1 são apresentados quatro computadores utilizando três diferentes aplicações que passam todas pela mesma interface.

Um outro problema vêm da necessidade de se transferir programas executáveis de uma máquina para outra. Quando um programa é gerado em uma máquina, ele irá ser construído com base naquela máquina, para aquele conjunto de instruções e só funcionará na máquina que o gerou. Para solucionar esse problema veio as chamadas máquinas virtuais, um programa executável pode ser executado por qualquer máquina, basta essa máquina ter a máquina virtual na qual o programa foi feito para executar.

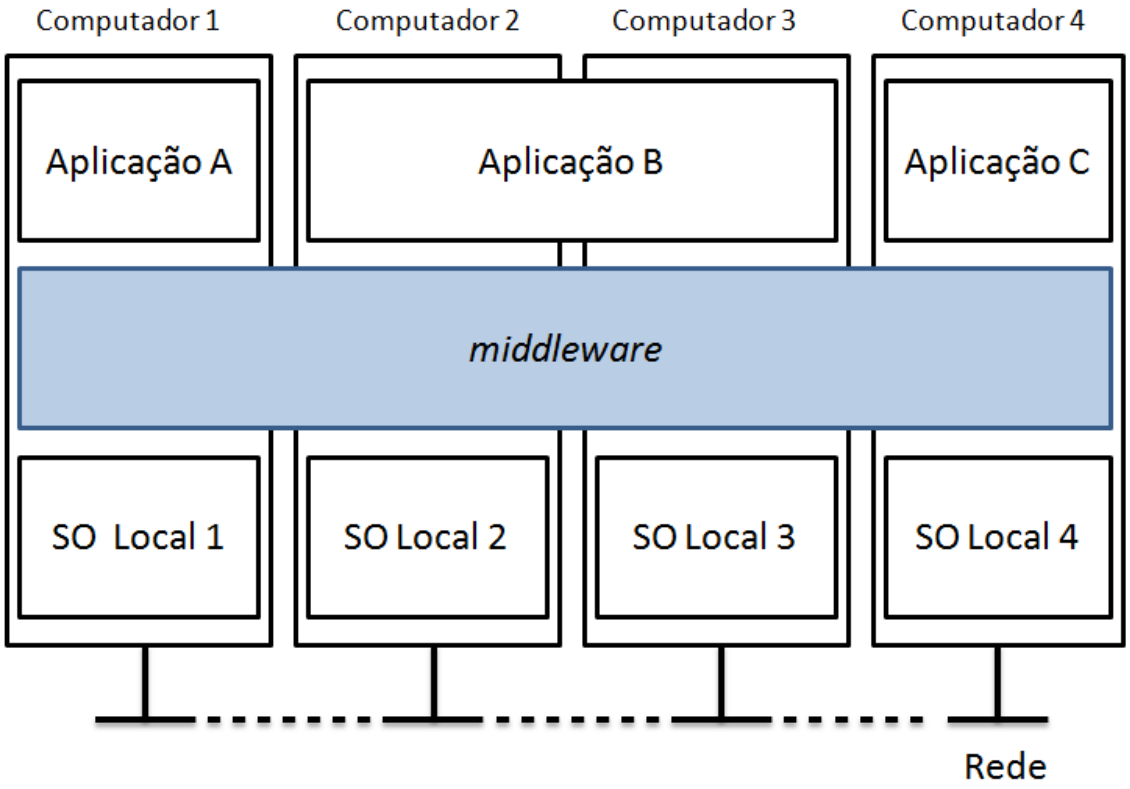


Figura 2.1: Um sistema distribuído organizado como middleware [27].

2.1.2 Sistemas abertos

Essa característica é determinada pela facilidade de se adicionar novos recursos e/ou serviços ao sistema, de maneira que fiquem disponíveis para uma variedade de programas clientes. Para garantir tal característica é necessário que o sistema tenha especificações mostrando as principais interfaces de seus componentes, o segundo passo seria tornar tais especificações acessíveis aos interessados. Diz-se que um sistema computacional é aberto quando ele pode ser estendido e reimplementado de várias maneiras [15].

Entretanto, publicar as interfaces não é a parte mais difícil, o desafio maior vem quando o sistema em questão é de alta complexidade, composto por vários componentes e elaborado por várias pessoas, nesse caso controlar a documentação das interfaces é algo complexo.

2.1.3 Transparência

Existem diferentes tipos de transparência, essencialmente a transparência que o sistema distribuído deve fornecer é a de se apresentar aos clientes um sistema como um todo e

não a coleção de componentes independentes.

São identificadas oito formas de transparência [15], são elas:

- Transparência de acesso: Permite o mesmo tipo de acesso a componentes remotos e locais.
- Transparência de localização: Faz com que os recursos acessados pelos clientes não contenham informações sobre sua localidade;
- Transparência de concorrência: Permite que os usuários operem de forma concorrente sem interferência entre eles;
- Transparência de replicação: Permite que várias instâncias dos recursos sejam usadas sem o conhecimento das réplicas pelos usuários;
- Transparência de falhas: Permite a ocultação de falhas, isso permite que os usuários concluam suas atividades no sistema, mesmo na presença de falhas;
- Transparência de mobilidade: Permite a transferência de recursos para diferentes componentes sem impactar na confiabilidade do sistema;
- Transparência de desempenho: Permite que o sistema seja reconfigurado para atender novas demandas de forma transparente;
- Transparência de escalabilidade: Permite a expansão do sistema sem a interrupção de seus serviços.

Para prover determinados serviços e/ou recursos, um sistema poderá, através do uso de diferentes tipos de transparência, omitir informações que não são necessárias aos seus clientes. As duas transparências mais importantes são a de acesso e a de localização; sua presença ou ausência afeta mais fortemente a utilização de recursos distribuídos.

2.1.4 Segurança

Grande parte dos recursos disponíveis e que são mantidos em sistemas distribuídos têm um alto valor para seus usuários. Por esse motivo a segurança é uma característica essencial. A segurança de informação tem basicamente três componentes, são eles: confidencialidade, integridade e disponibilidade. Esses componentes são responsáveis respectivamente pelo controle de acesso aos recursos, integridade dos dados e por garantir a acessibilidade dos recursos.

2.1.5 Escalabilidade

Os sistemas distribuídos funcionam de forma efetiva e eficaz em diferentes escalas. O significado de escalabilidade nesse contexto seria a capacidade do sistema se manter eficiente com a variação no número de recursos ou de usuários [7]. Existem alguns desafios para garantir essa característica em sistemas distribuídos, são elas:

- Controlar o custo de recursos físicos: à medida que a demanda por recurso aumenta o sistema deve ser capaz de, a baixo custo, se expandir para atender a nova demanda. Supomos que determinado sistema de arquivos esteja sobrecarregado, esse sistema deve ser capaz de ser incrementado com um novo servidor de arquivos e ser capaz de tratar maiores demandas.
- Controlar a perda de desempenho: o sistema poderá apresentar em determinadas situações queda no desempenho, seja pela quantidade de dados ou serviço que o sistema guarda e oferece, então sempre que se for pensar em disponibilizar um serviço é necessário uma análise do custo daquele serviço em diferentes escalas, a complexidade de se oferecer tal serviço deve ser controlada.
- Evitar que os recursos do sistema se esgotem: é necessário que haja uma previsão da quantidade de usuários que irão utilizar os recursos do sistema para evitar que os mesmos se esgotem. Os números utilizados para endereço IP (IPv4), são exemplos de recursos que se esgotaram.
- Evitar gargalos de desempenho: para evitar gargalos no desempenho os algoritmos devem ser descentralizados, quanto mais servidores envolvidos maior será a divisão de serviço entre eles e maior eficiência no atendimento.

Geralmente os problemas relacionados ao aumento de demanda pelos recursos são resolvidos adicionando novos processadores ou até substituindo os atuais por processadores mais fortes. Novos componentes incorporados ao sistema não devem impactar no funcionamento dos componentes já existentes.

2.1.6 Concorrência

A motivação principal de sistemas distribuídos vem da necessidade de se compartilhar recursos, e em uma aplicação que segue essa abordagem é comum nos depararmos com situações onde clientes concorrem pelo acesso de um determinado recurso. Para garantir que não haja inconsistência nos dados é necessário que eles estejam preparados para ambientes concorrentes. Para manter a coerência em ambientes concorrentes, suas operações devem ser sincronizadas de tal maneira que seus dados permaneçam consistentes [15]. Um

exemplo simples seria dois clientes tentando modificar a mesma variável, um deseja efetuar uma operação que aumente o valor da variável e o outro diminuir, se ambos fizerem essas operações ao mesmo tempo não saberemos o resultado final da variável, gerando inconsistência, para controlar essa situação as operações são sincronizadas, primeiro um cliente tem acesso e depois o outro. Dessa forma observamos que os clientes concorrem entre si para poder efetuar suas operações, essa concorrência deve ser, se possível, transparente aos clientes.

2.1.7 Tratamento de Falhas

Em sistemas distribuídos onde os serviços são fornecidos de maneira descentralizada, acaba sendo comum os componentes apresentarem falhas. No capítulo 3 serão apresentados conceitos relacionados a tolerância a falhas em sistemas distribuídos, bem como algumas técnicas para se tolerar falhas.

As falhas em sistemas distribuídos diferentemente do que ocorre normalmente em sistemas centralizados são falhas parciais, ou seja, alguns componentes poderão falhar enquanto outros continuam funcionando.

O sistema deverá ser capaz de identificar e tratar diferentes tipos de falhas garantindo continuidade no correto funcionamento do sistema.

2.2 Conclusão

Neste capítulo concluímos que um sistema distribuído é um conjunto de computadores independentes que trocam mensagens entre si para processar uma aplicação em diferentes localidades, vimos também que existem diferentes definições para sistemas distribuídos, mas que nenhuma delas foi considerada completa.

Foram apresentados também algumas características de sistemas distribuídos e os desafios que surgem na tentativa de se implementar uma aplicação distribuída. Ao fim desse capítulo foi dada uma breve introdução ao tratamento de falhas em sistemas distribuídos, que é um dos principais objetivos para esse tipo de sistema. No próximo capítulo iremos aprofundar em tolerância a falhas em sistemas distribuídos.

Capítulo 3

Tolerância a Falhas em Sistemas Distribuídos

A capacidade de tolerar falhas é um dos principais objetivos em sistemas distribuídos. Qualquer sistema está suscetível a falhas. Supõe-se que os sistemas funcionem apropriadamente, que os seus recursos estejam sempre disponíveis, mas alcançar tais objetivos é uma tarefa complexa. Em sistemas distribuídos deve-se prover o serviço de forma continuada com uma pequena queda de desempenho na presença de falhas[29]. A tolerância a falha nada mais é do que a capacidade de o sistema continuar operando mesmo na presença de falhas.

3.1 Conceitos Básicos

3.1.1 Falhas, Erros e Defeitos

Um defeito é definido como um desvio da especificação do sistema.Um sistema apresenta falhas quando seus serviços não estão em conformidade com a especificação ou por que a especificação não descreve adequadamente suas funções. Podemos ver um serviço como uma sequência de estados no sistema, quando um desses estados desviam do serviço correto podemos chamar esse estado de um erro.

As falhas podem ser classificadas como transientes, intermitente e permanente. Falhas do tipo transientes ocorrem apenas uma vez e somem, acontecem devido a algum mal funcionamento temporário, seja um componente de *hardware* que deixou de transimitir sinal durante um curto período de tempo ou algum outro caso. Já falhas intermitentes ocorrem repetidamente em pequenos intervalos de tempo, podendo desaparecer e reaparecer aleatoriamente. Por fim, falhas permanentes só serão resolvidas com a subistituição daquele componente onde esse tipo de falha ocorreu.

Modelo de Falhas em Sistemas Distribuídos

Existem diferentes tipos de falhas, elas podem ser classificadas segundo uma taxonomia proposta em [13] que as distingue, são elas:

- Falhas por *crash*: Quando não há resposta por parte do sistema perante a uma solicitação de serviço válida;
- Falhas de temporização: Nesse caso, as respostas do servidor chegam em tempo arbitrário;
- Falhas de resposta: Quando existem inconsistências nas respostas dadas pelo servidor;
- Falhas bizantinas: Também conhecida como falhas arbitrárias, quando o sistema age de forma arbitrária, um servidor por exemplo pode estar produzindo respostas que nunca deveriam ser produzidas e mesmo assim são dadas como corretas, ou até mesmo quando um servidor tem comportamento malicioso, um processo bizantino pode assumir a identidade de outro, enviar mensagens incorretas ou simplismente não enviar.

3.1.2 Dependabilidade

O objetivo de se utilzar técnicas de tolerância a falhas é alcançar dependabilidade [29]. Uma propriedade que define a capacidade dos sistemas de prestar um serviço que se pode confiar.

Dependabilidade é um conceito integrador que engloba os seguintes atributos básicos [28]:

- Disponibilidade: Tornar os recursos acessíveis;
- Confiabilidade: Fornecimento de serviços corretos de acordo com a especificação do sistema;
- Segurança: A capacidade do sistema operar normalmente sem oferecer riscos ao usuário ou até mesmo ao sistema.
- Confidencialidade: Garante que não haverá divulgação não autorizada de informações;
- Integridade: Garante que não haverá alterações impróprias no estado do sistema;
- Manutenabilidade: A facilidade de se efetuar reparos e manutenção no sistema.

Dependendo da aplicação a ser desenvolvida, pode-se dar uma ênfase a atributos específicos. Geralmente segurança, disponibilidade e confiabilidade são atributos são atributos considerados de essencial importância.

3.1.3 Meios para Alcançar a Dependabilidade

Após ter o conhecimento dos mais comuns tipos de falhas e seus conceitos, é importante conhecer algumas técnicas para o tratamento de falhas, são elas [7]:

- Detecção de falhas: Algumas falhas podem ser detectadas, como por exemplo em troca de mensagens pode-se utilizar um *bit* de paridade para poder detectar a falha e posteriormente tomar alguma ação corretiva. O desafio é gerenciar a ocorrência de falhas que não são detectáveis.
- Mascaramento de falhas: Existe a possibilidade de mascarar ou minimizar o impacto de algumas falhas detectáveis. Um exemplo disso seria forçar a retransmissão de mensagens, isso pode ser feito ocultando uma mensagem ou excluindo alguma mensagem corrompida para minimizar a gravidade da falha.
- Tolerância a falhas: Capacidade do sistema se manter funcionando corretamente mesmo na presença de falhas. Um sistema composto por um conjunto de componentes tem uma grande possibilidade de ocorrer falhas, essas falhas não devem influenciar na corretude dos serviços oferecidos pelo sistema.
- Recuperação de falhas: Os sistemas devem ser projetados considerando a possibilidade de haver falhas, para que possa estar pronto para se recuperar delas, podendo recuperar os dados (perdidos ou corrompidos) ou retroceder a um estado anterior por exemplo.
- Redundância: Para tornar sistemas distribuídos capazes de tolerar falhas, pode-se utilizar componentes redundantes em sua composição. Por exemplo, caso um servidor contendo um banco de dados apresente uma falha total, um outro componente do conjunto de réplicas poderá continuar fornecendo os recursos. Dependendo da quantidade de réplicas, pode-se tolerar diferentes tipos de falhas.

3.2 Consenso

Em um sistema distribuído, formado por vários processos independentes, o problema do consenso consiste em fazer com que todos os processos corretos acabem por decidir o mesmo valor, o qual deve ter sido previamente proposto por algum dos processos do sistema [25]. Formalmente, este problema é definido em termos de duas primitivas [13]:

- *propose(G,v)*: o valor v é proposto ao conjunto de processos G.
- *decide(v)*: executado pelo protocolo de consenso para notificar ao(s) interessado(s) que v é o valor decidido.

Estas primitivas devem satisfazer as seguintes propriedades de segurança (*safety*) e vivacidade (*liveness*) [5, 4]:

- **Acordo**: Se um processo correto decide v, então todos os processos corretos terminam por decidir v.
- **Validade**: Um processo correto decide v somente se v foi previamente proposto por algum processo.
- **Terminação**: Todos os processos corretos terminam por decidir.

A propriedade de acordo garante que todos os processos corretos decidem o mesmo valor. A validade relaciona o valor decidido com os valores propostos e sua alteração dá origem a outros tipos de consensos. As propriedades de acordo e validade definem os requisitos de segurança do consenso, já a propriedade de terminação define o requisito de vivacidade deste problema [25].

Um dos procedimentos para alcançar consenso sob falhas arbitrárias em sistemas síncronos é conhecido como o algoritmo dos generais bizantinos [17]. [17] também mostrou que consenso é impossível em sistemas assíncronos com falhas arbitrárias. Posteriormente [12] provou que consenso é impossível em sistemas assíncronos considerando qualquer modelo de falhas.

3.3 Difusão Atômica

O problema da difusão atômica [13], também conhecido como difusão com ordem total, consiste em fazer com que todos os processos corretos, membros de um grupo, entreguem todas as mensagens difundidas neste grupo na mesma ordem [25].

A difusão atômica é definida sobre duas primitivas básicas [13]:

- A-*multicast(G,m)* ou TO-*multicast(G,m)*: utilizada para difundir a mensagem m para todos os processos pertencentes ao grupo G.
- A-*deliver(p,m)* ou TO-*deliver(p,m)*: chamada pelo protocolo de difusão atômica para entregar à aplicação a mensagem m, difundida pelo processo p.

Formalmente, um protocolo de difusão atômica deve satisfazer as seguintes propriedades [13]:

- **Validade**: Se um processo correto difundiu m no grupo G, então algum processo correto pertencente à G terminará por entregar m ou nenhum processo pertencente à G está correto;
- **Acordo**: Se um processo correto pertencente a um grupo G entregar a mensagem m, então todos os processos corretos pertencentes a G acabarão por entregar m;
- **Integridade**: Para qualquer mensagem m, cada processo correto pertencente ao grupo G entrega m no máximo uma vez e somente se m foi previamente difundida em G (pelo emissor de m – *sender(m)*);
- **Ordem Total Local**: Se dois processos corretos p e q entregam as mensagens m e m' difundidas no grupo G, então p entrega m antes de m' se e somente se q entregar m antes de m'.

Quando consideramos que os processos estão sujeitos a faltas bizantinas, a propriedade de integridade precisa ser reformulada [13]:

- **Integridade**: Para qualquer mensagem m, cada processo correto pertencente ao grupo G entrega m no máximo uma vez e somente se o emissor de m *(sender(m))* é correto, então m foi previamente difundida em G.

Com esta definição, a propriedade de integridade se refere apenas a difusão (*multicast*) e entrega (*deliver*) de mensagens por processos corretos. No entanto, é difícil diferenciar um processo correto de um processo faltoso que está se comportando corretamente em relação ao protocolo [25].

Diversos procedimentos de tolerância a falhas baseiam-se em difusão confiável. Sempre que votação e consenso são necessários, por exemplo, difusão confiável pode ser aplicada.

3.4 Conclusão

Neste capitulo concluímos que um dos principais objetivos de um sistema distribuído é a capacidade de tolerar falhas, alcançar esse objetivo é o mesmo que alcançar a dependabilidade. A dependabilidade é a capacidade de o sistema fornecer serviços de maneira correta. Foram apresentados também neste capítulo meios para se alcançar a dependabilidade, problemas comuns em sistemas distribuídos e conceitos fundamentais.

No próximo capítulo iremos apresentar a replicação máquina de estados, que é uma das abordagens mais utilizadas em sistemas tolerantes a falhas.

Capítulo 4

Replicação Máquina de Estados

A Replicação Máquina de Estados (RME) [26] é uma abordagem muito utilizada na implementação de sistemas tolerantes a falhas, ela consiste em replicar os servidores e coordenar as interações entre os clientes e as réplicas, com o intuito de que as várias réplicas apresentem a mesma evolução em seus estados. São necessário *2f + 1* ou *3f + 1* réplicas para tolerar até *f* falhas por *crash* ou bizantinas, respectivamente [2].

Para que as réplicas apresentem a mesma evolução em seus estados, é necessário que: (i) partindo de um mesmo estado inicial e (ii) executando o mesmo conjunto de requisições na mesma ordem, (iii) todas as réplicas cheguem ao mesmo estado final, definindo o determinismo de réplicas [2]. Para prover o item (i) basta iniciar todas as réplicas com o mesmo estado, procedimento que pode não ser trivial se considerarmos a possibilidade de recuperação de réplicas faltosas [6].

Já garantir o item (ii) envolve a utilização de um protocolo de difusão atômica (Figura 4.1). O problema da difusão atômica [13] faz com que todas as réplicas corretas entreguem todas as requisições na mesma ordem. Por fim, para prover o item (iii), é necessário que as operações executadas pelas réplicas sejam deterministas, i.e., que a execução de uma mesma operação produza a mesma mudança de estado e tenha o mesmo retorno nas diversas réplicas do sistema.

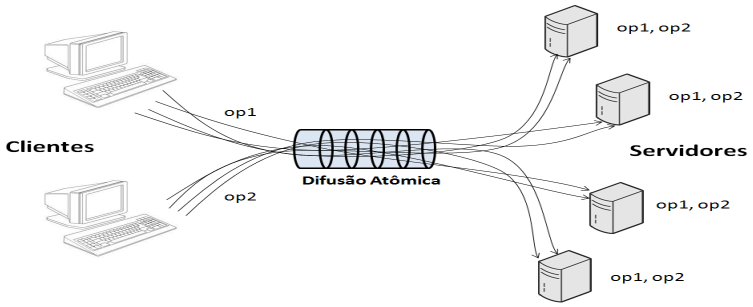
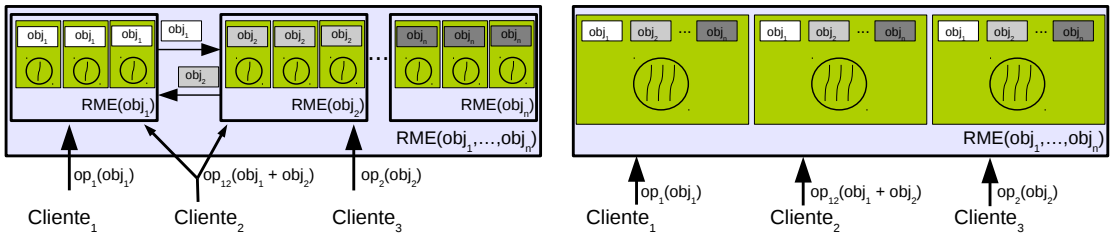


Figura 4.1: Replicação máquina de estados [2].

Um aspecto que recentemente começou a ser explorado é a otimização dos procedimentos necessários para a execução das requisições depois de ordenadas, modificando a visão sobre a forma de como o item (iii) pode ser atendido. Estas abordagens tiram proveito da semântica das operações para (1) dividir o estado da aplicação entre várias RMEs (RME escaláveis – Figura 4.2(a)) [19, 11, 19, 10] ou (2) executá-las paralelamente nas réplicas (RME Paralelas – Figura 4.2(b)) [16, 22, 21, 23, 2].



(a) RME Escalável.

(b) RME Paralela.

Figura 4.2: Abordagens para aumentar o desempenho de uma RME.

RME Escaláveis

A ideia básica desta abordagem é dividir o estado da aplicação entre várias RMEs (ou partições) que executam em paralelo (Figura 4.2(a)). As operações são mapeadas para a partição correspondente e o desempenho é melhorado visto que tanto a ordenação quanto a execução de operações enviadas para RMEs diferentes são executadas em paralelo. No entanto, uma operação que precisa acessar o estado de mais de uma partição deve ser enviada para as várias partições envolvidas e os objetos (estado) devem ser transferidos entre as partições, a fim de que uma delas execute a operação e as outras apenas aguardem esta execução [11]. Com o objetivo de evitar a execução de operações que envolvem várias partições e a consequente queda de desempenho, o estado da aplicação pode ser redistribuído entre as partições [19] ou ainda partições podem ser criadas ou removidas [9].

RME Paralelas

Em uma RME paralela, cada réplica possui o estado completo da aplicação como em uma RME tradicional, mas operações que acessam partes diferentes do estado (ou partições) são executadas em paralelo através de um conjunto de *threads* de execução (Figura 4.2(b)). Para isso, duas soluções foram propostas: *(1)* partes do estado da aplicação são associadas a diferentes *threads* e quando uma operação envolve o estado manipulado por várias *threads*, uma delas executa e as outras apenas aguardam [22, 2]; ou *(2)* um grafo de dependências é criado e as *threads* buscam requisições independentes neste grafo para

execução [16]. Como podemos perceber, os efeitos negativos causados pela execução de operações que envolvem várias partições tendem a ser menores nesta abordagem visto que não é necessária a transferência de objetos entre partições. Como nas RMEs escaláveis, qualquer técnica utilizada para melhorar o desempenho do protocolo de ordenação pode ser empregada, como a ordenação em lotes [6] ou a execução paralela de várias instâncias do protocolo de ordenação [21].

4.1 BFT-SMaRt

O BFT-SMaRt [6] representa a concretização de uma replicação Máquina de Estados [26] tolerante tanto a falhas por *crash* quanto bizantinas. Esta biblioteca *open-source* de replicação foi desenvolvida na linguagem de programação Java e implementa um protocolo similar aos outros protocolos para tolerância a falhas (ex.: [8] para tolerar falhas bizantinas ou [24], [20] para tolerar apenas *crashes*). Atualmente, o BFT-SMaRt implementa protocolos para reconfiguração, *checkpoints* e transferência de estados, tornando-se assim uma biblioteca completa para RME, a qual foi desenvolvida seguindo os seguintes princípios [6, 2].

1. *Modelo de falhas configurável* – é possível configurar o sistema para tolerar falhas bizantinas ou apenas *crashes*;
2. *Simplicidade* – não usa otimizações que aumentem a complexidade da implementação;
3. *Modularidade* – o BFT-SMaRt foi projetado de forma modular, apresentando uma notável separação entre os protocolos implementados (consenso, difusão atômica, *checkpoints*, transferência de estado e reconfiguração);
4. *API simples e extensível* – toda a complexidade de uma RME é encapsulada em uma API simples que pode ser estendida para implementação de uma aplicação mais complexa; e
5. *Proveito de arquiteturas multi-core* – como veremos adiante, a arquitetura do BFT-SMaRt ja se aproveita de arquiteturas com múltiplos núcleos para otimizar a ordenação sequencial de requisições e nos procedimentos de execução.

O BFT-SMaRt assume um modelo de sistema usual para Replicação Maquina de Estados [8], [6]: $n \geq 3f + 1$ (resp. $n \geq 2f + 1$) servidores para tolerar f falhas bizantinas (resp. *crashes*); um numero ilimitado de clientes que podem falhar; e um sistema parcialmente síncrono para garantir terminação. Através de reconfigurações [3], tanto n quanto f podem

sofrer alterações durante a execução. O sistema ainda necessita de canais ponto-a-ponto confiáveis para comunicação, que são implementados usando MACs (*message authentication codes*) sobre o TCP/IP. Além disso, pode-se configurar os clientes para assinar suas requisições, garantindo-se autenticação [2].

Arquitetura Básica do BFT-SMaRt

Podemos dividir, de uma forma geral, a arquitetura de cada réplica em três grandes partes: recebimento, ordenamento e execução de requisições [2]. Na Figura 4.3 temos uma visão geral dessa arquitetura.

Recebimento de Requisições. Os clientes enviam suas requisições ao sistema, essas requisições são replicadas e enviadas para cada réplica do conjunto responsável pelo atendimento da requisição, nessas réplicas, uma *thread* é inicializada para cada cliente, as requisições de cada cliente são colocadas em fila separada. A autenticidade das requisições é garantida por meio de assinaturas digitais, i.e., os clientes assinam suas requisições. Assim é garantido que qualquer réplica seja capaz de verificar a autenticidade das requisições e aceitar apenas as que são válidas.

Ordenamento de Requisições. Sempre que existirem requisições para serem executadas, uma *thread proposer* iniciara uma instância do consenso para definir uma ordem de entrega para um lote de requisições. Durante este processo, uma réplica se comunica com as outras através de outro conjunto de *threads*. Além disso, existe um conjunto de *threads* responsável por receber as mensagens enviadas pelas outras réplicas. Todo o processamento necessário para garantir a autenticidade destas mensagens e executado nestas *threads*. No BFT-SMaRt a ordenação é sequencial, i.e., uma nova instância do consenso só é inicializada após a instância anterior ter terminado. A ordenação de requisições em lotes visa aumentar o desempenho do sistema [2].

Caso uma requisição não seja ordenada dentro de um determinado tempo, o sistema troca a réplica líder. Um tempo limite para ordenação é associado a cada requisição r recebida em cada replica i. Caso este tempo se esgotar, i envia r para todas as réplicas e define um novo tempo para sua ordenação. Isto garante que todas as réplicas recebam r, pois um cliente malicioso pode enviar r apenas para alguma(s) replica(s), tentando forçar uma troca de líder. Caso este tempo se esgotar novamente, i solicita a troca de líder, que apenas é executada após $f + 1$ réplicas solicitarem esta troca, impedindo que uma réplica maliciosa force trocas de líder.

Execução de Requisições. Quando a ordem de execução de um lote de requisições é definida, este lote é adicionado em uma fila para então ser entregue a aplicação por uma *thread* de entrega. Após o processamento da cada requisição, uma resposta é enviada ao cliente que à enviou. O cliente, por sua vez, determina que uma resposta para sua

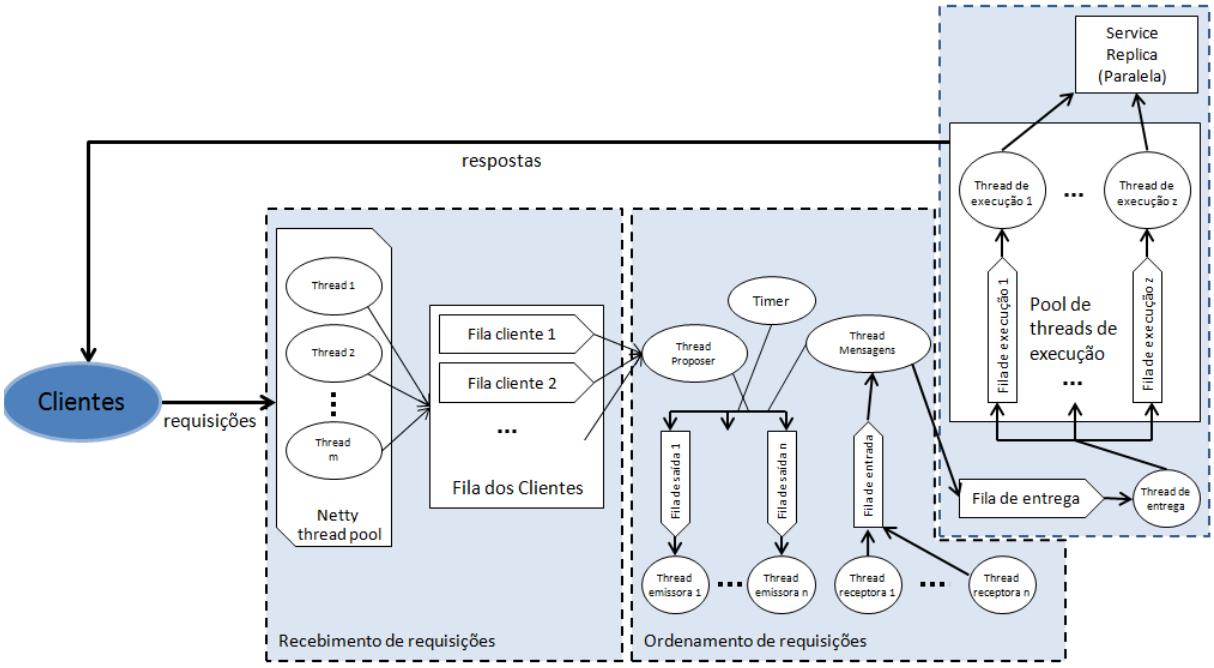


Figura 4.3: Arquitetura de uma Réplica do BFT-SMaRt [2].

requisição é válida se ele receber $f + 1$ respostas iguais (ou apenas *1* resposta caso o sistema esteja configurado para tolerar apenas falhas por *crash*), garantindo que pelo menos uma réplica correta obteve tal resposta. Atualmente foi proposto um protocolo para paralelizar essa parte de execução, tornando possível executar múltiplas requisições em paralelo para obter maior desempenho. A Figura 4.3 apresenta a arquitetura do BFT-SMaRt, com as múltiplas (*pool*) *threads* de execução já utilizando esse protocolo.

4.2 Conclusão

Neste capítulo concluímos que a RME é uma das abordagens mais utilizadas para sistemas tolerantes a falhas, dependendo da quantidade de réplicas disponíveis pode-se tolerar falhas tanto por parada quanto bizantinas, essa abordagem consiste em replicar os servidores e coordenar as interações entre os clientes e estas réplicas, com o objetivo de que as mesmas apresentem a mesma evolução nos seus estados. Foram apresentadas abordagens utilizadas para aumentar o desempenho de uma RME e por fim o BFT-SMaRt, que representa a concretização de uma RME com suporte para execuções paralelas.

No próximo capítulo será apresentada uma nova proposta de protocolo para execuções paralelas e as adaptações feitas nessse protocolo para que ele suportasse alterações no grau de paralelismo.

Capítulo 5

Replicação Máquina de Estados Paralela e Reconfigurável

Neste capítulo será apresenta uma proposta para uma RME paralela e reconfigurável. Primeiramente discutimos o protocolo para execuções paralelas, o qual faz uso de um escalonador (*scheduler*) que atribui as requisições para as *threads* de execução. Posteriormente, apresentamos os protocolos que permitem a reconfiguração do número de *threads* ativas de acordo com políticas de reconfiguração que podem ser definidas pelos usuários.

5.1 Paralelismo

Esta seção apresenta o protocolo para execuções paralelas de requisições em uma réplica, requisito de uma RME paralela (Figura 4.2(b)). O protocolo aqui apresentado é independente do protocolo de ordenação, o qual também pode executar várias instâncias em paralelo (ordenação em paralelo) [21] e/ou ordenar lotes de requisições em uma única instância (ordenação em lotes) [6].

A ideia básica do protocolo é dividir o estado da aplicação entre um conjunto de *threads* de execução, de forma que cada *thread* acesse a sua partição do estado para executar operações em paralelo. Sempre que uma operação envolve o estado de várias *threads*, uma delas acessa todo o estado necessário para executar a operação e as outras *threads* envolvidas aguardam esta execução, garantindo as propriedades de uma RME [11, 21].

Esta divisão do estado define grupos de execução (ou de dependências) e o cliente deve informar para qual grupo de execução sua requisição é endereçada. Em nosso protocolo é possível criar qualquer grupo de execução necessário para a aplicação (basta configurar as réplicas com o identificador do grupo e as *threads* que o compõem), mas os seguintes grupos são definidos por padrão: *(i)* `CONFLICT_ALL` – contém todas as *threads* e deve ser usado para realizar operações que acessam todo o estado da aplicação; *(ii)* `CONFLICT_NONE`

– usado para realizar operações que podem executar em paralelo com qualquer outra operação (geralmente operações que não modificam o estado), por qualquer *thread*; *(iii)* cada *thread* ainda possui um grupo com seu próprio identificador (as *threads* são identificadas de forma incremental começando em 0) que deve ser usado para operações que acessam apenas o estado associado à respectiva *thread* (este mapeamento é definido pelo cliente, i.e., operações que acessam uma mesma partição do estado devem sempre ser enviadas para a mesma *thread*).

Algoritmo 1 Algoritmo de distribuição das requisições (*thread scheduler* [2]).

```
variables: Variables and sets used by the scheduler.
    numThreads ← the number of worker threads
    queues[numThreads] ← the queues used to assign requests to the threads (see Algorithm 2)
    nextThread ← 0                  // id of the thread that will execute the next CONFLICT_NONE request
on initialization:
 1) start the barrier for the CONFLICT_ALL group with number of parties equal to numThreads
 2) start the barriers for other created conflict groups
on A-deliver(request):
 3) if request.groupId == CONFLICT_NONE then
 4)     queues[nextThread].put(request)                          //assigns the request to a thread...
 5)     nextThread = (nextThread + 1)%numThreads                 //...using a roud-robin policy
 6) else if request.groupId == CONFLICT_ALL then
 7)     for i = 0; i < numThreads; i++ do                        //assigns the request to all threads
 8)         queues[i].put(request)
 9)     end for
10) else if request.groupId < numThreads then                   //request directly sent to some thread
11)     queues[request.groupId].put(request)
12) else                                                        //request to a created conflict group
13)     group_queues ← get the queues of the threads in request.groupId
14)     for each q ∈ group_queues do                            //assigns the request to the threads in the group
15)         q.put(request)
16)     end for
17) end if
```

Escalonador

Sempre que uma requisição é entregue pelo protocolo de ordenação, o escalonador é executado (por uma outra *thread*) para atribuir uma determinada requisição a uma ou mais *threads* de execução (Algoritmo 1). A comunicação entre o escalonador e as *threads* de execução ocorre através de uma fila sincronizada, de acordo com o identificador do grupo de execução conforme segue:

- *CONFLICT_NONE* – a requisição é atribuida a uma única *thread* de execução, seguindo a política *round-robin* (linhas 3-5).
- *CONFLICT_ALL* – a requisição é atribuida a todas as *threads* (linhas 6-9).

- Caso a requisição seja endereçada para o grupo de uma *thread* específica, a mesma é atribuída para tal *thread* (linhas 10-12).
- Finalmente, caso a requisição seja endereçada para um grupo formado por mais de uma *thread*, a requisição é encaminhada para as *threads* do grupo (linhas 13-16).

Threads Executoras

Sempre que existir uma requisição disponível para ser executada, cada *thread* procede da seguinte forma, de acordo com o grupo ao qual a requisição foi endereçada (Algoritmo 2):

- *CONFLICT_NONE* ou seu próprio grupo – apenas executa a operação, pois nenhuma sincronização com outras *threads* é necessária (linhas 3-4).
- *CONFLICT_ALL* ou outro grupo – neste caso é necessário que ocorra uma sincronização entre as *threads* do grupo (linhas 5-14), de forma que primeiro é necessário esperar até que todas as *threads* cheguem a este ponto da execução (primeiro acesso à barreira – linhas 7 e 11) para então uma delas executar a requisição (linha 6) enquanto que as outras apenas aguardam por isso (segundo acesso à barreira – linhas 9 e 12). A função $getBarrier(request.groupId).await()$ sinaliza que uma determinada *thread* atingiu a barreira do grupo $request.groupId$.

Algoritmo 2 Algoritmo de execução de requisições (*threads* executoras).

variables: Variables and sets used by each worker thread.
$myId \leftarrow$ id received at initialization // thread *id* (the ids range from 0 to $maxThreads - 1$)
$queue \leftarrow$ a synchronized/blocking queue that contains the requests to be executed by this thread
on thread run:
1) **while** true **do**
2) $request \leftarrow queue.take()$ //get the next request to be executed, blocks until a request be available
3) **if** $request.groupId ==$ `CONFLICT_NONE` $\vee$ $req.groupId == myId$ **then** // no conflict
4) executes the request against the application state and sends the reply to the client
5) **else** //conflict: $request.groupId ==$ `CONFLICT_ALL` or some other created conflict group
6) **if** $myId == executor\ for\ request.groupId$ **then** // the thread with the smallest id
7) $getBarrier(request.groupId).await()$ // waits the other threads to stop
8) executes the request against the application state and sends the reply to the client
9) $getBarrier(request.groupId).await()$ // resumes the other threads execution
10) **else**
11) $getBarrier(request.groupId).await()$ //signalizes that will wait for the execution
12) $getBarrier(request.groupId).await()$ //waits the execution
13) **end if**
14) **end if**
15) **end while**

Execuções

Essa seção tem como objetivo simular e apresentar a execução de um conjunto de requisições utilizando o protocolo para execuções paralelas em uma RME.

Na primeira parte (Figura 5.1) temos a representação de uma fila de entrega contendo o conjunto de requisições a serem executadas, essas requisições já passaram pelo processo de ordenação, podemos observar também a existência de uma fila de execução para cada *thread* executora, nessa simulação utilizamos apenas quatro *threads* para execução e temos uma *thread* responsável pelo escalonamento das requisições que é chamada de escalonador.

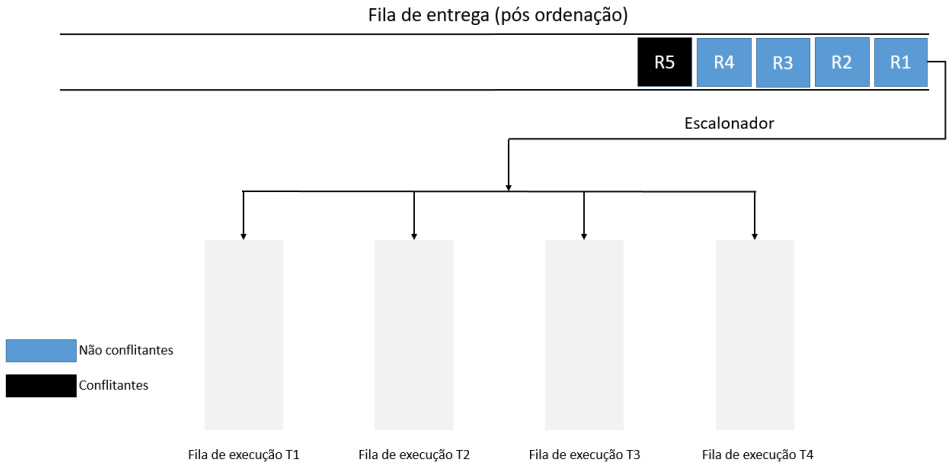


Figura 5.1: Execuções RME paralela parte 1

No segundo momento da simulação (Figura 5.2) consideramos que o escalonador distribuiu as quatro primeiras requisições pelas filas de execução utilizando a política *round-robin*. Na terceira parte, a requisição a ser distribuída é do tipo conflitante, nesse caso ela deverá ser distribuída por todas as *threads* como mostrado na Figura 5.3.

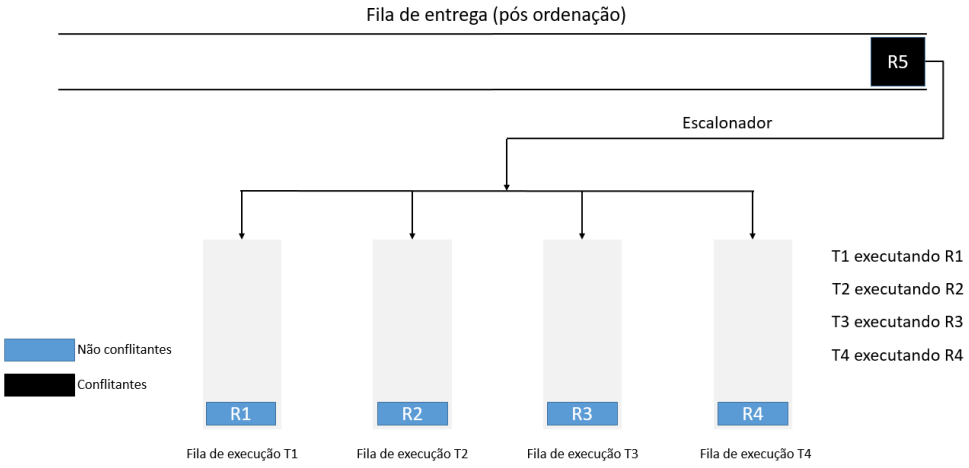


Figura 5.2: Execuções RME paralela parte 2

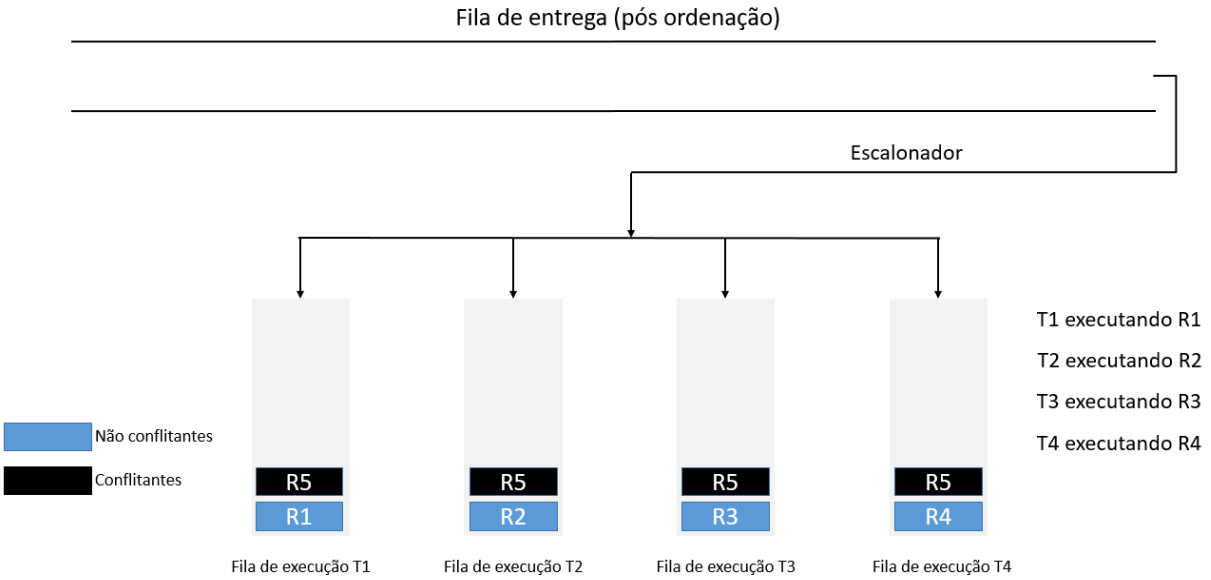


Figura 5.3: Execuções RME paralela parte 3

Na parte quatro (Figura 5.4) consideramos que as *threads* terminaram de executar suas requisições ao mesmo tempo (R1, R2, R3 e R4), em seguida irão executar a requisição do tipo conflitante, segundo o protocolo apenas uma *thread* poderá executar esse tipo de requisição enquanto as demais aguardam, essas requisições envolvem todo o estado da aplicação por isso são tratadas dessa forma, a T1 é escolhida para executar esse tipo de requisição nesse exemplo. Após a execução de R5 por T1 se encerra a simulação.

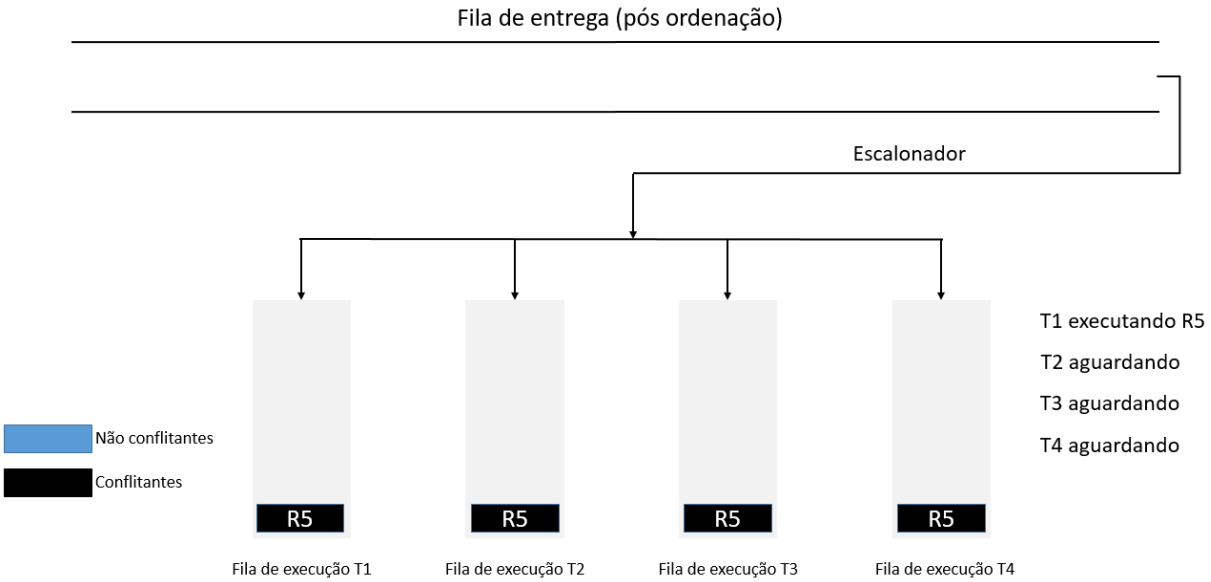


Figura 5.4: Execuções RME paralela parte 4

Podemos observar nessa simulação que quanto maior o número de requisições não conflitantes maior será o ganho utilizando-se um alto grau de paralelismo. Por outro lado, observamos o esforço para se executar uma requisição do tipo conflitante, nesse exemplo as *threads* tiveram que aguardar T1 executar apenas uma requisição para poder continuar, mas poderiam ser mais requisições aumentando assim o tempo de ociosidade das *threads* que precisam aguardar.

5.2 Reconfiguração

Apesar do protocolo da seção anterior implementar uma RME paralela, o número de *threads* de execução, que define o grau de paralelismo, é definido de forma estática na inicialização do sistema. Esta definição é muito importante e afeta diretamente o desempenho do sistema [22, 2]. Um número grande de *threads* tende a aumentar o desempenho em *workloads* com uma grande percentagem de requisições não conflitantes (`CONFLICT_NONE`), mas o desempenho diminui abruptamente com o aumento da quantidade de requisições conflitantes (`CONFLICT_ALL`). Por outro lado, um número pequeno de *threads* não faz com que o sistema seja capaz de atingir o desempenho máximo para *workloads* com poucas requisições conflitantes. O desempenho máximo do sistema ainda é limitado pelo *hardware* utilizado.

Devido ao fato de o *workload* poder mudar durante a execução, e dada as restrições anteriormente descritas, definir um número ideal para a quantidade de *threads* de execução é uma tarefa bastante difícil, se não impossível, e ao mesmo tempo fundamental para o desempenho do sistema. Neste sentido, esta seção apresenta um protocolo de reconfiguração para uma RME paralela, de forma que o número de *threads* pode sofrer alterações durante a execução da aplicação para tentar se adaptar ao *workload* atual.

A ideia básica deste protocolo é dividir as *threads* em ativas e inativas. Uma *thread* é ativa caso esteja apta a executar requisições, caso contrário é considerada inativa. Somente as *threads* ativas participam dos protocolos da RME paralela e, com isso, afetam o desempenho do sistema. Desta forma, a reconfiguração ocorre através da ativação e desativação de *threads*. Na inicialização do sistema, além de especificar o número inicial de *threads* ativas, o usuário também deve definir os números mínimo e máximo de *threads* que podem estar ativas ao mesmo tempo e fornecer uma política com as regras a serem seguidas para ativação e/ou desativação de *threads*.

Escalonador

Sempre que uma requisição é entregue pelo protocolo de ordenação o escalonador e executado (Algoritmo 3). Primeiramente, a política de reconfiguração é acessada para

verificar a necessidade de reconfiguração (linha 4), a qual deve retornar o número de *threads* a serem ativadas (retorno positivo) ou desativadas (retorno negativo).

Algoritmo 3 Algoritmo reconfigurável de distribuição das requisições (*thread scheduler*).

```
variables: Variables and sets used by the scheduler.
    minThreads ← the minimum number of active worker threads
    maxThreads ← the maximum number of active worker threads
    currentThreads ← the current number of active worker threads
    queues[maxThreads] ← the queues used to assign requests to the threads (see Algorithm 4)
    nextThread ← 0                  // id of the thread that will execute the next CONFLICT_NONE request
on initialization:
 1) start the barrier for CONFLICT_ALL group with number of parties equal to currentThreads
 2) start the barriers for other created conflict groups
 3) start the barrier reconfig_barrier for RECONFIG with number of parties equal to maxThreads
on A-deliver(request):
 4) recNum ← reconfigPolicy(request, minThreads, currentThreads, maxThreads)
 5) if recNum ≠ 0 ∧ (minThreads ≤ currentThreads + recNum ≤ maxThreads) then
 6)     currentThreads ← currentThreads + recNum
 7)     nextThread ← 0
 8)     reconfig_request.groupId ← RECONFIG
 9)     for i = 0; i < maxThreads; i++ do                    //assigns the request to all threads
10)         queues[i].put(reconfig_request)
11)     end for
12) end if
13) if request.groupId == CONFLICT_NONE then
14)     queues[nextThread].put(request)                      //assigns the request to a thread...
15)     nextThread = (nextThread + 1)%currentThreads         //...using a roud-robin policy
16) else if request.groupId == CONFLICT_ALL then
17)     for i = 0; i < currentThreads; i++ do            //assigns the request to all active threads
18)         queues[i].put(request)
19)     end for
20) else if request.groupId < maxThreads then            //request directly sent to some thread
21)     if request.groupId < currentThreads then
22)         queues[request.groupId].put(request)
23)     else
24)         request.groupId ← CONFLICT_ALL               //handles the request as CONFLICT_ALL
25)         for i = 0; i < currentThreads; i++ do        //assigns the request to all active threads
26)             queues[i].put(request)
27)         end for
28)     end if
29) else                                                 //request to a created conflict group
30)     if some thread belonging to request.groupId is not active then
31)         request.groupId ← CONFLICT_ALL               //handles the request as CONFLICT_ALL
32)         for i = 0; i < currentThreads; i++ do        //assigns the request to all active threads
33)             queues[i].put(request)
34)         end for
35)     else
36)         group_queues ← get the queues of the threads in request.groupId
37)         for each q ∈ group_queues do                 //assigns the request to the threads in the group
38)             q.put(request)
39)         end for
40)     end if
41) end if
```

As *threads* ativadas/desativadas são escolhidas com base nos seus identificadores, a *thread* inativa de menor identificardor será escolhida para ativação e a *thread* de maior identificador dentre as *threads* ativas será escolhida para desativação. Caso seja necessário reconfigurar o sistema (linhas 5-11), uma requisição especial (*RECONFIG*) é adicionada na fila de todas as *threads* (até mesmo as inativas, que também participam da reconfiguração) e o escalonador passa a atribuir requisições para as *threads* ativas ou a nao atribuir mais requisições para as *threads* inativas. Desta forma, para ativar ou desativar uma *thread* basta começar a adicionar ou parar de adicionar requisições em sua fila, respectivamente. A reconfiguração propriamente dita ocorre quando todas as *threads* atingem o mesmo ponto da execução onde esta requisição especial é executada.

O restante deste protocolo é semelhante ao do escalonador estático (Algoritmo 1). A principal diferença é que apenas as *threads* ativas são consideradas na distribuição das requisições e caso uma requisição seja endereçada para uma *thread* inativa ou para um grupo que a contenha, a mesma e considerada como uma requisição que conflita com todas (*CONFLICT_ALL*) pois de outro modo não seria executada. Requisições endereçadas para grupos que possuem alguma *thread* inativa poderiam ser executadas pelo próprio grupo caso o mesmo fosse reconfigurado (sua barreira reconfigurada para o novo número de *threads* ativas pertencentes ao grupo).

Threads Executoras

A principal diferença para o modelo estático está na execução de reconfigurações (Algoritmo 4), as quais são tratadas de forma semelhante às requisições do grupo `CONFLICT_ALL`, i.e., sem paralelismo. Durante a execução de uma reconfiguração, a barreira do grupo `CONFLICT_ALL` é reconfigurada de acordo com a nova configuração do sistema (linha 6). Todas as *threads* devem participar desta execução pois a barreira utilizada para reconfigurações ($reconfig\_barrier$) deve ser estática (não pode ser reconfigurada em meio a execução da reconfiguração). Como é esperado que a quantidade de reconfigurações seja proporcionalmente muito menor do que a quantidade de requisições de clientes, isso não afeta o desempenho do sistema.

Esta abordagem funciona pelo fato de que até a determinação da reconfiguração pelo escalonador, todas as *threads* ativas na configuração antiga receberam as requisições e portanto a barreira com a configuração antiga é utilizada até que todas cheguem ao ponto da execução da reconfiguração. A partir da determinação da reconfiguração, o escalonador passa a atribuir requisições apenas para as *threads* ativas na nova configuração e, do ponto da execução da reconfiguração em diante, a barreira estará reconfigurada para refletir este número atual de *threads* ativas.

Algoritmo 4 Algoritmo reconfigurável de execução de requisições (*threads* executoras).

variables: Variables and sets used by each worker thread.
$myId \leftarrow$ id received at initialization // thread *id* (the ids range from 0 to $maxThreads - 1$)
$queue \leftarrow$ a synchronized/blocking queue that contains the requests to be executed by this thread
on thread run:
1) **while** true **do**
2) $request \leftarrow queue.take()$ //get the next request to be executed, blocks until a request be available
3) **if** $request.groupId ==$ `RECONFIG` **then** // thread reconfiguration
4) **if** $myId == 0$ **then**
5) $reconfig\_barrier.await()$ //waits all the other threads to stop
6) reconfigure the barrier for `CONFLICT_ALL` with number of parties equal to $currentThreads$
7) $reconfig\_barrier.await()$ //resumes all the other threads
8) **else**
9) $reconfig\_barrier.await()$ //signalizes that will wait for the reconfiguration
10) $reconfig\_barrier.await()$ //waits the reconfiguration execution
11) **end if**
12) **end if**
13) lines 3–14 of Algorithm 2
14) **end while**

Execuções

Essa seção tem como objetivo simular e apresentar a execução de requisições utilizando o protocolo aqui proposto para execuções paralelas e reconfigurável.

O ambiente é o mesmo apresentado na simulação do protocolo de execuções paralelas na seção 5.1, a diferença é que foi inserido uma nova legenda de cor verde para requisições do tipo reconfiguração.

Na primeira parte da simulação (Figura 5.5), consideramos um conjunto com oito requisições a serem executadas na fila de entrega, na segunda parte (Figura 5.6) as quatro primeiras requisições já foram distribuídas, dessas quatro, três são conflitantes pois queremos um número elevado de requisições conflitantes nesse momento.

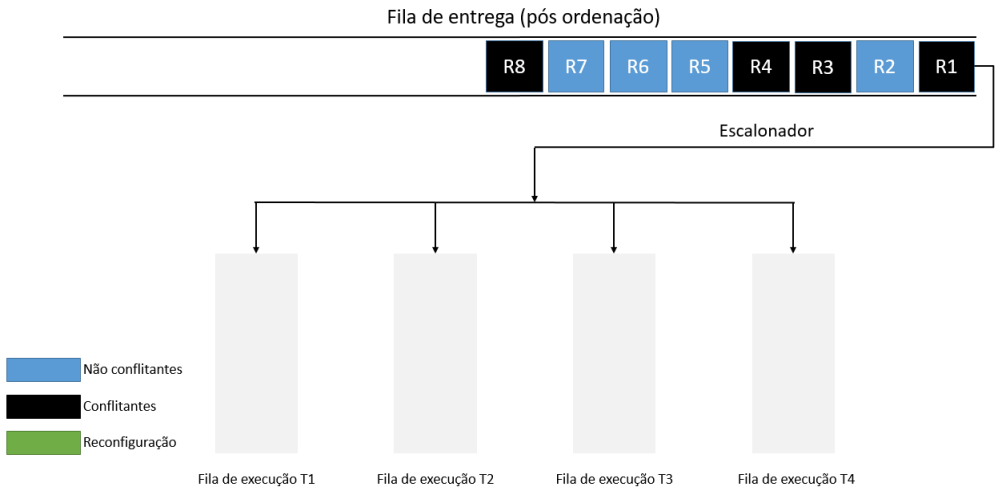


Figura 5.5: Execuções RME paralela reconfiguravel parte 1

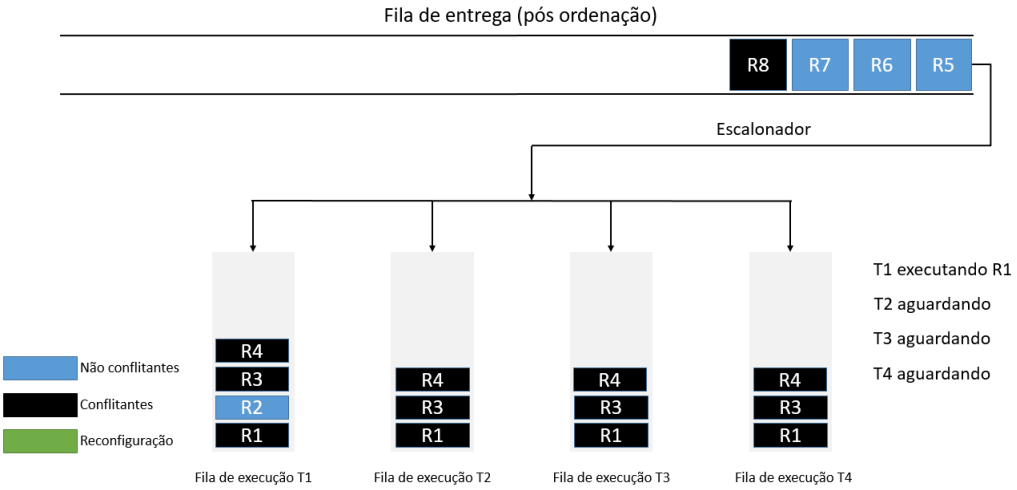


Figura 5.6: Execuções RME paralela reconfiguravel parte 2

No terceiro momento (Figura 5.7) a política utilizada para reconfiguração decide desativar uma *thread* de execução, a *thread* com maior ID dentre as *threads* ativas é escolhida para ser desativada, para isso o sistema distribui uma requisição do tipo reconfiguração por todas as filas de execução (semelhante a distribuição das requisições conflitantes), a partir disso a T4 é marcada como inativa e não irá receber mais requisições pelo escalonador.

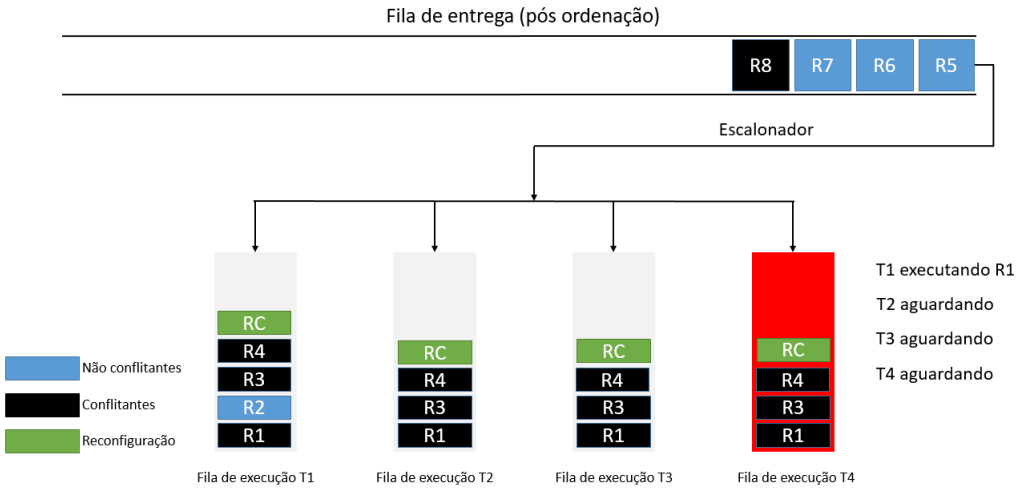


Figura 5.7: Execuções RME paralela reconfiguravel parte 3

Na parte 4 da simulação (Figura 5.8) consideramos que as requisições distribuídas (R1, R2, R3, R4 e RC) já foram executadas e as requisições restantes da fila de entrega já foram distribuídas pelas *threads* ativas, essas *threads* terminam de executar as requisições normalmente encerrando a simulação.

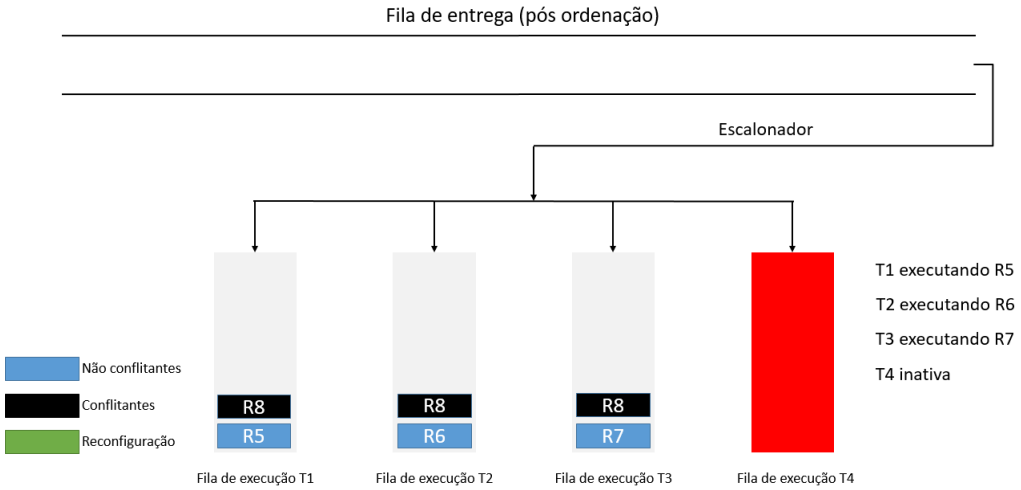


Figura 5.8: Execuções RME paralela reconfiguravel parte 4

Através dessa simulação podemos observar que a ativação/desativação das *threads* é feita com base nas requisições recebidas atualmente pelo sistema, podendo ele ativar *threads* quando as requisições recebidas são em sua maioria não conflitantes ou desativar caso contrário. Através da alteração no grau de paralelismo podemos influenciar no desempenho do sistema pois apenas as *threads* ativas participaram dos protocolos da RME paralela e, com isso, afetam no desempenho.

5.3 Políticas de Reconfiguração

A definição da nova configuração do sistema e de quando adotá-la segue uma política de reconfiguração (linha 4 –Algoritmo 3), que pode ser especificada pelo usuário. A política considera a requisição que está sendo escalonada, o número atual de *threads* ativas, além das configurações para o número mínimo e máximo de *threads* ativas.

A ideia básica das políticas de reconfiguração apresentadas nessa seção é monitorar uma quantidade N de requisições recebidas para execução, quando o número de requisições recebidas pela política for igual ao valor N verifica-se a porcentagem de requisições conflitantes recebidas nesse período, essa porcentagem é útilizada para decidir quantas *threads* deverão ser alteradas.

O Algoritmo 5 apresenta um exemplo de política, a qual estabelece que a cada 10.000 requisições é determinado o percentual de requisições conflitantes. Caso o *workload* apresente uma quantidade de até 20% de requisições conflitantes, uma *thread* é ativada até que a quantidade máxima seja atingida. Caso contrário, uma *thread* é desativada até que a quantidade mínima seja atingida. Com um *workload* acima de 20% conflitantes,

Algoritmo 5 Política de reconfiguração.

```
variables: Variables and sets used.
    conflict ← 0                                   // counter for the number of conflict requests
    notConflict ← 0                                // counter for the number of non conflict requests
    period ← 10000                                 // period (number of requests) to check for reconfigurations
reconfigPolicy(request,minThreads,currentThreads,maxThreads)
 1) if request.groupId == CONFLICT_ALL then
 2)     conflict++
 3) else
 4)     notConflict++
 5) end if
 6) if (conflict + notConflict) == period then
 7)     percent ← conflict * 100/period
 8)     conflict ← 0; notConflict ← 0
 9)     if (percent ≤ 20) ∧ ((currentThreads + 1) ≤ maxThreads) then
10)         return 1
11)     else if (percent > 20) ∧ ((currentThreads − 1) ≥ minThreads) then
12)         return -1
13)     end if
14) end if
15) return 0
```

gasta-se muito tempo sincronizando as *threads* (linhas 5-14 –Algoritmo 2) e geralmente o desempenho é melhor em uma execução sequencial [22, 2].

Nesse exemplo, escolhemos o valor de 10.000 para ser o intervalo entre as reconfigurações, não houve nenhum estudo para a escolha desse valor, apenas escolhemos um valor que facilitasse a observação dos experimentos. Vale ressaltar que não existe um número ideal para o intervalo de reconfiguração, isso depende da aplicação, se ela necessitar de um ajuste rápido talvez um número menor seja mais indicado, caso contrário um valor alto pode ser utilizado. A sugestão para a escolha do valor é que seja feito um estudo acerca do comportamento do sistema, com relação ao recebimento de requisições, e a partir disso definir as políticas e o intervalo para reconfiguração que melhor atenda a aplicação estudada.

O Algoritmo 6 apresenta um outro exemplo de política, semelhante ao Algoritmo 5, a diferença está na quantidade de *threads* alteradas a cada intervalo de reconfiguração, nesse caso a política é mais agressiva, ou ela utiliza o número máximo de *threads* (caso o percentual de requisições conflitantes seja menor que 20%) ou utiliza o número mínimo.

E por último o Algoritmo 7 que representa uma variação das outra políticas, nesse caso foram definidos alguns intervalos, caso a porcentagem de requisições conflitantes for menor que 25% o número de *threads* ativas passa a ser o máximo, caso seja entre 24% e 50% o sistema utilizará 60% do número máximo de *threads*, caso seja entre 49% e 75% o sistema utilizará 30% do número máximo de *threads*, caso seja maior ou igual 75% o sistema utilizará o mínimo de *threads*.

Algoritmo 6 Política de reconfiguração.

```
variables: Variables and sets used.
    conflict ← 0                                   // counter for the number of conflict requests
    notConflict ← 0                                // counter for the number of non conflict requests
    period ← 10000                                 // period (number of requests) to check for reconfigurations
reconfigPolicy(request,minThreads,currentThreads,maxThreads)
 1) if request.groupId == CONFLICT_ALL then
 2)     conflict++
 3) else
 4)     notConflict++
 5) end if
 6) if (conflict + notConflict) == period then
 7)     percent ← conflict * 100/period
 8)     conflict ← 0; notConflict ← 0
 9)     if (percent < 20) then
10)         return (maxThreads − currentThreads)                    // max number of threads
11)     else
12)         return ((currentThreads − 1) * (−1))                    // min number of threads
13)     end if
14) end if
15) return 0
```

Algoritmo 7 Política de reconfiguração.

```
variables: Variables and sets used.
    conflict ← 0                                   // counter for the number of conflict requests
    notConflict ← 0                                // counter for the number of non conflict requests
    period ← 10000                                 // period (number of requests) to check for reconfigurations
reconfigPolicy(request,minThreads,currentThreads,maxThreads)
 1) if request.groupId == CONFLICT_ALL then
 2)     conflict++
 3) else
 4)     notConflict++
 5) end if
 6) if (conflict + notConflict) == period then
 7)     percent ← conflict * 100/period
 8)     conflict ← 0; notConflict ← 0
 9)     if (percent < 25) then
10)         return (maxThreads − currentThreads)                    // max number of threads
11)     else if (percent < 50) then
12)         return ((maxThreads*60)/100 − currentThreads)           // 60 percent of maxThreads
13)     else if (percent < 75) then
14)         return ((maxThreads*30)/100 − currentThreads)           // 30 percent of maxThreads
15)     else
16)         return ((currentThreads − 1) * (−1))                    // min number of threads
17)     end if
18) end if
19) return 0
```

5.4 Conclusão

Neste capítulo concluímos que através do uso do protocolo para execuções paralelas conseguiu-se um grande aumento de desempenho em *workloads* favoráveis (com requisições

predominantemente independentes), mas impacta negativamente caso a predominância seja de requisições dependentes devido a necessidade de sincronização adicionais. Com algumas adaptações nesse protocolo e o uso de políticas de reconfiguração foi possível implementar uma RME paralela e reconfigurável. Através dessa nova proposta o sistema será capaz de adaptar o número de *threads* de execução de cada réplica de acordo com o *workload* atual, garantindo que o sistema possa alcançar desempenho máximo na presença de *workloads* favoráveis, e que ele possa se ajustar para sofrer um impacto menor na presença de *workloads* não favoráveis.

Capítulo 6

Experimentos

Visando analisar o desempenho das soluções propostas bem como o comportamento de uma RME com execuções paralelas e reconfigurável, os protocolos propostos foram implementados no Bft-SMaRt (Seção 4.1), que representa a concretização de uma RME desenvolvida na linguagem de programação Java, e alguns experimentos foram realizados no Emulab [30]. O principal objetivo destes experimentos não é determinar os valores máximos de desempenho, mas sim analisar as diferenças entre as abordagens e determinar os ganhos advindos com a possibilidade de reconfiguração.

Aplicação utilizada: Lista Encadeada. Esta aplicação foi implementada nos servidores através de uma lista encadeada (*LinkedList*), que é uma estrutura de dados não sincronizada, i.e., caso duas ou mais *threads* acessem esta estrutura concorrentemente e pelo menos uma delas modifique a sua estrutura, então estes acessos devem ser sincronizados. Neste experimento, a lista foi utilizada para armazenar inteiros (*Integer*) e as seguintes operações foram implementadas para seu acesso: *boolean add (Integer i)* – adiciona i no final da lista e retorna *true* caso i ainda não esteja na lista, retorna *false* caso contrário; *boolean remove(Integer i)* – remove i e returna *true* caso i esteja na lista, retorna *false* caso contrário; *Integer get(int index)* – retorna o elemento da posição indicada; e *boolean contains(Integer i)* – retorna *true* caso i esteja na lista, retorna *false* caso contrário. Note que todas estas operações possuem um custo de $O(n)$, onde n é o tamanho da lista. As seguintes dependências foram definidas para estas operações: *add* e *remove* são dependentes de todas as outras operações (`CONFLICT_ALL`), enquanto que *get* e *contains* não possuem dependências (`CONFLICT_NONE`), somente as já definidas com *add* e *remove*.

Configuração dos Experimentos. O ambiente para os experimentos foi constituído por 6 máquinas *d430* (2.4 GHz E5-2630v3, com 8 núcleos e 2 *threads* por núcleo, 64GB

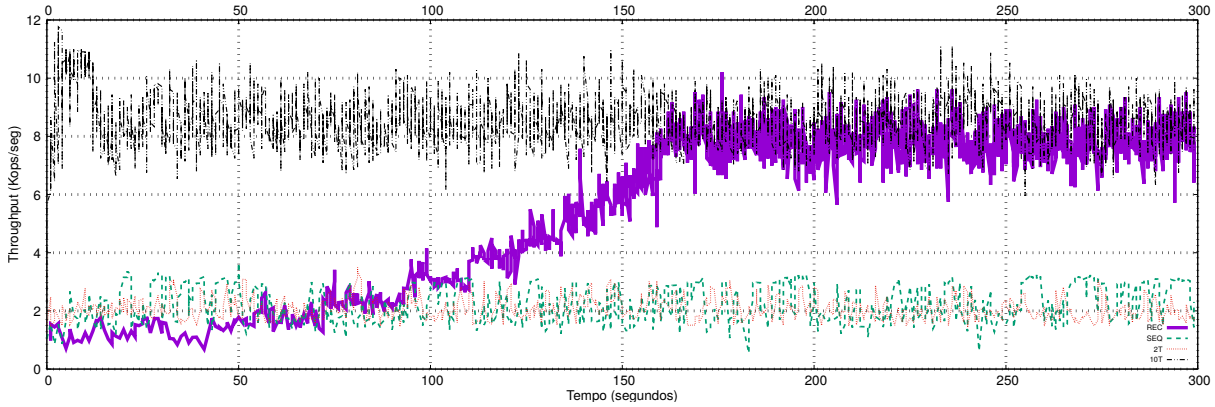
de RAM e interface de rede gigabit) conectadas a um *switch* de 1Gb. O Bft-SMaRt foi configurado com 3 servidores para tolerar até uma falha por parada (*crash*). Cada servidor executou em uma máquina separada, enquanto que 90 clientes foram distribuídos uniformemente nas outras 3 máquinas. O ambiente de *software* utilizado foi o sistema operacional Ubuntu 14 64-*bit* e máquina virtual Java de 64 *bits* versão 1.7.0_75.

Nos experimentos, a lista foi inicializada com $100k$ entradas em cada réplica e utilizamos apenas as operações *add* e *contains* para execução de operações `CONFLICT_ALL` e `CONFLICT_NONE`, respectivamente. O parâmetro destas operações sempre foi o último elemento da lista $(100k - 1)$ de forma que em todos os experimentos foi possível executar o mesmo conjunto de operações com os mesmos parâmetros. Para verificar o desempenho das diferentes abordagens, o *throughput* foi medido em um dos servidores (sempre o mesmo – líder do consenso [6]) a cada 1000 requisições durante um intervalo de 300 segundos. A política de reconfiguração utilizada foi a apresentada no Algoritmo 5, mas utilizamos diferentes períodos para calcular a percentagem de operações conflitantes, conforme descrito nos experimentos.

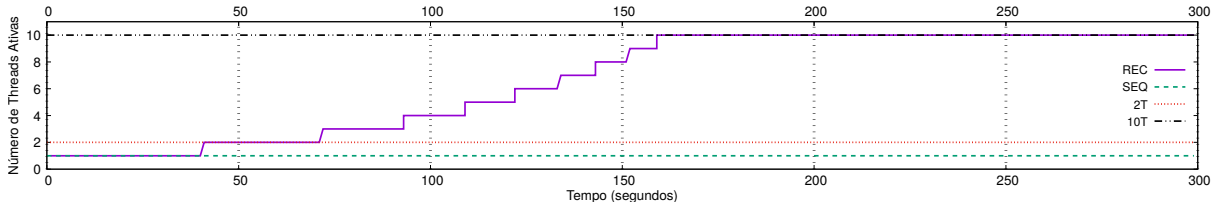
6.1 Resultados e Análises

Três experimentos foram realizados, os quais permitem mostrar que: *(1)* quando o *workload* favorece, o sistema se ajusta para uma configuração com o máximo de *threads* ativas visando aumentar o desempenho; *(2)* quando o *workload* não favorece, o sistema se ajusta para o mínimo de *threads* ativas causando o menor impacto possível no desempenho; e *(3)* quando o workload varia durante a execução, o sistema vai se ajustando para tentar sempre obter o melhor desempenho possível.

No primeiro experimento (Figura 6.1) os clientes geram um *workload* que favorece o paralelismo (100% `CONFLICT_NONE`) e o sistema foi configurado para iniciar com apenas 1 *thread* ativa, pondendo chegar até 10. O período para verificar a necessidade de reconfigurações foi especificado para cada 50.000 requisições. Para efeitos de comparação, os gráficos mostram os valores do *throughput* para o sistema com execuções paralelas e reconfigurações (REC), com execução sequencial que representa uma RME tradicional (SEQ), sem reconfigurações mas paralelo configurado com 2 (2T) e 10 (10T) *threads*. Apesar da melhor configuração para este *workload* ser a com 10 threads, isso só seria possível com um conhecimento prévio a respeito das operações que seriam executadas no sistema (neste caso apenas operações *contains*), o que não é realista.



(a) *Throughput*.

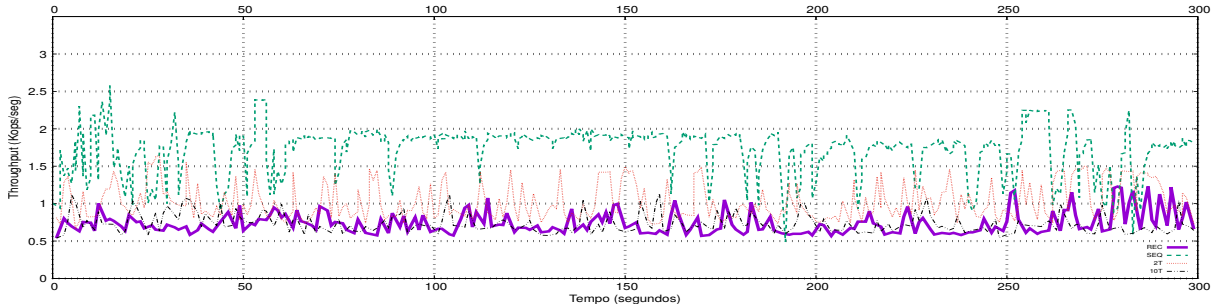


(b) Número de *threads* ativas.

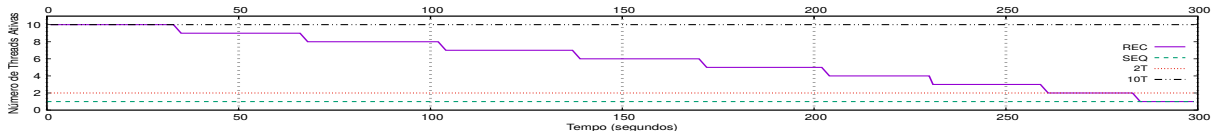
Figura 6.1: *Throughput* para *workload* de 0% conflitantes.

É possível perceber que com o passar da execução, o sistema com reconfiguração vai ativando as *threads* até atingir o limite máximo de 10, fazendo com que o desempenho seja praticamente o mesmo da configuração 10T. Outro ponto que merece destaque é que com apenas uma *thread* ativa, o desempenho é suavemente inferior ao da execução sequencial devido aos gastos na comunicação entre o escalonador e a *thread* através da fila sincronizada.

O segundo experimento (Figura 6.2) representa um cenário oposto ao primeiro e mostra que quando o *workload* não favorece (100% `CONFLICT_ALL`), mesmo inicializado com uma configuração inadequada (10 *threads* ativas), o sistema vai desativando *threads* até atingir o limite mínimo de 1 *thread* ativa causando o menor impacto possível no desempenho. Neste experimento, o período para verificar a necessidade de reconfigurações foi configurado para 25.000 requisições. Novamente são apresentados os valores para o sistema configurado como paralelo e reconfigurável (REC), sequencial (SEQ), paralelo com 2 (2T) ou 10 (10T) *threads*. Vale destacar que conforme as *threads* vão sendo desativadas através de reconfigurações, recursos podem ir sendo desalocados.



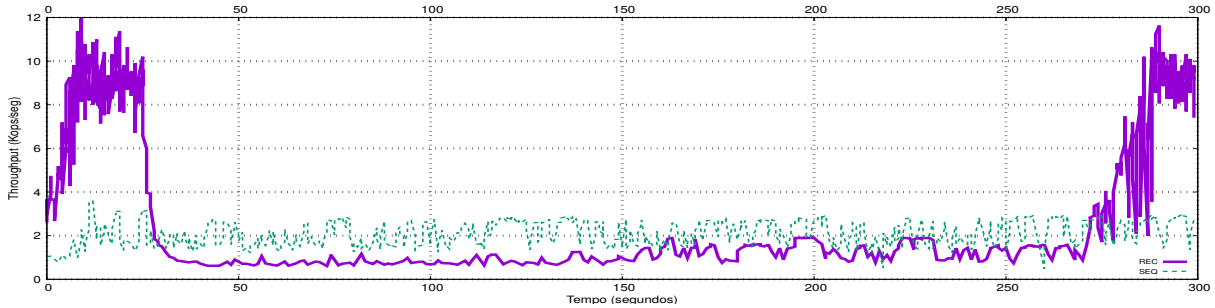
(a) *Throughput*.



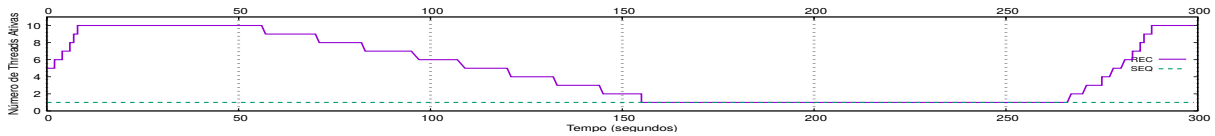
(b) Número de *threads* ativas.

Figura 6.2: *Throughput* para *workload* de 100% conflitantes.

Finalmente, o terceiro experimento (Figura 6.3(b)) representa um cenário em que o workload sofre variações durante a execução. Inicialmente cada cliente executa 5.000 requisições não conflitantes, depois passam a executar 5.000 requisições conflitantes e no final voltam a executar operações não conflitantes. Na execução com reconfiguração, o sistema foi inicializado com 5 threads ativas e o período de reconfigurações foi definido como a cada 10.000 requisições.



(a) *Throughput*.



(b) Número de *threads* ativas.

Figura 6.3: *Throughput* variando o *workload* durante a execução.

Através deste experimento podemos perceber que o sistema inicialmente ativa *threads* até atingir o limite máximo para obter o melhor desempenho visto que o *workload* favorece. Quando os clientes passam a executar operações conflitantes o sistema passa a desativar as *threads*, as quais são ativadas novamente quando os clientes voltam a executar operações não conflitantes.

Nestes experimentos utilizamos uma política de reconfiguração mais conservadora que ativa ou desativa apenas uma *thread* a cada reconfiguração, o que permitiu mostrar o comportamento dos protocolos propostos. Porém, políticas mais agressivas podem ser utilizadas com o objetivo de se atingir a configuração ideal mais rapidamente. Além disso, analisando sobre outra perspectiva, a taxa de conflitos é algo externo aos servidores e independe do estado interno de alocação e utilização de recursos. Neste sentido, uma política poderia criar níveis intermediários de paralelismo buscando mapear intervalos de taxas de conflitos para um número de *threads* ativas de forma que o balanço entre desempenho e ociosidade de recursos seja ótimo.

Capítulo 7

Conclusão

Este trabalho apresentou a proposta de uma RME paralela e reconfigurável. Essa nova proposta permite que o sistema ajuste o seu grau de paralelismo através do uso de políticas de reconfiguração. Como na RME paralela o grau de paralelismos é definido de maneira estática, a nossa proposta permite que as aplicações possam alterar o grau de paralelismo em tempo de execução, podendo alcançar o desempenho máximo na presença de *workloads* favoráveis, incrementando o número de *threads* de execução ao máximo, e pode também, caso os *workloads* não sejam favoráveis, diminuir esse número para que o impacto gerado pelas sincronizações adicionais na presença de requisições conflitantes seja menor.

7.1 Revisão dos Objetivos deste Trabalho

1. **Fazer uma revisão teórica sobre sistemas distribuídos e temas relacionados.**

   O capítulo 2 apresentou conceitos relacionados a sistemas distribuídos, suas características e os desafios que surgem na tentativa de implementar uma aplicação distribuída com essas características.

   O capítulo 3 foi responsável por apresentar um dos principais objetivos de um sistema distribuído, que é a capacidade de tolerar falhas, alcançar esse objetivo é o mesmo que alcançar a dependabilidade. Foram apresentados meios para se alcançar a dependabilidade e seus atributos. Ao final do capítulo foram abordados problemas comuns em ambientes distribuídos, como consenso e difusão atômica.

   O capítulo 4 apresentou conceitos relacionados a abordagem replicação máquina de estados, que é uma das abordagens mais utilizadas na construção de sistemas tolerantes a falhas. Foram apresentadas também algumas propostas para aumentar o desempenho de uma RME, a ideia dessas novas propostas é poder aproveitar

da arquitetura *multi-core* paralelizando seus processos. Ao final do capítulo foi apresentado o BFT-SMaRt, que é a concretização de uma replicação Máquina de Estados tolerante tanto a falhas por *crash* quanto bizantinas.

Com todo o conteúdo abordado nesses três capítulos, foi possível atingir o primeiro objetivo proposto no trabalho.

2. **Proposta de um protocolo para RME paralela que melhora os protocolos anteriormente propostos.**

   O capítulo 5 discute o protocolo propostos para implementação de uma RME paralela. A ideia básica do protocolo é dividir o estado da aplicação entre um conjunto de threads de execução, de forma que cada *thread* acesse a sua partição do estado para executar operações em paralelo. Esse protocolo permite a definição de grupos intermediários de dependências, além dos grupos de dependentes e independentes, aumentando o paralelismo na execução.

3. **Proposta de extensão do protocolo anterior para suportar reconfiguração no grau de paralelismo.**

   O capítulo 5 também apresenta como foi possível, através de adaptações no protocolo para RME paralela, implementar um protocolo responsável por permitir ajustes no grau de paralelismo da aplicação. Vimos que introduzindo os conceitos de *threads* ativas e inativas e do uso de mais um ponto de sincronização foi possível efetuar esses ajustes sem causar problemas no funcionamento do sistema.

4. **Definir e implementar algumas políticas de reconfiguração.**

   Ainda no capítulo 5, foi apresentado o conceito de políticas de reconfiguração, além disso, foram apresentados três exemplos de implementação dessas políticas, basicamente essas políticas são responsáveis por monitorar a porcentagem de requisições conflitantes, com base nessa porcentagem as políticas decidirão o número de *threads* que deverão seguir ativas no sistema.

5. **Apresentação e análise de uma série de experimentos realizados com uma implementação dos protocolos propostos.**

   No capítulo 6 apresentamos os experimentos realizados utilizando os algoritmos propostos, foi possível ver através dos gráficos o comportamento do sistema e o ganho pelo uso dinâmico de número de *threads*.

7.2 Trabalhos Futuros

Como trabalhos futuros pretedemos explorar outras formas de escalonar as requisições, como o escalonamento em lotes de forma que o custo da sincronização entre as *threads* seja diluído entre as requisições do lote e, principalmente, analisar políticas de reconfiguração baseadas em outros parâmetros, como o *throughput* e/ou a latência atual apresentada pelo sistema, além do cenário atual de alocação de recursos.

Referências

[1] Eduardo Adilio Pelinson Alchieri. *Protocolos Tolerantes a Faltas Bizantinas para Sistemas Distribuídos Dinâmicos*. Tese de doutorado em engenharia de automação e sistemas, Universidade Federal de Santa Catarina, Setembro 2011. 2

[2] Eduardo Adilio Pelinson Alchieri. Suportando execuções paralelas no BFT-SMaRt. In *XVI Workshop de Testes e Tolerância a Falhas, 2015, Vitória - ES. XVI Workshop de Testes e Tolerância a Falhas (XXXIII Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos)*, 2015. ix, 2, 16, 17, 18, 19, 20, 22, 26, 32

[3] Eduardo Adilio Pelinson Alchieri, Alysson Neves Bessani, e Joni da Silva Fraga. Replicação máquina de estados dinâmica. In *Anais do XIV Workshop de Teste e Tolerância a Falhas- WTF 2013*, 2013. 18

[4] James Aspnes. Randomized protocols for asynchronous consensus. *Distrib. Comput.*, 16(2-3):165–175, September 2003. 14

[5] Hagit Attiya e Jennifer Welch. *Distributed Computing: Fundamentals, Simulations and Advanced Topics*. John Wiley & Sons, 2004. 14

[6] Alysson Bessani, João Sousa, e Eduardo Alchieri. State machine replication for the masses with BFT-SMaRt. In *Proc. of the International Conference on Dependable Systems and Networks*, 2014. 16, 18, 21, 36

[7] George F Coulouris, Jean Dollimore, Tim Kindberg, Gordon Blair. *Distributed Systems: Concepts and Design*. Pearson, 5nd edition, 2011. 9, 13

[8] Miguel Castro e Barbara Liskov. Practical Byzantine fault-tolerance and proactive recovery. *ACM Transactions on Computer Systems*, 20(4):398–461, November 2002. 2, 18

[9] Davi da Silva Böge, Joni da Silva Fraga, e Eduardo Alchieri. Reconfigurable scalable state machine replication. In *Proceedings of the 2016 Latin-American Symposium on Dependable Computing*, 2016. 17

[10] da Silva Böge, D., da Silva Fraga, J., e Alchieri, E. Reconfigurable scalable state machine replication. In *IIn Proceedings of the 2016 Latin-American Symposium on Dependable Computing.*, 2016. 17

[11] Bezerra, C. E., Pedone, F., e Renesse, R. V. Scalable state-machine replication. In *In 44th IEEE/IFIP International Conference on Dependable Systems and Networks.*, 2014. 17, 21

[12] Michael J. Fischer, Nancy A. Lynch, e Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, April 1985. 14

[13] Vassos Hadzilacos e Sam Toueg. A modular approach to the specification and implementation of fault-tolerant broadcasts. Technical report, Department of Computer Science, Cornell, May 1994. 12, 13, 14, 15, 16

[14] Maurice Herlihy e Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programing Languages and Systems*, 12(3):463–492, July 1990. 2

[15] George F Coulouris, Jean Dollimore, Tim Kindberg. *Distributed Systems: Concepts and Design*. Addison Wesley, 4nd edition, 2005. 6, 7, 8, 9

[16] Ramakrishna Kotla e Mike Dahlin. High throughput byzantine fault tolerance. In *Proceedings of the International Conference on Dependable Systems and Networks*, 2004. 2, 17, 18

[17] Leslie Lamport, Robert Shostak, e Marshall Pease. The Byzantine generals problem. *ACM Transactions on Programing Languages and Systems*, 4(3):382–401, July 1982. 14

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

[19] Bezerra, C. E., Le, L. H. e Pedone, F. Dynamic scalable state machine replication. *2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 13–24, 2016. 2, 17

[20] Barbara Liskov e James Cowling. Viewstamped replication revisited. Technical Report MIT-CSAIL-TR-2012-021, MIT, July 2012. 18

[21] Parisa Jalili Marandi, Carlos Eduardo Bezerra, e Fernando Pedone. Rethinking state-machine replication for parallelism. In *Proceedings of the 34th Int. Conference on Distributed Computing Systems*, 2014. 2, 3, 17, 18, 21

[22] Parisa Jalili Marandi e Fernando Pedone. Optimistic parallel state-machine replication. In *Proceedings of the 33rd International Symposium on Reliable Distributed Systems*, 2014. 2, 3, 17, 26, 32

[23] Odorico Mendizabal, Parisa Jalili Marandi, Fernando Dotti, e Fernando Pedone. Recovery in parallel state-machine replication. In *Proc. of Int. Conference on Principles of Distributed Systems*, 2014. 17

[24] B. Oki e Barbara Liskov. Viewstamped replication: A new primary copy method to support highly-available distributed systems. In *Proceedings of the 7th Annual ACM Symposium on Principles of Distributed Computing*, pages 8–17, 1988. 18

[25] Alchieri, Eduardo Adilio Pelinson. Uma infra-estrutura com segurança de funcionamento para coordenação de serviços web cooperantes. 2007. 13, 14, 15

[26] Fred B. Schneider. Implementing fault-tolerant service using the state machine aproach: A tutorial. *ACM Computing Surveys*, 22(4):299–319, December 1990. 2, 16, 18

[27] Andrew S. Tanembaum. *Distributed Systems: Principles and Paradigms*. Pearson, 2nd edition, 2008. ix, 7

[28] Algirdas Avizienis Ucla, Algirdas Avizienis, Jean claude Laprie, e Brian Randell. Fundamental concepts of dependability, 2001. 12

[29] Taisy Silva Weber. Um roteiro para exploração dos conceitos básicos de tolerância a falhas. 2002. 11, 12

[30] Brian White, Jay Lepreau, Leigh Stoller, Robert Ricci, Shashi Guruprasad, Mac Newbold, Mike Hibler, Chad Barb, e Abhijeet Joglekar. An Integrated Experimental Environment for Distributed Systems and Networks. In *Proc. of 5th Symp. on Operating Systems Design and Implementations*, 2002. 35

[31] Maciej Zbierski. Parallel byzantine fault tolerance. In Antoni Wilinski, Imed El Fray, e Jerzy Pejas, editors, *Soft Computing in Computer and Information Science*, volume 342 of *Advances in Intelligent Systems and Computing*, pages 321–333. Springer International Publishing, 2015. 2