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

Replicação Máquina de Estados Paralela com Escalonamento Híbrido

Aldênio de V. Burgos

Dissertação apresentada como requisito parcial para
conclusão do Mestrado em Informática

Orientador
Prof. Dr. Eduardo Adilio Pelinson Alchieri

Brasília
2021

Ficha catalográfica elaborada automaticamente,
com os dados fornecidos pelo(a) autor(a)

de Vilaça Burgos, Aldenio
dD278r Replicação Máquina de Estados Paralela com Escalonamento Híbrido / Aldenio de Vilaça Burgos; orientador Eduardo Adilo Pelinson Alchieri. -- Brasília, 2021.
100 p.

Dissertação (Mestrado - Mestrado em Informática) -- Universidade de Brasília, 2021.

1. Replicação Máquina de Estados. 2. Replicação Máquina de Estados Paralela. 3. Blockchain. I. Adilo Pelinson Alchieri, Eduardo, orient. II. Título.

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

Replicação Máquina de Estados Paralela com Escalonamento Híbrido

Aldênio de V. Burgos

Dissertação apresentada como requisito parcial para
conclusão do Mestrado em Informática

Prof. Dr. Eduardo Adilio Pelinson Alchieri (Orientador)
Universidade de Brasília

Prof. Dr. Odorico Machado Mendizabal
Universidade Federal de Santa Catarina

Prof.a Dr.a Alba Cristina M. A. de Melo
Universidade de Brasília

Prof.a Dr.a Genaina Nunes Rodrigues
Coordenadora do Programa de Pós-graduação em Informática

Brasília, 17 de agosto de 2021

Dedicatória

A minha esposa Carolina e aos meus filhos Davi, Théo e Melissa.
Sem vocês eu nada seria.

Agradecimentos

Primeiramente gostaria de agradecer a Deus, por tornar tudo isso possível. Agradeço também a minha família, pelo incentivo e apoio que me deram durante a realização deste trabalho. Avante, Autobots! Agradeço ao meu professor orientador Eduardo Adílio Pelinson Alchieri e ao doutor André Henrique de Siqueira, por todas as conversas e pela atenção a mim despendida durante o mestrado. Agradeço também aos professores dedicados do Departamento de Informática da UnB. Foi grande o aprendizado adquirido com eles durante este período de convivência. Também quero agradecer aos colegas que fizeram e fazem o Banco Central do Brasil. Por fim agradeço a todos que me ajudaram direta ou indiretamente na minha jornada.

Resumo

A crescente demanda por aplicações confiáveis e com tempos de resposta cada vez mais baixos vem estimulando as pesquisas sobre o desempenho de serviços tolerantes a falhas há vários anos. Sabemos que projetar aplicações que toleram falhas é fundamental, mas não é suficiente. Uma aplicação confiável cujo tempo de resposta ultrapasse a tolerância (cada vez mais curta) de seus clientes pode ser tão destrutiva para os negócios quanto uma aplicação não confiável.

A técnica de Replicação Máquina de Estados (RME), lançada em 1990 e utilizada na implementação de serviços confiáveis, trás consigo um efeito colateral. Em sua forma clássica, ela elimina qualquer tipo de concorrência ou paralelismo no atendimento das requisições dos clientes.

Há anos que os principais fabricantes de microprocessadores têm procurado aumentar o desempenho de seus equipamentos, pelo aumento de sua capacidade de paralelismo, adicionando cada vez mais núcleos de processamento em suas arquiteturas *multi-core*. Por conseguinte, o comportamento sequencial da RME clássica tornou-se uma séria desvantagem. Isso abriu o caminho para um vasto esforço de pesquisa no sentido de aumentar o desempenho desses serviços replicados, e assim surgiu a Replicação Máquina de Estados Paralela (RMEP). A ideia principal desta abordagem é separar os comandos em conflitantes e não conflitantes, de forma a permitir que comandos não conflitantes sejam executados em paralelo. Dois comandos conflitam quando acessam uma mesma variável e pelo menos um deles altera o seu estado.

Nas últimas duas décadas surgiram novas técnicas, algoritmos e estudos no sentido de aperfeiçoar a RMEP. Um aspecto fundamental destas soluções é como escalonar as requisições dos clientes de modo a permitir que uma parte delas seja executada em paralelo nas réplicas do serviço. Neste contexto, esta dissertação de mestrado propõe uma abordagem de RMEP que utiliza um método de escalonamento híbrido, com o objetivo de aumentar o desempenho do sistema pelo aumento do seu paralelismo. O método desenvolvido é chamado híbrido pois deriva do aperfeiçoamento de uma combinação de dois proeminentes métodos de escalonamento para RMEP anteriores, o antecipado e o tardio.

O primeiro tem um desempenho excelente com cargas de trabalho com predominância de comandos não conflitantes, porém esse desempenho cai abruptamente na medida que aumentamos a taxa de conflitos. Já o segundo, que supera o primeiro na medida que a taxa de conflitos cresce, tem em seu funcionamento um fator limitante do desempenho quando a taxa de conflitos é reduzida visto que utiliza um paralelizador único para gerenciar as dependências dos comandos. A abordagem híbrida proposta neste trabalho busca agrupar as vantagens de cada uma destas abordagens. Experimentos realizados mostram que a abordagem híbrida apresentou um desempenho superior aos de seus antecessores em diversas cargas de trabalho, além de eliminar as limitações anteriormente elencadas. Também foi realizado um estudo de caso sobre a aplicação de uma RMEP, com o método de escalonamento híbrido, em sistemas de *Blockchain*.

Palavras-chave: Replicação Máquina de Estados Paralela, Replicação Máquina de Estados, Blockchain

Abstract

The growing demand for reliable applications with ever lower response times has been driving research on the performance of fault-tolerant services for several years. We know that designing fault-tolerant applications is critical, but it is not enough. A trusted application whose response time exceeds the (ever-shorter) tolerance of its customers can be just as destructive to the business as an untrusted application.

The State Machine Replication (SMR) technique, launched in 1990 and widely used in the implementation of reliable services, brings with it a side effect. In its classic form, it eliminates any kind of concurrency or parallelism in fulfilling customer requests. For years, the main microprocessor manufacturers have been looking to increase the performance of their equipment, by increasing their parallelism capacity, adding more and more cores to their multi-core architectures. That is, in an increasingly parallel world, the sequential behavior of classical SMR has become a serious disadvantage. This paved the way for a vast research effort to improve the performance of these replicated services, and thus came Parallel State Machine Replication (PSMR).

In the last two decades, new techniques, algorithms and studies have emerged in order to improve PSMR. A key aspect of these solutions is how to schedule client requests in order to allow a part of them to be executed in parallel in the service's replicas. In this context, this master's dissertation proposes an PSMR approach that uses a hybrid scheduling method, with the objective of increasing the system performance by increasing its parallelism. The developed method is called hybrid because it derives from the improvement of a combination of two prominent scheduling methods for earlier PSMR, Early and Late.

The former performs excellently with non-conflicting workloads, but this performance drops off sharply as we increase the conflict rate. [1] The second, which surpasses the first as the conflict rate grows,[2] has a performance limiting factor in its operation when the conflict rate is reduced. We demonstrated in the tests carried out that the proposed approach performed better than its predecessors, in addition to eliminating its main limitations.

[1] The percentage of conflicts is inversely proportional to the opportunities for parallelism between operations.

[2] Up to a certain limit, as very high conflict rates indicate typically sequential workloads.

A case study was also carried out on the application of an PSMR, with the proposed scheduling method, in Blockchain systems.

Keywords: Parallel State Machine Replication, State Machine Replication, Blockchain

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Abreviaturas e Siglas

COS Conflict Ordered Set.

DAG Directed Acyclic Graph.

DMT Deterministic MultiThreading.

P2P Peer-to-Peer.

PSMR Parallel State Machine Replication.

RME Replicação Máquina de Estados.

RMEP Replicação Máquina de Estados Paralela.

SMR State Machine Replication.

Capítulo 1

Introdução

A crescente demanda por aplicações confiáveis, acessíveis a qualquer hora e lugar, e com tempos de resposta cada vez mais baixos, deu novo fôlego às pesquisas sobre serviços tolerantes a falhas. Sabemos que projetar aplicações que toleram falhas é fundamental. A Replicação Máquina de Estado (RME) clássica de 1990 [1], muitas vezes referenciada pela sigla em inglês *State Machine Replication (SMR)*, é uma abordagem simples e abrangente para desenvolver aplicações tolerantes a falhas e com consistência forte (linearizabilidade) [2, 3]. De modo geral, a ideia do RME consiste em replicar o servidor da aplicação e, por meio de um protocolo de ordenação, executar todos comandos dos clientes em todas as réplicas na mesma ordem e de forma determinística. Ao começar no mesmo estado inicial e executar a mesma sequência de comandos, as réplicas passam pela mesma sequência de estados e produzem as mesmas respostas para cada comando. Essa abordagem causa, no cliente, a ilusão de estar se comunicando com um serviço único enquanto esconde a complexidade da comunicação com várias réplicas e um conjunto de possíveis falhas em componentes do sistema.

O RME clássico foi usado com sucesso em muitos serviços *online* (e.g., [4, 5]), apesar da sua configuração clássica oferecer um desempenho limitado pela capacidade de processamento sequencial da máquina. Essa limitação no desempenho ocorre porque, para garantir a consistência forte (linearizabilidade) do sistema, cada réplica precisa executar todos os comandos de forma sequencial, independentemente de quantos processadores ou núcleos de processamento disponha. Para alcançar o alto desempenho, cada vez mais demandado, as réplicas devem executar o maior número de comandos possível de forma paralela, porém sem perder a consistência forte (linearizabilidade) com as outras réplicas. Esse é o grande desafio dos algoritmos de *Replicação Máquina de Estados Paralela (RMEP)*.

As técnicas de RMEP se baseiam na observação de que comandos independentes podem ser executados simultaneamente, enquanto os comandos dependentes devem ser seri-

alizados e executados na mesma ordem pelas réplicas [1]. Dois comandos são considerados dependentes (ou conflitantes) quando acessam ao menos uma variável de estado comum e pelo menos um deles modifica essa variável, e são considerados independentes em todas as outras situações. Executar comandos dependentes simultaneamente pode resultar em estados inconsistentes nas réplicas, quebrando a principal premissa da RME.

Seguindo essa linha, diversas abordagens foram propostas ao longo dos anos. Foram propostos métodos que adotam estratégias otimistas e arriscam uma deterioração do desempenho no caso da hipótese otimista falhar, como o EVE [6]. Foram também propostos outros que utilizam um sistema "líder-seguidores" (*primary-copy*) [7], com alto tempo de recuperação em caso de falhas do líder, como o REX [8]. Ainda há outros, que possuem implementações tão complexas que dependem de modificações nos sistemas operacionais das máquinas que executam as réplicas do processo servidor, como o CRANE [9].

Embora o desempenho de uma RMEP dependa tanto da técnica utilizada quanto da mistura na carga de trabalho entre comandos independentes e dependentes, trabalhos anteriores mostraram que muitas cargas de trabalho são dominadas por comandos independentes (por exemplo, [10, 11, 12]) e que as abordagens paralelas resultam em melhorias substanciais no desempenho (por exemplo, [10, 11, 6, 12, 13]).

Algumas abordagens mais recentes realizam o escalonamento das requisições com base em conhecimentos específicos da aplicação (por exemplo, saber antecipadamente se duas operações conflitam) para processar comandos paralelos deterministicamente. De fato, um aspecto fundamental no contexto de RMEP é como escalonar as requisições para serem executadas por um conjunto de *threads* executoras (ou executores) seguindo as restrições já descritas. Essas técnicas não dependem do acaso como as técnicas baseadas em estratégias otimistas e conseguem mascarar as falhas em suas réplicas melhor do que a abordagem "líder-seguidores" quando ocorre uma falha no líder. Além disso, estas abordagens são independentes do sistema operacional e do hardware no qual executam. Dentre essas técnicas, duas se destacam por suas características mais próximas do modelo de RME tradicional: no *escalonamento tardio* [10, 14] as requisições são escalonadas para execução após serem ordenadas pelas réplicas, enquanto que no *escalonamento antecipado* [15] parte das decisões de escalonamento são realizadas antes da ordenação e precisam ser respeitadas durante a execução.

O escalonamento tardio foi lançado em Kotla [10], nele, dentre as diversas *threads* em cada réplica, uma faz o papel de paralelizador enquanto as outras executam as operações enviadas pelos clientes. O paralelizador examina as requisições e as insere num grafo de dependências, adicionando arestas para representar as relações de conflito. As *threads* executoras buscam requisições livres nesse grafo para executar e depois remover. Uma requisição está livre quanto não possuir arestas de entrada, indicando que todas

as requisições conflitantes anteriormente inseridas no grafo já foram executadas. Esta proposta inicial foi aperfeiçoada em *Escobar et al.* [14], com a proposição de uma abstração chamada de COS (*Conflict-Ordered Set*) que é usada para resolver as dependências, juntamente com a adoção de uma implementação livre de bloqueios (***LockFree***), que apresentou melhor desempenho.

Já no escalonamento antecipado [15], o estado da aplicação é particionado e os processos clientes atribuem suas requisições a uma *thread* ou *pool de threads* antes de enviá-las para o servidor. Nas réplicas, uma *thread* classificadora recebe as requisições em ordem total e as insere nas filas das *threads* executoras escolhidas pelo cliente. As *threads* executoras consomem e executam as requisições de suas filas na ordem (*FiFo*).

Apesar do escalonamento tardio distribuir melhor o trabalho entre os executores e apresentar um desempenho superior à execução sequencial até para índices de conflitos considerados altos (exemplo, até 25% de conflitos [14]), um paralelizador único fica responsável por inserir e remover requisições do grafo, limitando o desempenho desta solução. Por outro lado, o escalonamento antecipado tira proveito de decisões previamente realizadas, como por exemplo a divisão do estado da aplicação em partições e a definição de qual executor executará determinadas requisições, e exige pouco processamento no escalonamento, executado pelo classificador. Porém, são necessárias sincronizações adicionais nos executores para execução de requisições conflitantes, o que diminui drasticamente seu desempenho na presença de conflitos [15]. Além disso, esta solução não faz um bom gerenciamento de carga e insere bolhas de inatividade para coordenar as *threads* de execução [16].

Neste trabalho propomos uma abordagem *híbrida* para escalonamento, que se aproveita das vantagens de cada uma das soluções anteriores: a simplicidade e rapidez do classificador do modelo antecipado, bem como a distribuição de carga e a sincronização dos executores através de um grafo do modelo tardio. A ideia principal é particionar o estado da aplicação e atribuir um subgrafo para cada partição, com seu paralelizador correspondente. Cada requisição é recebida no classificador e encaminhada para os paralelizadores de acordo com a(s) partição(ões) que acessa. Quando uma requisição endereçada a mais de uma partição é recebida nos paralelizadores, cada um a insere em seu subgrafo, junto com suas dependências, de forma que esta requisição passa a conectar estes subgrafos. Por fim, um conjunto de executores atrelados a cada subgrafo executa as requisições que estiverem prontas, i.e., cujas dependências já foram resolvidas. Vale destacar que o escalonador híbrido é baseado nas ideias por trás dos escalonadores antecipado e tardio, e não uma simples união de ambos os escalonadores. Experimentos mostram que o escalonador híbrido possui as vantagens do escalonamento tardio e ao mesmo tempo não possui o desempenho limitado por aquilo que um único paralelizador consegue processar,

superando as demais abordagens em diversos cenários.

1.1 Objetivos

Esta dissertação de mestrado tem como objetivo principal propor um novo protocolo de escalonamento para Replicação Máquina de Estados Paralela que visa solucionar problemas de desempenho encontrados em soluções anteriores de RMEP encontradas na literatura. Para atingir este objetivo geral, os seguintes objetivos específicos foram definidos:

- Proposta de um protocolo para escalonamento híbrido em RMEP, visando remover as limitações impostas pelas soluções anteriores de escalonamento antecipado ou tardio.
- Implementação deste protocolo e integração com uma biblioteca de RME.
- Implementação de uma aplicação (para *microbenchmark*) e análise de desempenho da solução proposta, comparando-a com o estado da arte dos protocolos antecipado e tardio.
- Realizar um estudo de caso a fim de analisar como a solução para RMEP proposta poderia melhorar uma limitação de desempenho de *blockchains* privadas.

1.2 Organização do texto

O restante do trabalho está organizado da seguinte forma: O Capítulo 2 apresenta a fundamentação teórica necessária para a compreensão do trabalho desenvolvido. O Capítulo 3 mostra o estado da arte nas áreas de conhecimento relacionadas e detalha as duas abordagens que servirão como base para a nossa proposta. O Capítulo 4 descreve detalhadamente a abordagem híbrida. O Capítulo 5 expõe os resultados encontrados nos testes experimentais e explana o estudo de caso assim como os resultados encontrados em seus testes experimentais. O Capítulo 6 conclui a dissertação e aponta a direção para trabalhos futuros.

Capítulo 2

Fundamentação Teórica

A fim de permitir uma melhor compreensão, este capítulo apresenta a fundamentação teórica sobre os principais conceitos relacionados com o trabalho proposto.

2.1 Sistemas computacionais

Um sistema é definido como uma entidade que interage com outras entidades, ou seja, outros sistemas [17]. Sistemas são formados por um conjunto de componentes que interagem entre si. Cada componente de um sistema é também um sistema em si mesmo. Todos os sistemas externos a um dado sistema constituem seu ambiente. Componentes de hardware, softwares, seres vivos e fenômenos naturais são exemplos de sistemas.

Um sistema computacional consiste num conjunto de dispositivos eletrônicos (hardware) capazes de processar informações de acordo com um ou mais programas (software). Um programa de computador é um conjunto de instruções que descrevem uma tarefa a ser realizada pelo computador. Os computadores modernos costumam ser sistemas complexos, compostos por múltiplos componentes. Para reduzir a complexidade na produção e manutenção de programas de computador, é comum que programas utilizem serviços prestados por outros programas, ao invés de implementá-los novamente. Os sistemas operacionais (SO) são programas que surgiram para gerenciar os componentes dos computadores modernos e fornecer um conjunto de funcionalidades capaz de abstrair os detalhes do hardware subjacente [18].

Os sistemas operacionais modernos alocam de forma ordenada e controlada os recursos computacionais, viabilizando a execução de um ou mais processos. Um **processo** é a abstração de um programa em execução. Atualmente, um processo consiste em um ambiente de execução, junto a uma ou mais *threads* [19]. O ambiente de execução é a unidade de gerenciamento de recursos locais do sistema operacional.

Uma ***thread*** é a abstração do sistema operacional para uma atividade em execução na sua unidade central de processamento (CPU). As *threads* de um processo compartilham o acesso aos recursos de seu ambiente de execução e são as entidades escalonadas pelo SO para execução na CPU. O objetivo principal em se ter múltiplas *threads* de execução é maximizar o grau de execução concorrente entre operações, permitindo, assim, a sobreposição da computação com operações de entrada e saída, e possibilitando a execução simultânea de atividades em máquinas com múltiplos núcleos de processamento (*multi-core*) [19].

2.2 Falhas, erros e defeitos

Nos sistemas computacionais as palavras falha, erro e defeito possuem significados distintos. Defeito é o evento que ocorre quando a resposta dada pelo sistema difere da resposta correta esperada. A diferença entre o funcionamento correto e o funcionamento entregue pelo sistema é o erro. A falha é a causa do erro, e o erro, quando se propaga até a interface do sistema, ou serviço, causa o defeito [17].

As falhas podem ser internas ou externas ao sistema. Uma falha pode estar ativa, quando é a causa direta de um erro, ou dormente quando não encontrou as condições suficientes e necessárias para produzir alterações indesejadas no serviço entregue. O defeito em um componente do sistema S é uma falha para esse sistema S. Se essa falha for ativada, provocará um defeito no sistema S. Esse defeito se tornará uma falha para o sistema S' do qual o sistema S é componente, e assim por diante.

Dependabilidade é a capacidade de um sistema de evitar a ocorrência de defeitos mais severos ou frequentes do que um padrão pré-determinado. Essa capacidade é definida pela integração de seus atributos, ou seja, das propriedades abaixo [17]:

- Disponibilidade: indica a proporção de tempo que o sistema está pronto para aceitar novas demandas em um determinado período.
- Confiabilidade: indica a proporção de tempo em que o sistema manteve prestando seus serviços corretamente.
- Inofensividade: indica a ausência de consequências nocivas para o usuário ou ambiente, devido à sua operação.
- Integridade: indica a ausência de alterações inadequadas no sistema.
- Manutenibilidade: indica a capacidade do sistema de sofrer reparos, modificações e atualizações.

- Confidencialidade: indica a ausência de acesso não autorizado a suas operações ou informações.

Devido à relevância dos servidores no modelo arquitetura prevalente (Cliente-Servidor), várias técnicas foram desenvolvidas para aumentar os seus atributos de dependabilidade, vejamos:

- Técnicas de previsão de falhas: para estimar o número atual e futuro de falhas, tal como suas consequências prováveis.
- Técnicas de prevenção de falhas: para prevenir a introdução e ocorrência de falhas.
- Técnicas de tolerância a falhas: para evitar defeitos no sistema mesmo na presença de falhas.
- Técnicas de remoção de falhas: para reduzir o número de falhas e seus graus de severidade.

2.3 Comunicação entre processos

O tipo de rede determina como os processos se comunicam. Em uma rede ponto-a-ponto, a comunicação ocorre por meio de *links* que conectam pares de processos. Nesse tipo de rede, um processo pode enviar uma mensagem para um único processo por meio de um *link*. Em uma rede de *broadcast*, a comunicação de rede ocorre em um único canal compartilhado que conecta todos os processos. Em tal rede, um processo pode transmitir uma mensagem a todos os outros processos de uma só vez.

Em qualquer um dos tipos de rede, quando um processo se comunica unicamente com outro processo, chamamos tal transmissão de *unicast*. Quando um processo se comunica com um grupo específico de processos em uma única operação lógica, dá-se o nome de *multicast*. O *broadcast* acontece quando todos os processos da rede integram o grupo receptor da comunicação.

Existem inúmeras variantes dos algoritmos de comunicação de processos que disponibilizam primitivas com características e propriedades distintas [20]. Para manter o foco nos conceitos que fundamentam o trabalho realizado, vamos detalhar um pouco mais um desses grupos, o *multicast* tolerante a falhas. O *multicast* confiável é a primitiva de comunicação em grupo[1] mais simples dentre os *multicasts* tolerantes a falhas. Em termos gerais, os algoritmos de *multicast* confiáveis precisam garantir três propriedades [20]: (1) todos

[1]Técnica de comunicação entre processos por meio da qual uma mensagem é enviada para um grupo e, então, entregue a todos os membros do grupo, sem que o remetente saiba a identidade dos destinatários [19].

os processos corretos concordam com o conjunto de mensagens que entregam; (2) todas as mensagens transmitidas por processos corretos são entregues; e (3) nenhuma mensagem espúria é entregue. Algumas variantes do *multicast* confiável impõem requisitos adicionais na ordem em que as mensagens são entregues, conforme segue [2]:

- *Multicast FiFo* garante que as mensagens transmitidas pelo mesmo remetente são entregues na ordem em que foram transmitidas.
- *Multicast* causal garante que as mensagens sejam entregues de acordo com a relação de precedência causal [2], ou seja, se a transmissão de m precede causalmente a transmissão de m', então m deve ser entregue antes de m'.
- *Multicast* atômico garante que os processos entreguem todas as mensagens na mesma ordem.
- *Multicast FiFo* atômico garante que os processos entreguem todas as mensagens na mesma ordem e que as mensagens transmitidas pelo mesmo remetente são entregues na ordem em que foram transmitidas.
- *Multicast* causal atômico garante que os processos entreguem todas as mensagens na mesma ordem e de acordo com a relação de precedência causal.

Perceba que cada algoritmo garante exatamente as suas propriedades e nada além disso, por exemplo: (a) no *multicast* causal se duas mensagens não estiverem causalmente relacionadas processos diferentes podem entregá-las em ordens diferentes; (b) o *multicast FiFo* não determina relação de ordem entre mensagens de clientes diferentes; e (c) a ordem de entrega do *multicast* atômico independe da ordem de transmissão, contanto que as mensagens sejam entregues na mesma ordem aos processos do grupo receptor.

2.4 Sistemas distribuídos

Os sistemas distribuídos são definidos como "aqueles nos quais os componentes de hardware ou software, estando localizados em computadores interligados em rede, comunicam-se e coordenam suas ações apenas enviando mensagens entre si" [19]. Devido à concorrência inevitável entre seus componentes, combinada com a dificuldade de se ter um controle global esses são sistemas notoriamente difíceis de construir. No entanto, esta dificuldade é bastante reduzida quando se conta com primitivas de comunicação de grupo [21], como as citadas na Seção 2.3.

Sistemas Distribuídos podem ser síncronos, assíncronos ou parcialmente síncronos. Os sistemas síncronos são aqueles onde o tempo de execução das etapas de cada processo,

o tempo de transmissão de cada mensagem e a taxa de desvio do relógio local de cada processo possuem limites conhecidos [22]. Já nos sistemas assíncronos não se pode fazer qualquer consideração sobre esses três valores. Existem também os sistemas parcialmente síncronos, que, por sua vez, se subdividem em dois grupos: (i) onde os limites de tempo e desvio de relógios existem, mas não são conhecidos, a priori; (ii) onde os limites são conhecidos, mas só são garantidos a partir de algum tempo T desconhecido.

Existem dois estilos clássicos de arquitetura de sistemas distribuídos, resultantes da função dos processos individuais: cliente-servidor e *peer-to-peer* (P2P). O estilo historicamente mais importante, e que segue sendo prevalente entre os sistemas que nos servem em nosso dia-a-dia é o modelo cliente-servidor. Essa arquitetura apresenta uma divisão clara de responsabilidades e cargas de trabalho entre os processos que fornecem um serviço, chamados servidores, e os processos que consomem esse serviço, chamados clientes. Num sistema cliente-servidor é comum que os processos clientes e servidores estejam em máquinas distintas que se comunicam por meio de uma rede de computadores, apesar dessa separação não ser necessária. O mesmo processo que atua como servidor atendendo a uma requisição, também pode atuar como cliente em outra requisição, ao solicitar uma operação de outro servidor.

Uma das funções da arquitetura cliente-servidor é o compartilhamento de recursos, como equipamentos, dados ou programas. Nesse modelo os servidores têm um papel fundamental, são eles que gerenciam os recursos, cujo acesso será compartilhado com um número potencialmente grande de clientes por meio do conjunto de operações que disponibiliza. Os servidores são processos passivos, ou seja, só executam alguma função quando demandados, e funcionam continuamente aguardando a chegada de requisições dos clientes. Os processos clientes não compartilham seus recursos, em vez disso, utilizam os recursos disponibilizados pelos servidores, enviando-lhes mensagens com pedidos e recebendo mensagens com suas respostas.

A arquitetura P2P se caracteriza pela união de processos com as mesmas responsabilidades e funções semelhantes, que, formando uma rede, colaboram entre si como pares a fim de prestar um serviço [19]. Em termos práticos, todos os processos participantes executam o mesmo programa e oferecem o mesmo conjunto de interfaces uns para os outros. A ideia que levou ao desenvolvimento de sistemas P2P foi que a rede e os recursos computacionais pertencentes aos clientes de um serviço também poderiam ser utilizados para suportar esse mesmo serviço.

2.5 Tolerância a falhas e replicação

O funcionamento correto de um sistema distribuído é ameaçado quando ocorre uma falha, que pode levar a um defeito em qualquer um de seus componentes. Isso inclui todo hardware e software onde seus processos executam, assim como, a rede que os interliga [19]. As principais classes de falhas de um sistema distribuído são as falhas de tempo, falhas por omissão, falhas por colapso, falhas de resposta e as falhas arbitrárias [19].

- **Falhas de tempo**: ocorrem quando o servidor ou o meio de comunicação executa suas tarefas, porém o resultado é entregue fora de um intervalo de tempo-real pré-definido. Quando essa falha ocorre no servidor, também é chamada de falha de desempenho.
- **Falhas por omissão**: são aquelas em que o cliente não recebe o resultado de sua solicitação. Essas falhas podem acontecer por diversos motivos, como por exemplo:
  - A requisição não foi transmitida com sucesso do cliente para o servidor.
  - O servidor não conseguiu realizar o serviço.
  - A resposta não foi transmitida com sucesso do servidor para o cliente.
- **Falhas por colapso (crash)**: similar a falha por omissão, porém, nesse caso, o processo falho perde seu estado e não volta mais a participar do sistema.
- **Falhas de resposta**: ocorre quando o servidor atende à requisição incorretamente. Essas falhas podem ocorrer de duas formas, quando o servidor devolve uma resposta errada para o cliente, ou, quando o servidor reage de forma inesperada a uma requisição alterando seu estado interno para um estado indesejado.
- **Falhas arbitrárias**: são aquelas em que qualquer tipo de erro pode ocorrer. Por exemplo, um processo pode atribuir valores incorretos a seus dados, retornar um valor errado em resposta a uma invocação, ou simplesmente não responder.

Tolerância a falhas é a propriedade que alguns sistemas computacionais possuem de continuar operando sem defeitos mesmo após a ocorrência de falhas. Um sistema pode se tornar tolerante a falhas com o uso de componentes redundantes [19]. A técnica de replicação dos componentes críticos é um mecanismo de tolerância a falhas que aumenta simultaneamente a confiabilidade e a disponibilidade de um sistema [23]. Quando um processo cliente emite solicitações para serviços remotos, esses serviços processam a solicitação, possivelmente modificam seu estado interno e enviam de volta uma resposta ao cliente. Em um serviço replicado, existem várias cópias do processo servidor, chamadas

de réplicas, e o sistema deve garantir que o estado interno de cada réplica permaneça consistente com o das outras. Existem duas técnicas gerais para replicação: replicação ativa e passiva.

Na replicação passiva, também chamada de abordagem de líder-seguidores (*primary-backup*) [7], apenas uma réplica (primária) executa as ações e atualiza as outras réplicas (*backups*). Essa técnica requer (1) selecionar um primário e (2) atualizar os *backups* adequadamente. A replicação passiva permite que ações não determinísticas sejam executadas e demanda menos poder de processamento do que a replicação ativa [23]. Apesar de suas vantagens, a replicação passiva é comumente criticada por ter na réplica primária um ponto sensível de falhas e às vezes mostrar um desempenho extremamente ruim quando essas falhas acontecem.

Na replicação ativa, também chamada de Replicação Máquina de Estados (RME) [1, 24, 25], todas as réplicas partem do mesmo estado inicial e executam as mesmas ações, na mesma ordem e de maneira determinística, permanecendo assim idênticas. A principal vantagem da replicação ativa é que ela mantém um bom tempo de resposta em caso de falhas. Mesmo quando algumas réplicas falham, o serviço continua sendo prestado, muitas vezes sem que os clientes cheguem a perceber a ocorrência de qualquer falha. A implementação da replicação ativa depende de um protocolo de comunicação que garanta que de todas as requisições dos processos clientes sejam entregues a todas as réplicas exatamente na mesma ordem, mesmo na presença de falhas. A seção seguinte apresenta a técnica de RME em maiores detalhes.

2.6 Replicação Máquina de Estados (RME)

A Replicação Máquina de Estados é uma tradicional abordagem para o projeto de sistemas tolerantes a falhas [1]. A técnica consiste em replicar o servidor e coordenar a execução de comandos dos clientes entre as réplicas [1, 24, 25]. Dessa forma uma RME replicada com $\boldsymbol{n}$ réplicas é capaz de tolerar falhas em até $\boldsymbol{f}$ servidores, sendo que a relação entre $\boldsymbol{f}$ e $\boldsymbol{n}$ é dada pelo protocolo aplicado.

O serviço definido por uma RME consiste em variáveis que codificam o estado da aplicação e um conjunto de comandos que mudam esse estado (i.e., a entrada). Os comandos podem: (i) ler variáveis de estado, (ii) modificar variáveis de estado e (iii) produzir uma resposta para o comando (i.e., a saída). Contudo, devem ser determinísticos, ou seja, as mudanças no estado e a resposta do comando devem ser funções das variáveis de estado que o comando lê e do próprio comando.

A aplicação da RME fornece aos clientes a abstração de um sistema altamente disponível, enquanto esconde a existência de múltiplas réplicas operando coordenadamente. Este último aspecto é capturado pela propriedade de linearizabilidade.

Linearizabilidade. A linearizabilidade é um critério de consistência importante para RME [3]. Uma execução é linearizável se houver uma maneira de ordenar totalmente as operações de tal forma que (a) respeite a semântica dos objetos acessados pelas operações, expressa em suas especificações sequenciais; e (b) respeite a ordenação no tempo das operações. Existe uma ordem no tempo entre duas operações se uma operação termina em um cliente antes que a outra operação comece em outro cliente. As técnicas tradicionais de RME garantem a linearizabilidade pela forma como os comandos são propagados e executados pelas réplicas. Assim, toda réplica em operação deve: (i) receber todos os comandos; (ii) executar os comandos na mesma ordem; (iii) executar os comandos deterministicamente.

A aplicação da RME tradicional melhora a tolerância a falhas dos sistemas, ao custo de uma redução no desempenho devido à necessidade de coordenação entre as réplicas e execução sequencial dos comandos. Em muitos casos, servidores individuais não replicados superam o desempenho de conjuntos de réplicas oferecendo os mesmos serviços [11]. Essa vantagem foi atribuída ao aproveitamento, por parte do servidor não replicado, de todos os recursos dos processadores modernos que são essencialmente paralelos com seus múltiplos núcleos. Devido a essa característica, iniciou-se uma nova linha de estudos visando aumentar o aproveitamento do paralelismo do *hardware* moderno por parte das réplicas, tornando a RME em uma RME paralela, ou RMEP.

2.7 Replicação Máquina de Estados Paralela

Como vimos, a RME clássica subutiliza processares modernos com vários núcleos, pois a estratégia adotada para manter a consistência, foi a entrega dos comandos na mesma ordem total em todas as réplicas, e sua execução determinística e sequencial. A ideia chave por trás da RMEP consiste em permitir que dois ou mais comandos de uma RME executem paralelamente sempre que a ordem dessas execuções seja indiferente para o resultado final da máquina de estados [11]. Embora a execução paralela de comandos (multi-cores) possa resultar em não determinismo, Schneider F.B. [1] mostrou que comandos independentes podem ser executados simultaneamente sem violar a consistência forte entre as réplicas.

Os algoritmos de escalonamento que discutiremos exploram a concorrência entre comandos. Dois comandos entram em conflito se acessam o estado comum e pelo menos um dos comandos modifica o estado compartilhado. Seja C o conjunto de comandos possíveis

e $\{\#_C \subseteq C \times C\}$ a relação de conflito entre os comandos. Se $\{c_i, c_j\} \in \#_C$, então os comandos c_i e c_j conflitam e as réplicas devem serializar sua execução; caso contrário, as réplicas podem executar c_i e c_j simultaneamente.

As técnicas mais recentes propostas para RMEP diferem entre si principalmente na forma de como os comandos são escalonados para execução por um conjunto de *threads*. Algumas abordagens foram sugeridas na literatura para executar comandos independentes de forma simultânea, por meio do uso de conhecimentos sobre a semântica da aplicação (e.g., [15, 6, 10, 11, 12, 26, 27, 28]). Com o objetivo de melhorar o desempenho da RME, essas técnicas se baseiam na suposição de que se dois comandos podem ser comutados (por exemplo, incrementar um contador), então réplicas diferentes podem executá-los em ordem diferente e ainda atingir o mesmo estado final. Esses trabalhos visam reduzir o atraso na entrega de um comando, evitando um protocolo de ordenação caro, quando possível. Na Seção 3 apresentaremos as principais classes de escalonamento para RMEP, juntamente com os trabalhos mais atuais em cada classe.

2.8 *Blockchain* e Sistemas de Pagamentos

A rápida ascensão do mercado de cripto-ativos, desde o surgimento de seu precursor e mais proeminente membro, o Bitcoin [29], difundiu o uso do termo *blockchain* (corrente de blocos) que vem sendo utilizado para denominar conceitos diversos como estruturas de dados e modelo de sistemas.

Estrutura de dados: um *blockchain*, como estrutura de dados, tipicamente denomina uma lista encadeada de blocos de dados, ligados por ponteiros criptográficos (*hash pointers*). Novos dados são agrupados formando um novo bloco que é ligado ao final da cadeia de blocos pré-existente por meio um ponteiro de *hash*. O bloco mais recente aponta para o bloco imediatamente anterior (Figura 2.1), e assim por diante, até chegar no primeiro bloco, o bloco gênesis (origem da cadeia). Os ponteiros de *hash* tanto identificam um bloco unicamente, quanto apresentam o seu *hash* criptográfico. O valor desse *hash* serve para verificar se houve ou não alterações nos dados referenciados. Como os blocos são encadeados por esses ponteiros, não é possível alterar, incluir ou excluir qualquer informação da cadeia de blocos, sem que a modificação seja facilmente detectada. Por tratar-se de uma estrutura que visa garantir a imutabilidade dos dados, essas estruturas costumam ser do tipo *append-only*, ou seja, as únicas operações permitidas num *blockchain* são a leitura e a adição de novos blocos de dados.

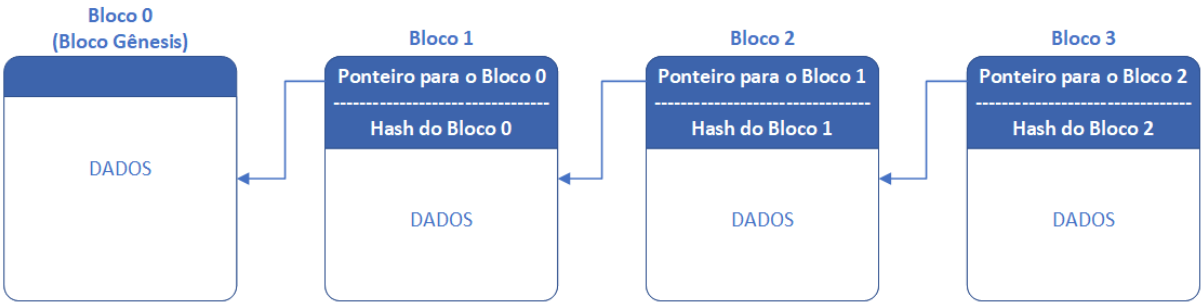


Figura 2.1: Encadeamento de blocos.

Modelo de sistema: como modelo de sistema, um *blockchain* costuma ser um sistema distribuído, P2P, onde os pares colaboram entre si, seguindo um protocolo predefinido, a fim de evoluir em consenso e de forma sincronizada uma corrente de blocos. Uma característica típica desses sistemas é a independência operacional dos pares. Ou seja, mesmo que todos os nós ativos e corretos da rede par-a-par cooperem seguindo o mesmo protocolo, cada nó mantém e evolui de forma autônoma, e privativa, sua própria cópia integral da cadeia de blocos, desde o bloco inicial (gênesis). O protocolo de comunicação e consenso é que garante a consistência na evolução da corrente de blocos e a correção de possíveis desvios, caso ocorram. Essa resiliência é uma das propriedades mais destacadas dos sistemas *blockchain*. O Bitcoin [29], por exemplo, possui milhares de nós ativos, porém, mesmo que 99% desses nós sofressem uma parada total simultânea, teoricamente, o sistema continuaria operando com os nós restantes.

2.8.1 Classificação dos sistemas Blockchain

Com base na maneira como o blockchain evoluiu nos últimos anos, ele pode ser dividido em várias categorias com atributos distintos, embora às vezes parcialmente sobrepostos [30]. Aqui vamos apresentar apenas duas categorias que consideramos as principais.

Blockchains Públicos. Os *blockchains* públicos, também chamados de não permissionados, não pertencem a ninguém. Eles são abertos ao público e qualquer pessoa pode participar como um nó no processo de tomada de decisão, ou sair da rede a qualquer momento, sem qualquer restrição. Uma vez conectado à rede, o novo nó receberá de seus pares todos os dados que precisa para participar ativamente do protocolo em vigor. Esses sistemas em regra utilizam um cripto-ativo subjacente para criar incentivos econômicos que motivem os operadores a dedicar recursos para a sustentação da própria rede. O Bitcoin foi o primeiro e é o maior exemplo de *blockchain* público da atualidade, com uma média de 10.000 nós conectados nos anos de 2019, 2020 e 2021 [31]. A maioria dos

blockchains públicos têm por objetivo a manutenção do próprio cripto-ativo subjacente, como o Bitcoin [29]. Porém, outros utilizam a cripto-ativo como instrumento de incentivo econômico para a manutenção da rede e regulação do uso de seus serviços (contratos inteligentes), como é o caso do Ethereum [32].

Blockchains Privados. Os *blockchains* privados estão abertos apenas para um grupo de indivíduos ou organizações que decidiram compartilhar a corrente de blocos entre si. A entrada de participantes é controlada. Novos nós precisam ser autorizados, de alguma maneira, para que possam participar da rede e ter acesso aos dados já produzidos. Cada plataforma privada controla a entrada de novos participantes à sua maneira. As implementações desses *blockchains* são bastante heterodoxa. Existem implementações que partem de uma plataforma pública e agregam algumas funções úteis ao mundo empresarial, como é o caso do Quorum [33] e do MultiChain [34]. Outras implementações são completamente independentes como o Hyperledger Fabric [35] e o Tendermint [36].

2.8.2 Sistemas de pagamentos

Economicamente falando, um sistema de pagamento consiste em um conjunto de instrumentos, procedimentos e, normalmente, sistemas de transferência interbancária de fundos que garantem a circulação de dinheiro [37]. Os sistemas de pagamento são normalmente baseados em um acordo entre os participantes e o operador do acordo, no qual todos os participantes devem confiar. Tecnologicamente falando, um sistema de pagamento é um sistema computacional que consegue gerenciar a representação digital de valores, garantindo sua propriedade exclusiva e a possibilidade de transferência dessa propriedade.

Décadas atrás, a solução encontrada pelos mercados para viabilizar a movimentação digital de valores financeiros foi a intermediação. Ou seja, as instituições financeiras ocupam o papel de intermediário (ou operador confiável) numa transação entre duas partes, garantindo para o recebedor e para o pagador que a transação seja confirmada dentro dos parâmetros acordados e que o valor transferido não possa ser reutilizado pelo mesmo pagador em outra transação (gasto duplo). Essa intermediação está baseada num sistema de confiança, onde as partes envolvidas em uma transação eletrônica devem confiar no mesmo operador financeiro, ou seus operadores financeiros devem confiar num terceiro operador financeiro comum, e assim por diante, para que a transação seja consolidada. Esse mecanismo de fato viabilizou a movimentação digital de valores, mas trouxe consigo custos e dificuldades [38], por exemplo, custos operacionais, custos de reconciliação, baixa transparência para as partes, alto tempo de confirmação – principalmente quando falamos de transações internacionais ou em países com o sistema financeiro pouco desenvolvido, – e a possibilidade do cancelamento mesmo contra a vontade das partes.

O Bitcoin foi o primeiro sistema de pagamentos[2] eletrônico desintermediado que se tem notícia [29]. Sua maior contribuição foi a forma prática com a qual resolveu o problema do gasto duplo, utilizando um *blockchain* associado a um protocolo específico. O problema do gasto duplo diz respeito à incapacidade de se garantir a transferência e a propriedade exclusiva de um artefato digital simultaneamente. Uma cédula em papel-moeda é uma representação física, ou *token* físico, de um determinado valor em uma determinada moeda. Quando usamos a cédula para pagar por algum produto, ou serviço, o valor que ela representa é transferido juntamente com sua posse para o vendedor. Para gastar o valor representado pela mesma cédula mais de uma vez, seria preciso copiá-la, o que é ilegal e passível de detecção devido às tecnologias utilizadas na produção do numerário físico. Já a cópia de um artefato digital é indistinguível de seu original, permitindo assim, a circulação indetectável de um número potencialmente ilimitado de cópias do mesmo artefato.

2.8.3 Limitações de desempenho

A tecnologia *blockchain* tem o potencial de melhorar a vida das pessoas de várias formas, inclusive por meio de sistemas de pagamentos mais baratos e eficientes (sua aplicação mais fundamental, vide Bitcoin [29]). Contudo, a adoção dessa tecnologia em sistemas de alto impacto social tem sido adiada devido as suas limitações atuais. Dentre essas limitações encontra-se em destaque o baixo desempenho.

Nas redes de Blockchain com consenso baseado em prova de trabalho – PoW [29] – (e.g. Bitcoin e Ethereum), percebe-se um alto tempo de confirmação das transações (finalidade) e uma baixa taxa de processamento de transações por segundo, independentemente do número de participantes. Por exemplo, no Bitcoin este tempo é configurado para 10 minutos por bloco, sendo necessário 6 blocos para garantir com alta probabilidade que as transações não serão desfeitas. Já nas redes com consensos derivados do Paxos [39, 40], baixos tempos de finalidade e altas taxas de processamento só são encontradas quando se mantém um reduzido número de participantes no consenso como são os casos do Tendermint e do Hyperledger Fabric.

Conforme já discutido, o funcionamento de um *blockchain* possui muitas similaridades com o de uma RME, principalmente quando consideramos os *blockchains* privados. Um trabalho recente [41] mostra também que o modelo de execução sequencial de uma RME quando aplicado a um sistema de pagamento (moeda virtual), causa um impacto significativo em seu desempenho, visto que a execução da maioria das requisições envolve operações custosas como a verificação de assinaturas criptográficas. Na Seção 5.2, apresentaremos um estudo de caso e analisaremos como uma RMEP com a abordagem de

[2]Oficialmente, no Brasil, o Bitcoin não é um sistema de pagamentos, porque o *token* movimentado por ele é considerado um ativo digital e não dinheiro.

escalonamento híbrido pode aumentar o desempenho de um *Blockchain* para sistemas de pagamento, por meio de um estudo de caso com um protótipo de sistema de pagamento, o *ParallelCoin*.

2.9 Considerações finais

Neste capítulo vimos os fundamentos teóricos para a construção de uma RME e uma RMEP. Vimos também uma breve definição de *Blockchain*, sua relevância no futuro dos sistemas de pagamentos e suas principais limitações de desempenho. No Capítulo 3, serão apresentados o estado da arte e os trabalhos relacionados a Replicação Máquina de Estados Paralela (RMEP).

Capítulo 3

Trabalhos Relacionados

Este capítulo apresenta os trabalhos relacionados e o estado da arte em técnicas de Replicação Máquina de Estados Paralela (RMEP). As abordagens de escalonamento tardio e antecipado são detalhadas com maior profundidade, pois servirão como base para o entendimento do escalonamento híbrido que será apresentado em capítulo posterior.

3.1 Escalonamento tardio

Nessa categoria de protocolos, as réplicas recebem os comandos na mesma ordem total e, em cada réplica, uma *thread* paralelizadora encaminha as requisições recebidas para serem executadas por um conjunto de *threads* executoras. O processo de escalonamento deve disparar a execução de cada comando levando em consideração a ordem de recebimento e as dependências entre os comandos. Ou seja, se dois comandos c_1 e c_2 são dependentes entre si e o comando c_1 foi recebido antes do comando c_2, então o comando c_1 deve ser executado antes do comando c_2. Se os comandos forem independentes entre si, poderão ser executados em qualquer ordem, inclusive em paralelo. A característica fundamental deste modelo é que todas as decisões de escalonamento são tomadas após a ordenação e transferência das requisições pela camada de comunicação (*multicast* atômico). A seguir são apresentadas duas técnicas que se encaixam neste modelo.

3.1.1 CBASE

Em 2004, Kotla et al [10] propuseram o CBASE, uma estratégia de escalonamento tardio que ao ser aplicada em conjunto com as técnicas de replicação ativa [1, 42] permite a execução paralela de algumas operações criando uma RMEP. O CBASE utilizou um grafo e algum conhecimento sobre a semântica da aplicação para controlar as dependências entre as requisições. Sua estrutura de escalonamento em cada réplica é composta por uma

thread do paralelizador e $\boldsymbol{T}$ *threads* de executores, como pode ser visto na Figura 3.1. Em cada réplica o paralelizador recebe os comandos em ordem total, entregues pela camada de comunicação e ordenação (*multicast* atômico) e os inclui em um grafo interno. Esse grafo é compartilhado com todas as *threads* executoras e é construído de tal forma que seus vértices representam os comandos e suas arestas representam as dependências entre os comandos.

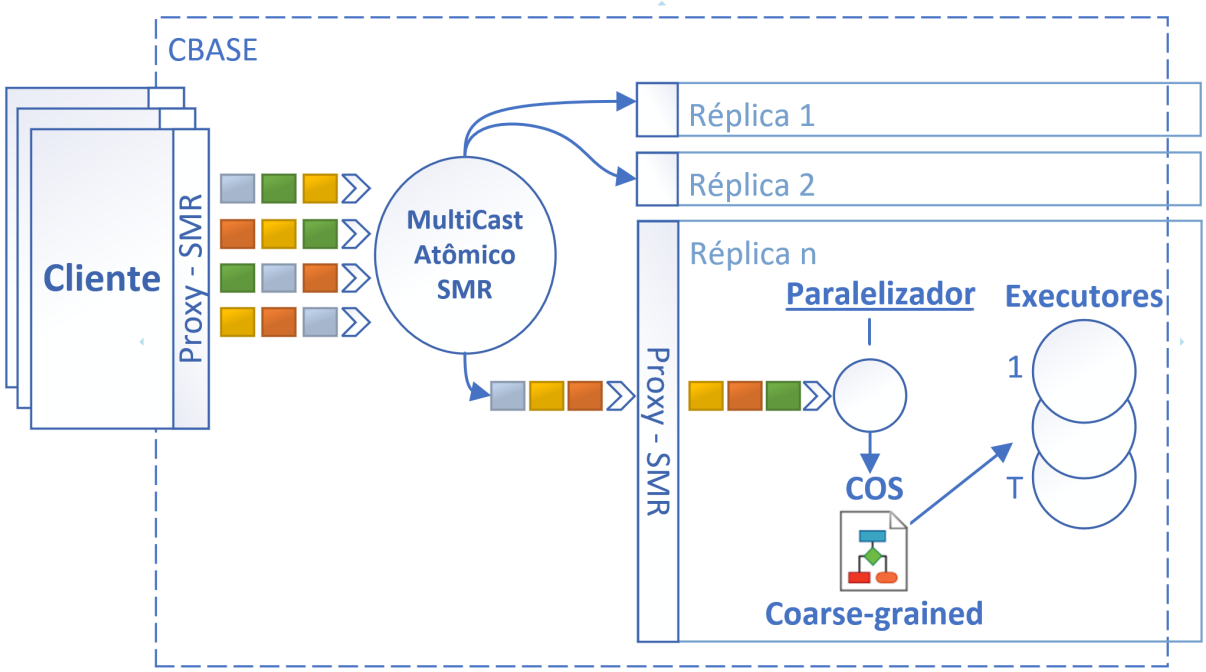


Figura 3.1: Visão geral do CBASE.

Cada executor opera em um laço infinito onde: (1) inspeciona o grafo em busca do comando que será processado (respeitando as relações de dependência), (2) executa o comando e, (3) remove o comando executado do grafo. Como todas as threads do escalonamento concorrem pelo mesmo grafo, o CBASE [10] utilizou primitivas de sincronização para garantir o acesso exclusivo ao grafo de dependências. Com esse bloqueio integral do grafo, uma única *thread* por vez tem acesso às suas estruturas, enquanto todas as outras aguardam a liberação do bloqueio.

De modo geral, o paralelizador insere os comandos, um a um, em ordem total no grafo, criando as relações de dependência entre os novos comandos e os comandos já inseridos. Quando um comando fica pronto, ou seja, livre de dependências, os executores são sinalizados e um deles adquire o bloqueio do grafo como um todo para varrê-lo em busca do comando a ser executado. Durante a execução do comando, o grafo é liberado para que outras *threads* possam trabalhar nele. Ao final da execução o próprio executor torna a concorrer pelo grafo, agora, a fim de remover o comando executado e as arestas de

dependências associadas, liberando espaço para novos comandos e possivelmente tornando outros comandos prontos para execução.

3.1.2 CBASE - Revisitado

Em 2019, Escobar et al. [14] demonstrou que o protocolo CBASE [10], ao tratar o grafo de dependências como uma única seção crítica, tornou-o um gargalo na execução da réplica. Para demonstrar seu ponto, por meio de uma refatoração de código, o artigo separou o grafo de dependências do código do paralelizador e generalizou as funcionalidades do grafo de dependências em um tipo abstrato de dados chamado *Conflict Ordered Set (COS)*. O COS representa o conjunto ordenado de conflitos e controla a ordem entre comandos conflitantes (dependências). Este tipo abstrato é definido por três primitivas conforme a especificação abaixo:

- $insert(c \in C)$ que insere o comando c na estrutura de dados;
- $get() : c \in C$ que retorna c se, e somente se:
  - c está na estrutura de dados,
  - nenhuma chamada anterior ao método $get()$ retornou c, e
  - não existe um comando c', inserido antes de c, tal que $(c, c') \in \#_C$;
- $remove(c \in C)$ que remove c da estrutura de dados.

O desempenho do escalonamento tardio foi então testado com três diferentes implementações do COS, que diferiam em suas técnicas de sincronização:

- *Coarse-Grained Lock*: essa é a implementação original de CBASE [10], onde o acesso ao grafo é realizado por uma *thread* de cada vez, ou seja, a sessão crítica aqui é o grafo como um todo.

- *Fine-Grained Lock*: nessa implementação, a concorrência no acesso ao grafo é controlada por meio de técnicas de acoplamento de *locks*, reduzindo o tamanho da seção crítica a vértices individuais;

- *LockFree*: nessa implementação do COS o acesso concorrente ao grafo é totalmente livre de bloqueios e seções críticas.

A Figura 3.2 mostra um esquema simplificado do escalonamento tardio com o desacoplamento entre o COS e o código do paralelizador. A estratégia *Coarse-Grained Lock* para o COS foi apresentada na Seção 3.1.1 e pode ser resumida assim: o grafo é acessado por uma *thread* de cada vez. A estratégia *Fine-Grained Lock*, em vez de bloquear o grafo inteiro, cada operação bloqueia o primeiro nó do grafo e, então, para atravessar o conjunto ordenado de nós, a *thread* vai passo-a-passo bloqueando o nó sucessor antes de liberar o nó anterior. Esse procedimento é executado simultaneamente pelo paralelizador na inserção

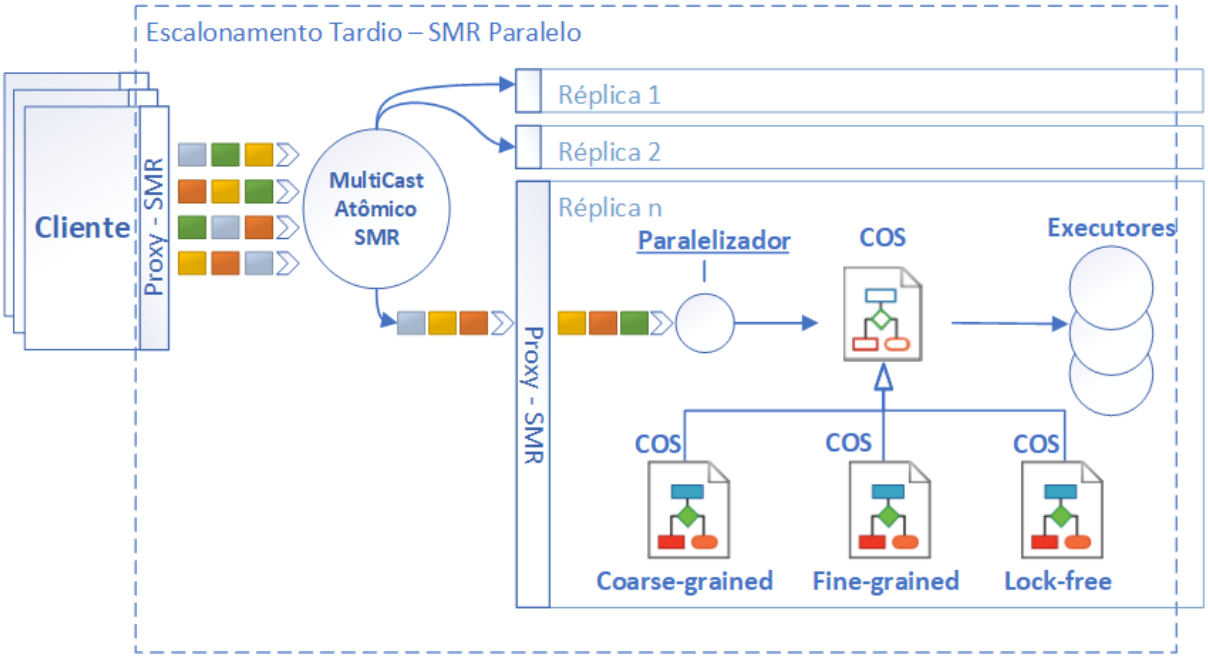


Figura 3.2: Escalonamento Tardio.

dos comandos e pelos executores, tanto na busca por comandos prontos para execução, quanto na exclusão dos comandos terminados. Já no COS *LockFree*, o paralelizador é a única *thread* que insere e remove os nós do grafo, os executores vasculham o grafo livremente em busca de comandos prontos para execução. Ao encontrar um nó "pronto", o executor altera seu estado de forma atômica para "em execução" e posteriormente para "logicamente removido".

Com a introdução do COS, foi possível testar isoladamente o impacto de diferentes técnicas de sincronização no desempenho do escalonamento tardio. Além disso, o algoritmo do paralelizador e dos executores no escalonamento tardio ficaram mais simples já que os requisitos de sincronização foram encapsulados na implementação do COS. O artigo também demonstrou que o escalonamento tardio com o COS *LockFree* apresentou um desempenho muito superior ao CBASE [10] e ao COS com *Fine-Grained Lock*.

3.1.3 Protocolo

Nesta sessão apresentamos detalhadamente a implementação do escalonamento tardio com o COS *LockFree* utilizada por Escobar et al. [14], pois seu conhecimento servirá como base para o entendimento do escalonamento híbrido.

Paralelizador e Executor

O Algoritmo 1 detalha o comportamento do paralelizador e dos executores baseados em COS. O paralelizador, quando instado pela camada de comunicação e ordenação (linha 7), reserva um espaço para a nova requisição, insere-a no COS e sinaliza para os executores caso o novo nó esteja livre de dependências (linhas 8-10). A função *insert* (linha 9) retorna o valor 1 se o nó recém-inserido estiver disponível para execução, ou, o valor 0, caso contrário. As *threads* dos executores, por sua vez, executam um laço infinito (linha 12), onde esperam pela sinalização de comandos prontos, recuperam um comando pronto do COS e executam-no (linhas 13-15). Em seguida, o comando é removido do COS, um espaço é liberado e caso a remoção deste comando tenha liberado outros comandos para execução, uma nova sinalização de comandos prontos é enviada (linhas 16-18). A função *remove* (linha 16) retorna a quantidade de nós que ficaram prontos para a execução com a remoção lógica de c.

Algoritmo 1 Escalonamento tardio: paralelizador e executoras

```
1:  Constantes e estrutura de dados:
2:    T: set of working thread identifiers
3:    COS: the conflict-ordered set
4:    space ← new Semaphore(maxSize)        {controle do espaço livre no grafo}
5:    ready ← new Semaphore(0)              {controle dos nós prontos}
6:  Paralelizador executa assim:
7:    onDeliver(req):                       {a cada nova requisição}
8:      space.down()                        {aguarde um espaço disponível}
9:      rdy ← COS.insert(req)               {insira a requisição no grafo}
10:     ready.up(rdy)                       {sinalize para o controle de nós prontos}
11: Cada thread executora (t_id ∈ T) executa assim:
12:   while true do                         {laço infinito}
13:     ready.down()                        {aguarde nós prontos}
14:     c ← COS.get()                       {recupere um comando livre para execução}
15:     execute(c)                          {execute c}
16:     rdy ← COS.remove(c)                 {remova c do grafo}
17:     space.up()                          {libere um espaço para novos comandos}
18:     ready.up(rdy)                       {sinalize para o controle de nós prontos}
```

COS LockFree (sem bloqueios)

Aqui apresentamos a implementação do COS sem bloqueios (*LockFree*) utilizada por Escobar et al [14]. O algoritmo usa tipos atômicos[1] nativos e uma operação atômica de

[1]Tipos atômicos, presentes em algumas linguagens de programação modernas, oferecem operações não preemptíveis de atualização de suas variáveis internas.

comparação e ajuste. É assumido que as chamadas à função *insert* são sequenciais, de acordo com a ordem definida pela difusão atômica (ver Algoritmo 1). Isso garante que as réplicas lidem com comandos conflitantes de maneira consistente. Enquanto as invocações da função *insert* são sequenciais (entre si), as invocações das funções *get* e *remove* são concorrentes com quaisquer invocações.

O algoritmo constrói um grafo acíclico direcionado (DAG) para representar os comandos e suas dependências. Cada $Node$ do grafo (Algoritmo 2, linha 2) contém um comando c, um campo atômico com o estado do nó, uma lista *depOn* de referências aos nós dos quais este nó depende, uma lista $depMe$ de referências aos nós que dependem deste nó, e o campo nxt, que representa a ordem total entre os comandos. Cada comando é embutido em um nó que transita pelos seguintes estados, na ordem mostrada:

- $\perp$ (iniciando): o nó está sendo inserido no gráfico;
- wtg (em espera): o nó já foi inserido, mas depende de outros nós para executar;
- rdy (pronto): todas as dependências foram resolvidas e seu comando pode ser executado;
- exe (executando): o comando foi levado para execução por alguma *thread* executora;
- rmd (removido): o nó foi logicamente removido da estrutura de dados do COS.

A estratégia usada para permitir operações sem bloqueios é alterar a topologia do grafo exclusivamente pelo paralelizador, na operação de inserção. Enquanto isso, as operações *get* e *remove* podem recuperar comandos para execução, bem como remover nós executados. Bastando para isso, que alterem apropriadamente e de maneira atômica o estado do nó, sem modificar qualquer outro campo ou a topologia do grafo. A operação de remoção é lógica, pois marca o nó como removido enquanto a remoção real do grafo ocorre durante a posterior inserção de um novo nó, usando a técnica da ajuda [43], onde uma operação ajuda a outra a realizar suas modificações.

A operação *insert* (Algoritmo 2, linha 13) assume que há espaço no grafo para criar um nó (ver Algoritmo 1). O novo nó é criado com o comando c (Algoritmo 3, linhas 1 e 2). A função *calculateDependencies* (Algoritmo 3, linha 12) percorre os nós em N, do mais antigo ao mais novo, e constrói as dependências necessárias. Se algum nó $n' \in N$, que ainda não foi removido logicamente, entrar em conflito com o novo nó n_n, então as linhas 18 e 19 inserem arestas em ambas as direções para representar esta dependência. Durante uma remoção física (Algoritmo 3, linha 16) os nós logicamente removidos tem suas arestas de entrada excluídas (Algoritmo 3, linhas 22 até 24).

Após o "cálculo das dependências", no Algoritmo 2, o novo nó é incluído na lista N para se tornar acessível no grafo, e seu estado é alterado para *em espera* (Algoritmo 2, linhas 16 e 17). Finalmente, a função *testReady* (Algoritmo 3, linha 7) verifica a existência de dependências. Se nenhuma for encontrada, o nó é atomicamente atualizado para a situação de pronto (Algoritmo 3, linha 9). Quando um nó fica pronto já na inserção, um

Algoritmo 2 LockFree COS: tipos de dados e operações

```
1:  Tipos de dados:
2:    Node : {
3:        c : Command,
4:        atomic st : {⊥, wtg, rdy, exe, rmd}        {o nó pode estar iniciando, ...}
5:                                                   {...aguardando, pronto, executando, removido}
6:        depOn : set of NodeRef                     {nós que este nó depende}
7:        depMe : set of NodeRef                     {nós que dependem deste nó}
8:        nxt : NodeRef                              {próximo nó em ordem de chegada}
9:    }
10:   NodeRef : atomic reference to Node
11: Variáveis:
12:   N : NodeRef                                    {nós no COS}
13: insert(c : Command): int
14:   n_n ← createNode(c)                            {cria um novo nó}
15:   n ← calculateDependencies(N, n_n)              {computa as dependências}
16:   if n = ⊥ then N ← n_n else n.nxt ← n_n
17:   n_n.st ← wtg                                   {agora o nó está aguardando...}
18:   return testReady(n_n)                          {...e é testado quanto a sua prontidão}
19: get(): NodeRef                                   {assumindo que existe um nó pronto}
20:   n ← N                                          {inicie a busca pelo nó pronto}
21:   while n ≠ ⊥ do                                {considere cada nó, por ordem de chegada}
22:     if compareAndSet(n.st, rdy, exe) then        {se o nó estiver pronto, marque-o...}
23:       return n                                   {...e retorne-o}
24:     n ← n.nxt                                    {caso contrário, vá para o próximo nó}
25: remove(n : NodeRef): int                         {assumindo que existe um nó n e n.st = exe}
26:   n.st ← rmd                                     {marque n como logicamente removido}
27:   rdys ← 0                                       {contador de nós prontos}
28:   for all n_i ∈ n.depMe do                       {para todos os nós que dependem de n}
29:     rdys ← rdys + testReady(n_i)                 {verifique se n_i está pronto e conte}
30:   return rdys                                    {retorne o número de nós prontos}
```

sinal é retornado da linha 10 do Algoritmo 3 para a linha 18 do Algoritmo 2, e de lá para a linha 9 do Algoritmo 1, onde é usado em seguida para sinalizar os executores.

A operação *get* (Algoritmo 2, linha 19) retorna um nó do grafo que esteja pronto para executar. Quando esta operação é chamada, existe ao menos um nó pronto no grafo devido ao Algoritmo 1, linhas 10 e 18. O primeiro nó testado atomicamente cujo estado seja *rdy* (ou seja, pronto) é definido como *exe* (ou seja, em execução) e retornado. Como a inserção segue a ordem de entrega, o nó retornado é o mais antigo dentre os nós prontos para execução. Devido à atomicidade de *compareAndSet* (Algoritmo 3, linha 3) o nó é retornado por no máximo uma chamada da função *get*.

A operação *remove* (Algoritmo 2, linha 25) marca o nó como removido logicamente

Algoritmo 3 LockFree COS: operações auxiliares

```
1: createNode(c : Command): NodeRef
2:   return reference to new Node{c, ⊥, ∅, ∅, ⊥}
3: compareAndSet(a, b, c): boolean
4:   return atomic { if a = b then a ← c; true else false }
5: conflict(n_i, n_j : Node): boolean
6:   return (n_i.c, n_j.c) ∈ #_C                    {este par está no conjunto de conflitos?}
7: testReady(n : NodeRef) : 0, 1
8:   if {n_i ∈ n.depOn | n_i.st ≠ rmd} = ∅ then     {se n não tem dependências}
9:     if compareAndSet(n.st, wtg, rdy) then        {troca de wtg para rdy}
10:      return 1
11:  return 0
12: calculateDependencies(N, n_n : NodeRef) : NodeRef
13:  n' ← n ← N                                     {n' e n recebem N}
14:  while n' ≠ ⊥ do                                {considere cada nó por ordem de chegada}
15:    if n'.st = rmd then                          {se n' foi logicamente removido...}
16:      helpedRemove(N, n', n)                     {...então remove-o do grafo}
17:    else if conflict(n', n_n) then               {caso contrário, se há um conflito com n_n...}
18:      n'.depMe ← n'.depMe ∪ {n_n}                {...agora n' sabe que n_n depende dele...}
19:      n_n.depOn ← n_n.depOn ∪ {n'}               {...e n_n sabe que depende de n'}
20:    n ← n'; n' ← n'.nxt                          {vá para o próximo nó}
21:  return n
22: helpedRemove(N, n', n : NodeRef)
23:  for all n_i ∈ n'.depMe do                      {para todo nó que depende de n'}
24:    n_i.depOn ← n_i.depOn \ {n'}                 {remove n' de suas dependências}
25:  if !compareAndSet(N, n, n'.nxt) then           {se n não é o primeiro nó}
26:    atomic { n.nxt ← n'.nxt }                    {remova n' do grafo}
```

atribuindo *rmd* ao seu estado. Em seguida, verifica se cada um dos nós dependentes ficou pronto (linhas 28 e 29). As arestas do nó mais antigo (aquele removido logicamente) para os nós mais novos (*depMe*) identificam quais nós podem ficar prontos quando essa dependência for resolvida. As arestas na outra direção (*depOn*) permitem avaliar se um nó tem todas as dependências resolvidas, ou seja, se todos os nós dos quais depende estão removidos logicamente (Algoritmo 3, linha 8). Com isso, a operação de remoção lógica é capaz de sinalizar novos nós prontos apenas verificando o estado dos nós referidos, sem qualquer modificação na estrutura do grafo, que é deixada para a próxima execução da função *insert* → *calculateDependencies* → *helpedRemove*.

3.1.4 Discussão de Corretude

O principal argumento para a corretude é que a operação *insert* executa todas as modificações estruturais para o grafo e suas invocações são sequenciais (de acordo com a ordem de entrega). As operações *get* e *remove* não alteram a topologia do grafo, ou seja, não criam nem excluem nós nem arestas. Essas operações marcam os nós como tomados para execução ou como logicamente removidos, usando o atributo atômico *st* de *Node*. Como discutido, a operação *insert* de cada novo nó é responsável por remover do grafo todo e qualquer nó logicamente removido. Isso exclui qualquer possível inconsistência topológica devido à simultaneidade.

As travessias do grafo são executadas apenas pelas operações *insert* e *get*. Essas travessias simultâneas são consistentes, apesar da operação *insert* poder alterar a topologia do grafo, uma vez que ambas as operações atravessam os nós na mesma ordem. Se a operação *insert* realizar uma modificação atômica na referência ao próximo nó, seja pela remoção ou inserção de um nó, a operação *get* usará a referência anterior ou a nova. Em ambos os casos, o resultado da função *get* será o mesmo.

Considere a função *get* no momento em que atravessa o grafo, passando por um nó removido logicamente. Como o nó já foi executado, não será retornado. Mesmo se este nó for separado do grafo, ele ainda se refere corretamente ao próximo nó, e como um esquema de coleta de lixo é assumido, o nó removido do grafo permanecerá disponível na memória enquanto existirem referências a ele - por exemplo, pela thread executando *get* - assim, o percurso do *get* continuará sem erros.

A operação *remove* marca atomicamente o nó referenciado como logicamente removido, não percorrendo o gráfico. Ela também lê as informações dos nós dependentes daquele que está sendo marcado, para verificar se algum está pronto para ser executado. Como essas operações podem apenas alterar de forma atômica o estado dos nós, elas não afetam outras operações. Além disso, pela suposição do uso de um coletor de lixo, uma vez que os nós verificados são referenciados pelo nó logicamente removido, eles existem e a operação é bem-sucedida.

O progresso decorre do fato de que o grafo é um *Directed Acyclic Graph (DAG)*. Uma vez que os comandos podem depender apenas dos anteriores em conflito, de acordo com uma ordem total, dependências nunca construirão um ciclo (ou seja, o grafo é acíclico). Após a execução de um comando, ele removerá indutivamente as dependências de outros comandos, garantindo que haja pelo menos um comando que possa ser executado ou nenhum comando restante.

3.1.5 Exemplo de Execução

Para facilitar o entendimento vamos apresentar um exemplo de execução descrevendo a jornada de 4 comandos. Considere um sistema imaginário bi-particionado com operações de leitura e escrita locais (em partições individuais) e de escritas globais (em todas as partições). Em sistemas assim é comum que as operações locais de uma partição não conflitem com as operações locais de outra partição. No nosso sistema hipotético as escritas locais conflitam com as escritas e as leituras da mesma partição. As leituras locais não conflitam entre si e as escritas globais conflitam com todas as operações.

Vamos acompanhar 4 operações específicas O_{R1}, O_{R2}, O_{W1} e O_{Wg} que representam respectivamente, uma leitura na partição 1, uma leitura na partição 2, uma escrita na partição 1 e uma escrita global. O quadro 1 da Figura 3.3 apresenta as quatro operações sendo enviadas por quatro processos clientes distintos. A difusão é realizada por uma biblioteca de *multicast* atômico padrão, típica dos sistemas de RME. A primitiva de comunicação é síncrona e bloqueante, sendo assim, a *thread* do cliente que realiza a difusão fica bloqueada aguardando a resposta do serviço remoto. A biblioteca de *multicast* atômico ordenará as requisições recebidas e as entregará a todas as réplicas do serviço na mesma ordem total.

Do quadro 2 em diante da Figura 3.3 mostramos o tratamento do lote de operações em uma das réplicas. A biblioteca de *multicast* atômico aciona o paralelizador da réplica com o lote de comandos na ordem total. O paralelizador então os insere no COS construindo as relações de dependência corretamente, dado o seu conhecimento sobre a semântica da aplicação, i.e., ele precisa saber quando duas operações conflitam[2].

No quadro 3, por existirem dois comandos livres de dependências O_{R1} e O_{R2}, dois executores são notificados e varrem o grafo por ordem de chegada capturando de modo exclusivo o primeiro comando pronto que encontrar e executando-o logo em seguida. Podemos perceber duas *threads* de execução processando os comandos O_{R1} e O_{R2} simultaneamente enquanto outros executores seguem aguardando por comandos prontos.

No quadro 4, ao encerrar o processamento dos comandos as *threads* executoras marcam-nos como logicamente removidos e verificam se essa ação foi suficiente para liberar um novo comando para processamento. No exemplo exposto, percebe-se que todas as dependências da operação O_{W1} foram logicamente excluídas. Neste ponto uma terceira *thread* executora é notificada, ela varre o grafo por ordem de chegada, captura exclusivamente o comando O_{W1} e já inicia seu processamento.

Quando O_{W1} termina, o executor remove-o logicamente do COS e verifica se essa remoção liberou outro comando para execução. O comando O_{Wg} é então liberado para execução e uma *thread* de execução é notificada. A *thread* despertada varrerá o grafo na

[2]Este conhecimento é adquirido através de uma função fornecida pelo programador da aplicação.

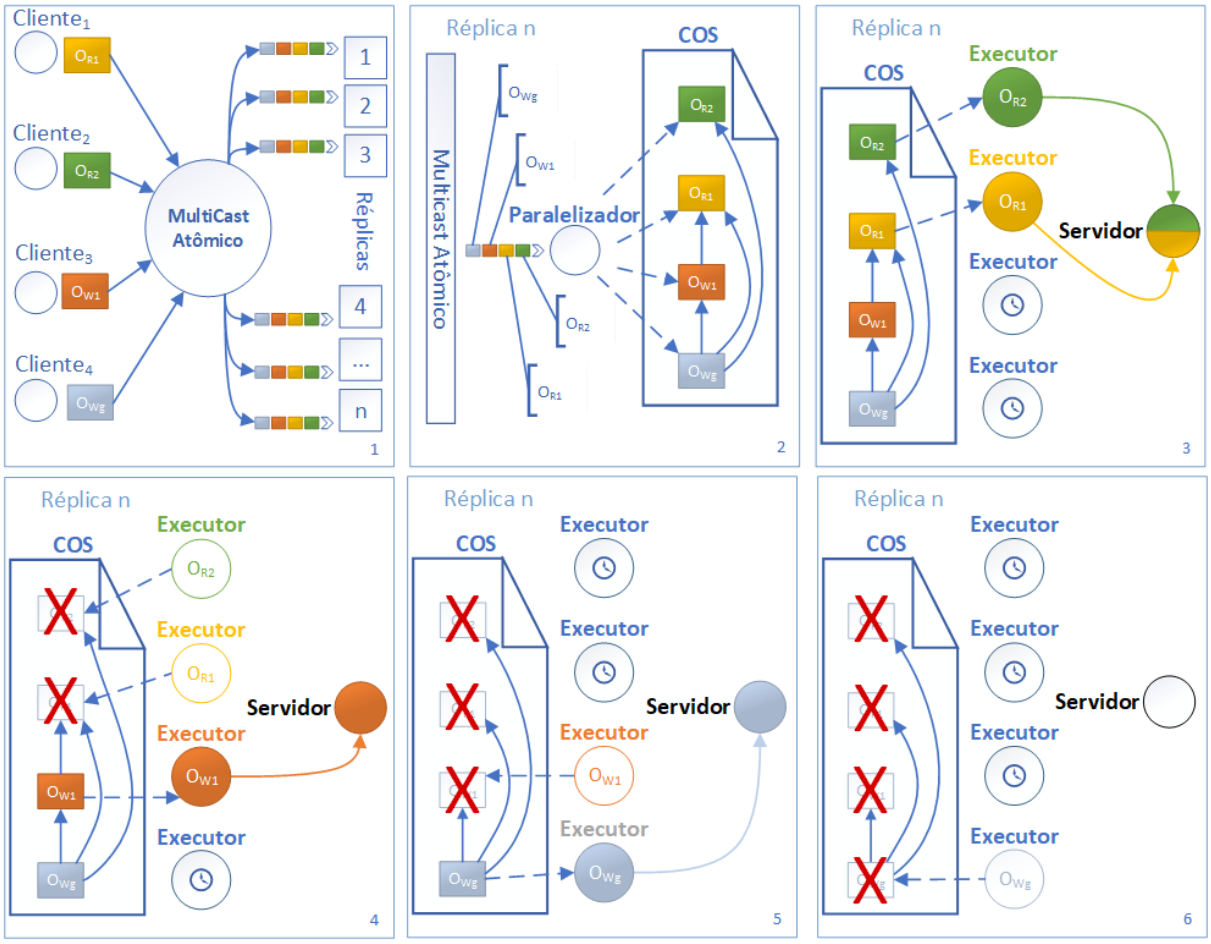


Figura 3.3: Exemplo de funcionamento do escalonamento tardio.

ordem de chegada dos comandos até encontrar o comando O_{Wg}, para então captura-lo, executa-lo e depois marca-lo como logicamente removido, verificando em seguida se essa ação liberou outro comando do grafo para execução (quadros 5 e 6 da Figura 3.3).

No exemplo exposto, os quatro comandos permaneceram no COS, mesmo estando logicamente removidos. A remoção física dos comandos do COS é realizada pelo paralelizador no ato da inserção de novos comandos. Ou seja, no ato da inclusão de um novo comando após os quadros apresentados, o paralelizador removerá do COS todos os comandos logicamente excluídos.

3.2 Escalonamento antecipado

O escalonamento antecipado [15, 13] é aplicado em conjunto com as técnicas de replicação ativa [1, 42] e inovou ao reduzir o custo do escalonamento de novas requisições, quando comparado com o escalonamento tardio (Seção 3.1). Intuitivamente, o escalonamento de

uma nova requisição é mais rápido pois não exige a varredura de todas as requisições pendentes num grafo de dependências, como no escalonamento tardio. Para conseguir isso, o escalonamento antecipado introduz a noção de classes de requisição e adota um modelo de execução baseado nessas classes para sincronizar requisições de forma eficiente.

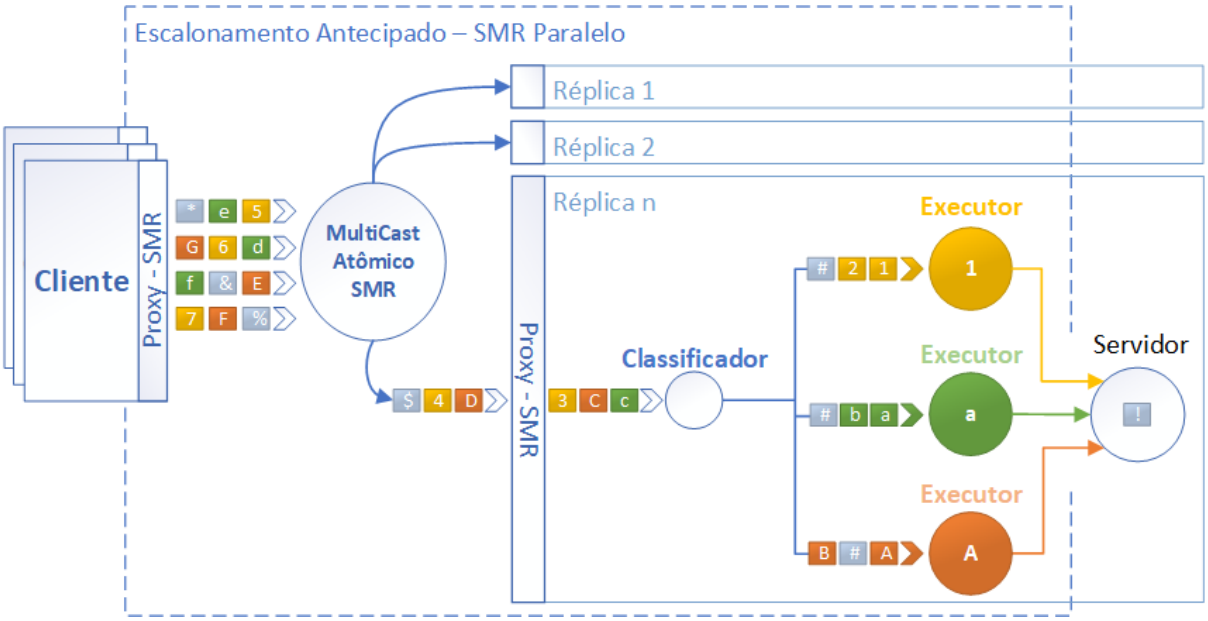


Figura 3.4: Escalonamento Antecipado.

Assim como no escalonamento tardio, a estrutura de escalonamento em cada réplica é composta por uma *thread* especial, aqui chamada de classificador, e $\boldsymbol{N}$ *threads* de execução, como pode ser visto na Figura 3.4. Basta ao classificador da réplica saber a classe da requisição para atribuí-la à *thread* de execução apropriada, ou, às *threads* de execução apropriadas. Quando uma requisição é atribuída a mais de uma *thread* executora, se faz necessária uma etapa adicional de sincronização entre essas *threads* para manter a consistência da Replicação Máquina de Estados, enquanto se desfruta da execução paralela de comandos num hardware *multi-core*. Uma vantagem dessa abordagem é que o custo de escalonamento de uma nova requisição permanece constante, independentemente da população de requisições pendentes.

As classes de requisições são sempre disjuntas e não vazias, ou seja, cada requisição só pode pertencer a uma única classe e classes sem requisições não são consideradas. Uma classe C_n pode ser livre de conflitos, ou possuir conflitos que podem ser internos ou externos. Segundo as definições abaixo:

- **Livre de conflito:** $\{\forall\ r_1, r_2 \in R \ \mid \ r_1 \in C_n \wedge \ r_2 \in C_n \Longrightarrow \{r_1, r_2\} \notin \#_C\}$
- **Conflito interno:** $\{\exists\ r_1, r_2 \in R \ \mid \ r_1 \in C_n \wedge \ r_2 \in C_n \Longrightarrow \{r_1, r_2\} \in \#_C\}$

- **Conflito externo:** $\{\exists\ r_1, r_2 \in R\ \mid\ r_1 \in C_n \wedge\ r_2 \notin C_n \Longrightarrow \{r_1, r_2\} \in \#_C\}$

Nas réplicas, cada classe é mapeada para uma, ou mais, filas de execução [15]. Cada fila de execução possui uma *thread* executora exclusiva. Ou seja, cada executor atende a uma única fila de execução. O mapeamento de classes de requisições em *threads* de execução foi modelado como um problema de otimização com o objetivo de maximizar o número de requisições em execução paralela em um dado momento e deve obedecer as seguintes regras [15]:

R1. *Toda classe está associada a pelo menos uma thread executora.* Este é um requisito necessário para garantir que as requisições em todas as classes sejam eventualmente executadas.

R2. *Se uma classe tem conflitos internos, ela deve ser sequencial.* Nesse caso, todas as executoras associadas à classe devem ser sincronizadas para que cada requisição seja executada na mesma ordem em todas as réplicas. Associar mais de uma *thread* executora a uma classe sequencial não ajuda no desempenho, mas permite a sincronização com outras classes.

R3. *Se duas classes conflitam entre si, pelo menos uma delas deve ser sequencial.* A regra anterior pode ajudar a decidir qual classe, entre duas classes, deve ser sequencial.

R4. *Se a classe concorrente c_{con} conflita com a classe sequencial c_{seq}, então o conjunto de threads executoras associado a c_{con} deve ser incluído no conjunto associado a c_{seq}.* Essa regra garante que as requisições em c_{seq} sejam serializadas com as requisições em c_{con}.

R5. *Se duas classes sequenciais c_{seq1} e c_{seq2} forem conflitantes entre si, devem ter ao menos uma thread executora em comum.* Essa *thread* comum garante que as requisições nas classes sejam serializadas.

3.2.1 Protocolo

Nesta sessão apresentamos detalhadamente a implementação do escalonamento antecipado conforme Alchieri et al. [15], pois seu conhecimento servirá como base para o entendimento do escalonamento híbrido. Antes mesmo de enviar suas requisições para as réplicas, os clientes as rotulam com a classe correspondente e as enviam para a camada de comunicação e ordenação. Essa camada entrega as requisições dos clientes na mesma ordem total para todas as réplicas.

Classificador. Cada réplica possui uma *thread* para classificar e N *threads* para executar as requisições, cada executor possui uma fila (FIFO) de entrada independente e exclusiva. O Algoritmo 4 apresenta os tipos de dados do escalonamento antecipado, assim como o algoritmo do classificador e as variáveis acessadas. O classificador ao receber uma requisição r, por meio da chamada ***onDeliver*** (linha 15), agenda a execução de r para uma ou mais executoras de acordo com as seguintes regras: (a) se a classe de r for sequencial, então todas os executores mapeados para a classe recebem r para sincronizar sua execução (linhas 16-18); (b) se a classe de r for concorrente, o classificador atribui r a um único executor dentre os mapeados para a classe de r (linha 19–20), seguindo uma política round-robin (função *next* na linha 20).

As funções $threads(classId)$ e $smode(classId)$ (linhas 10-11) realizam um mapeamento direto do identificador de uma classe para os índices das filas de suas *threads* executoras e para seu modo de sincronização, respectivamente.

Algoritmo 4 Escalonamento antecipado: classificador.

```
1:  Tipos de dados:
2:    T : set of thread identifiers
3:    C : set of class identifiers
4:    Req : {c : Command,                          {uma requisição tem o comando a ser executado}
5:           classId : C}                          {e o identificador da classe a que ele pertence}
6:    CtoT =                                       {o mapeamento de classes para threads é...}
7:      ∀ classId ∈ C →                            {para cada classe}
8:        smode : {Seq, Conc},                     {...um modo de sincronização...}
9:        threads : P(T)                           {...e um subconjunto não vazio de threads}
10:   smode(classId) = CtoT(classId).smode
11:   threads(classId) = CtoT(classId).threads
12: Variáveis:
13:   queues[0, ..., n − 1] ← ∅                    {uma fila por thread}
14: Classificador executa assim:
15:   onDeliver(req):
16:     if smode(req.classId) = Seq then           {se a execução é sequencial}
17:       ∀t ∈ threads(req.classId)                {para cada thread conflitante}
18:         queues[t].fifoPut(req)                 {sincronize para executar a requisição}
19:     else                                       {caso contrário}
20:       queues[next(threads(req.classId))].fifoPut(req)   {atribua req a uma thread em
          round-robin}
```

Executor. No Algoritmo 5 podemos ver que cada executor retira uma requisição por vez de sua fila de entrada segundo a ordem FIFO (linha 6) e decide como prosseguir com

base no modo de sincronização da classe da requisição obtida: (a) se sequencial, o executor sincroniza com os outros executores da mesma classe usando a execução com barreiras (linha 8); (b) se concorrente, o executor simplesmente executa a requisição (linha 10).

Algoritmo 5 Escalonamento antecipado: executores.

```
1:  Variáveis:
2:    queue[T]                                   {uma fila vazia de requisições por thread em T}
3:    barrier[C]                                 {uma barreira inicializada por classe de requisição}
4:  Cada thread executora (t_id ∈ T) executa assim:
5:    while true do                              {laço infinito}
6:      req ← queue[t_id].fifoGet()              {aguarde a próxima requisição disponível}
7:      if smode(req.classId) = Seq then         {se a execução for sequencial}
8:        execWithBarrier(req, barrier[req.classId])   {execute usando barreiras}
9:      else                                     {caso contrário}
10:       exec(req)                              {execute livremente}
11: execWithBarrier(req, barrier):
12:   if t_id = min(threads(req.classId)) then   {se for a thread de menor id}
13:     barrier.await()                          {aguarde o sinal}
14:     exec(req)                                {execute a requisição}
15:     barrier.await()                          {libere as outras threads executoras}
16:   else                                       {caso contrário}
17:     barrier.await()                          {sinalize para a thread executora}
18:     barrier.await()                          {aguarde execução}
```

Vale lembrar que, quando uma requisição r pertence a uma classe sequencial c_{seq}, pelo funcionamento do classificador (Algoritmo 4, linhas 16-18), r será incluída na fila de todos os executores vinculados a c_{seq}. A execução com barreiras (Algoritmo 5, linhas 11–18), utilizada para as requisições de classes sequenciais, garante que apenas uma executora execute a requisição, enquanto todas as outras da mesma classe aguardam.

3.2.2 Discussão de Corretude

Argumentamos por análise de caso que qualquer mapeamento que segue as regras R1 a R5 gera execuções linearizáveis. Consideramos os quatro possíveis cenários abaixo:

(1) *A classe c não tem conflitos internos ou externos.* Então, qualquer requisição $r \in c$ é independente de quaisquer outras requisições e pode ser despachada para a fila de entrada de qualquer *thread* atribuída a c. De acordo com R1, essa *thread* existe. A *thread* retira a requisição da fila e a executa sem sincronização adicional (Algoritmo 5, linha 10).

(2) *A classe c tem conflitos internos, mas nenhum conflito externo.* Então, pela regra R2, c é sequencial e qualquer requisição $r \in c$ é enfileirada nas filas de entrada de todas as *threads* associadas à classe c, de acordo com a ordem de entrega. Essas *threads* eventualmente desenfileiram r devido à mesma ordem em suas filas e sincronizam para executá-lo (Algoritmo 5, linha 8), ou seja, as requisições pertencentes a classe c são executadas respeitando a sua ordem de entrega.

(3) *A classe c_1 não tem conflitos internos, mas está em conflito com c_2.* Pela regra R3, uma das classes deve ser sequencial. Sem perda de generalidade, suponha que c_2 seja sequencial e c_1 seja concorrente. De R4 temos que as *threads* que implementam c_1 estão contidas no conjunto de *threads* que implementam c_2. Segue-se então que cada requisição de c_1 é executada antes ou depois de qualquer requisição de c_2, de acordo com sua ordem de entrega. Observe que isso é válido mesmo que as requisições em c_1 sejam simultâneas.

(4) *Classes c_1 e c_2 têm conflitos internos e conflitam entre si.* Então c_1 e c_2 são sequenciais. Ambos sincronizam suas *threads* para executar requisições. De acordo com a restrição R5, essas classes têm pelo menos uma *thread* comum t_x que é suficiente para impor que c_1 e c_2 executem suas requisições de acordo com a ordem de entrega.

Para garantir o progresso, é essencial garantir que as dependências entre as requisições sejam acíclicas. Dado que (i) a ordem de entrega não tem ciclos de dependência (isto é, uma requisição r só pode depender de requisições previamente entregues); (ii) A fila de requisições de cada *thread* executora preserva esta ordem; e (iii) Cada executora processa suas requisições sequencialmente. Temos que todas as dependências entre requisições são acíclicas em todas as filas de execução. Com isso, é sempre possível encontrar uma requisição que não depende de nenhuma outra a ser executada.

3.2.3 Exemplo de Execução

Para facilitar o entendimento, assim como fizemos com o escalonamento tardio, vamos apresentar um exemplo de execução descrevendo a jornada de 4 comandos de uma aplicação no escalonamento antecipado. A aplicação é o mesmo sistema imaginário bi-particionado com operações de leitura e escrita locais (em partições individuais) e de escritas globais (em todas as partições) que foi utilizado no exemplo de execução da Seção 3.1.5. Novamente, no nosso sistema hipotético as escritas locais conflitam com outras escritas e leituras locais, ou seja, da mesma partição. As leituras locais não conflitam entre si e as escritas globais conflitam com todas as operações.

Para um sistema assim, teremos um esquema de classes como o da Figura 3.5. As classes C_{R1} e C_{R2} são classes de requisições concorrentes que conflitam externamente com as classes sequenciais C_{W1} e C_{W2}, respectivamente, e todas as classes conflitam externamente com a classe sequencial C_{Wg}.

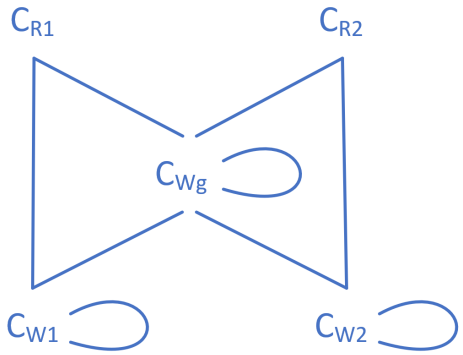


Figura 3.5: Classes de requisições e modelo de concorrência: leitores locais (C_{R1} e C_{R2}), escritores locais (C_{W1} e C_{W2}) e escritores globais (C_{Wg}). As arestas no diagrama representam conflitos externos entre as classes e laços representam conflitos internos.

No nosso exemplo consideramos que existem 4 *threads* de execução ($Executor_1$, $Executor_2$, $Executor_3$ e $Executor_4$) e que o mapeamento de classes para *threads* (CtoT) fica assim:

- $C_{R1} \rightarrow Executor_1 \wedge Executor_2$;
- $C_{W1} \rightarrow Executor_1 \vee Executor_2$;
- $C_{R2} \rightarrow Executor_3 \wedge Executor_4$;
- $C_{W2} \rightarrow Executor_3 \wedge Executor_4$;
- $C_{Wg} \rightarrow Executor_1 \wedge Executor_2 \wedge Executor_3 \wedge Executor_4$.

Vamos acompanhar 4 operações específicas $O_{R1} \in C_{R1}$, $O_{R2} \in C_{R2}$, $O_{W1} \in C_{W1}$ e $O_{Wg} \in C_{Wg}$ que representam respectivamente, uma leitura na partição 1, uma leitura na partição 2, uma escrita na partição 1 e uma escrita global (as mesmas da Seção 3.1.5).

O quadro 1 da Figura 3.6 apresenta as quatro operações sendo enviadas por quatro processos clientes distintos. A difusão é realizada por uma biblioteca de *multicast* atômico. A primitiva de comunicação é síncrona e bloqueante, sendo assim, a *thread* do cliente que realiza a transmissão fica bloqueada aguardando a resposta do serviço remoto. A biblioteca de *multicast* atômico ordenará as requisições recebidas e as entregará em todas as réplicas do serviço na mesma ordem total.

Perceba que, diferentemente do escalonamento tardio, antes de entregar os comandos à camada de transmissão e ordenação, ainda no quadro 1, cada cliente rotula seu comando

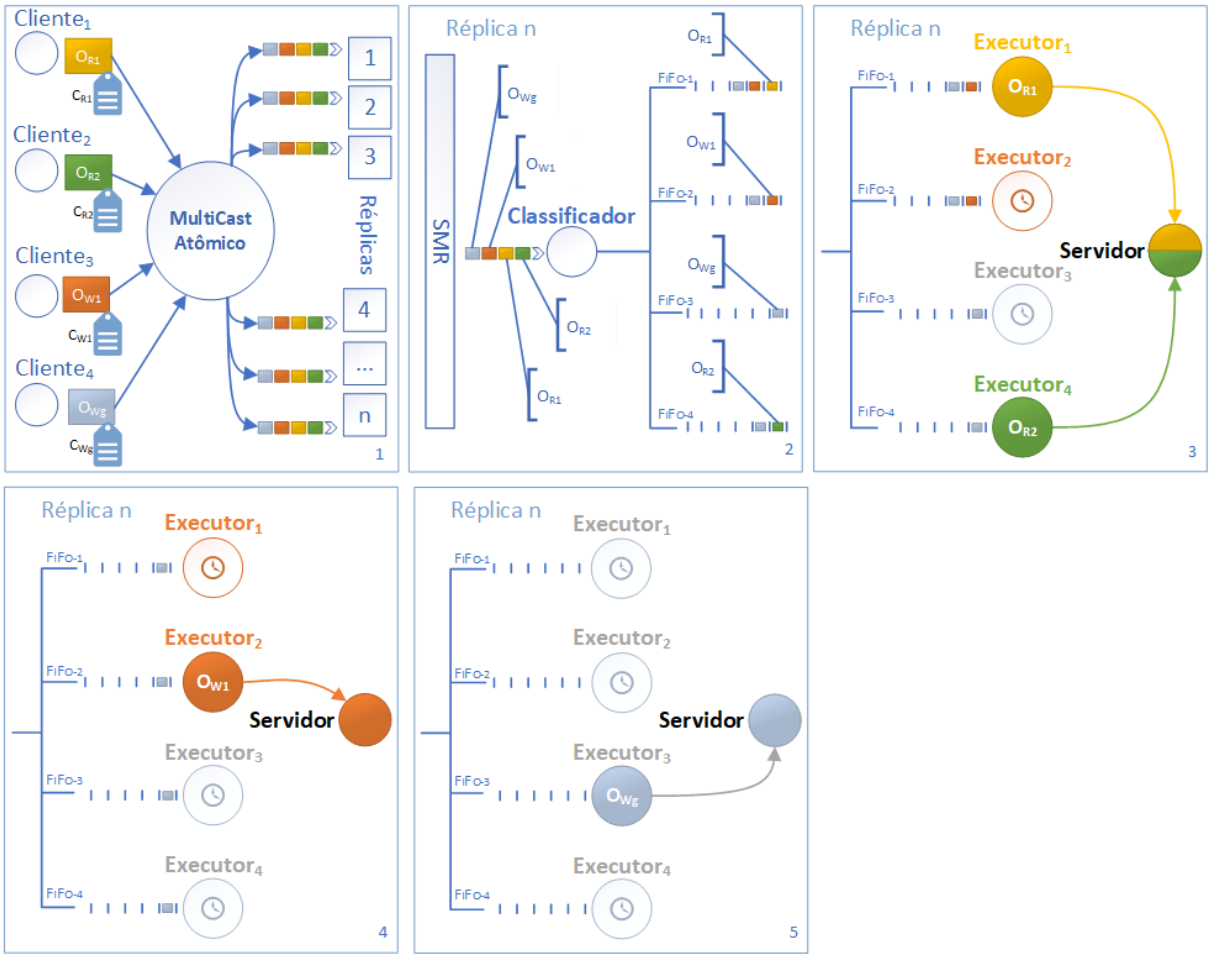


Figura 3.6: Exemplo de funcionamento do escalonamento antecipado.

com o identificador da classe de requisições adequada. Esse rótulo é o que determina quais filas de processamento a *thread* classificadora deve considerar ao tratar a requisição. Os quadros de 2 a 5 da Figura 3.6 representam as estruturas internas de uma réplica. No quadro 2 podemos ver que o classificador escolheu a fila *FiFo-1* para o comando de leitura O_{R1}[3] e a fila *FiFo-4* para o outro comando de leitura, o O_{R2}[4]. Já o comando O_{W1}, por pertencer à classe C_{W1}, foi incluído ao mesmo tempo nas filas *FiFo-1* e *FiFo-2*. Assim como o comando O_{Wg} que foi incluído em todas as filas.

O quadro 3 apresenta as quatro *threads* executoras assim que detectam o primeiro comando de suas listas de entrada. O $Executor_1$ ao ler o comando O_{R1} da classe concorrente C_{R1}, iniciou imediatamente sou processamento. O $Executor_4$ procedeu exatamente da mesma forma, levando ao processamento paralelo dos comandos O_{R1} e O_{R2}. Os executores 2 e 3 foram bloqueados pelas barreiras iniciais de sincronização das classes sequenciais

[3]Note que também poderia ser escolhida a fila *FiFo-2*.
[4]Note que também poderia ser escolhida a fila *FiFo-3*.

C_{W1} e C_{Wg}, respectivamente.

Assim que os executores 1 e 4 terminam o processamento de seus comandos, eles passam a tratar o próximo comando de suas filas, na ordem FIFO, como está representado no quadro 4 da Figura 3.6. Neste momento o $Executor_4$ fica bloqueado na barreira inicial de sincronização da classe C_{Wg}. O $Executor_1$ encontra a barreira inicial de sincronização da classe C_{W1}, mas esse encontro dispara a barreira liberando a execução do comando O_{W1}. O $Executor_2$ então inicia o processamento de O_{W1} enquanto o $Executor_1$ encontra a segunda barreira de sincronização da classe C_{W1}, onde ficará bloqueado até que o processamento de O_{W1} encerre. Ao encerrar seu processamento, o $Executor_2$ encontra a segunda barreira de sincronização de C_{W1}, o que libera a si próprio e ao $Executor_1$ para tratar o próximo comando.

Ao ler o comando O_{Wg} de suas filas, os executores 1 e 2 alcançam a primeira barreira de sincronização da classe C_{Wg}. Nesse momento a primeira barreira libera as 4 *threads*, o $Executor_3$ executa a operação O_{Wg} e os outros 3 executores ficam bloqueados na segunda barreira de sincronização de C_{Wg}. O quadro 5 representa exatamente esse momento. Encerrado o processamento de O_{Wg}, o $Executor_3$ alcançará a segunda barreira de sincronização, liberando todos os executores para continuar trabalhando em suas filas.

3.2.4 Escalonamento estático

O escalonamento estático [11] é uma variante do escalonamento antecipado [44]. Essa variante não possui uma *thread* classificadora nas réplicas. Na abordagem estática, a definição de qual *thread* executora (da réplica) tratará o comando é realizada exclusivamente no processo cliente. Os comandos são então enviados às réplicas por diversos grupos de *multicast* atômicos simultâneos. Cada grupo *multicast* alimenta uma *thread* executora de comandos nas réplicas. Ou seja, os clientes separam suas requisições em classes e depois as enviam pelo(s) grupo(s) de *multicast* atômico adequados para aquela classe. As réplicas atendem cada ordem parcial de comandos entregue por um grupo de *multicast* atômico de forma sequencial por uma *thread* exclusiva. Quando a requisição pertence a uma classe com conflitos externos (Seção 3.2), ela é transmitida pelos grupos de *multicast* envolvidos. Ao receber a requisição conflitante, as *threads* desses grupos se coordenam por meio de barreiras, tal que só uma delas execute o comando enquanto as outras aguardam. Dessa forma a relação de dependência entre os comandos é preservada. Como pode ser visto no esquema da Figura 3.7, nenhuma decisão sobre a escolha da *thread* de execução é tomada por parte do servidor, pois cada grupo de *multicast* tem sua *thread* executora exclusiva.

Por desconhecer a distribuição de carga entre as *threads* do servidor durante a execução das requisições, essa abordagem não consegue realizar um balanceamento de carga efetivo, correndo o risco de possuir *threads* ociosas e requisições enfileiradas simultaneamente.

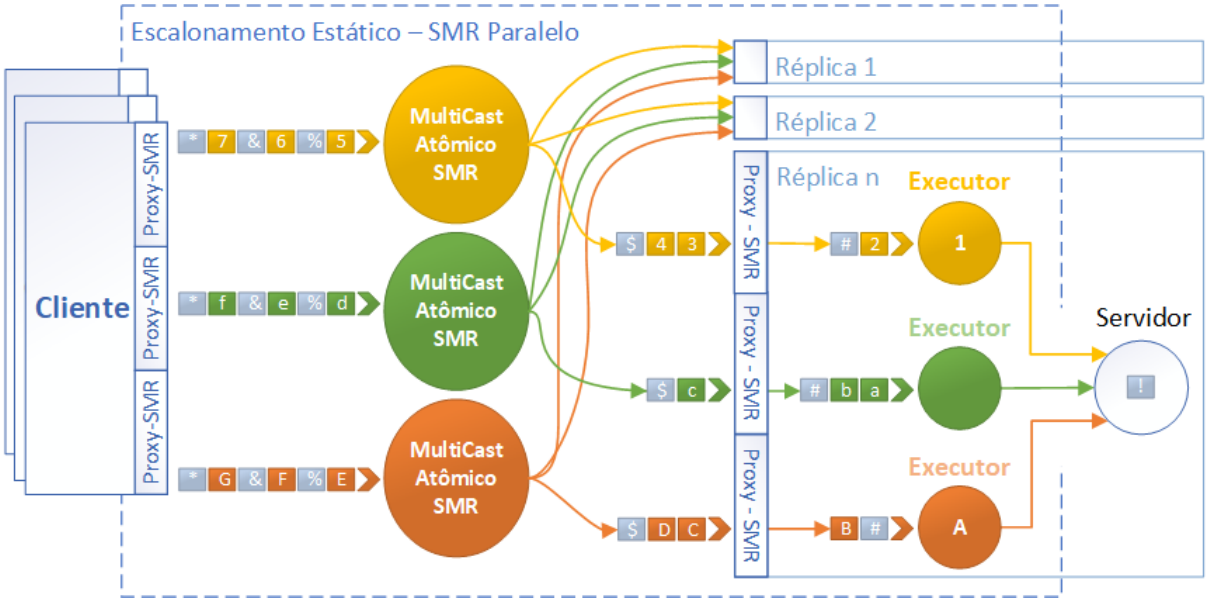


Figura 3.7: Escalonamento Estático.

Além disso, a coordenação entre as *threads* executoras, quando existem dependências entre ordens parciais diversas, pode levar a uma redução significativa de desempenho. Nestes casos, quando a *thread* encontra uma requisição dessas, ela interrompe o processamento de sua fila de requisições recebidas via *multicast* até que as outras *threads* envolvidas no conflito alcancem a mesma requisição em suas filas, para que então uma delas venha a executar a requisição e desbloquear todas as outras.

3.3 Outras soluções

Esta seção apresenta outras soluções que permitem execuções paralelas em RME, as quais seguem abordagens diferentes das soluções discutidas até aqui.

3.3.1 Algoritmos independentes da semântica da aplicação

Existem técnicas que não utilizam informações relativas à aplicação para paralelizar sua execução, ditos algoritmos independentes da semântica da aplicação. O LSA [45], o CRANE [9] e o Rex [8], descritos a seguir, são exemplos dessas técnicas:

LSA

O *Loose Synchronization Algorithm* (LSA) [45] utiliza uma abordagem líder-seguidor para RMEP. No LSA a comunicação da aplicação com o sistema operacional para o bloqueio e

desbloqueio de seções críticas do código é substituída por uma implementação específica em tempo de compilação. Uma réplica é escolhida como líder, todos os clientes se comunicam com essa réplica e ela se comunica e coordena as outras réplicas, só depois de receber as respostas de suas seguidoras o líder responde aos clientes com o consenso da maioria. A réplica líder recebe as requisições e repassa para suas seguidoras, em seguida executa suas *threads* sem qualquer restrição, porém segue transmitindo a ordem das atribuições de bloqueio e desbloqueios das regiões críticas para suas seguidoras. Quando as réplicas seguidoras recebem as requisições do líder também começam a processar com suas *threads*. Porém, ao encontrar uma operação de bloqueio de sessão crítica a *thread* da réplica seguidora só prosseguirá se, e quando, for a sua vez de adquirir esse bloqueio pela ordem estabelecida e compartilhada pelo líder.

Percebemos como deficiências nessa abordagem as múltiplas etapas de comunicação e consenso entre o líder e os seguidores que aumentam o tempo de invocação que o cliente observa [46], além do alto tempo de recuperação para falhas da réplica líder quando comparado a uma RMEP com replicação ativa.

CRANE

O sistema CRANE [9] mantém as réplicas de um serviço sincronizadas usando diversos mecanismos em tempo de execução. A interface de comunicação do servidor principal é alterada com a introdução de um módulo de acesso que só repassa a requisição para o próprio servidor, depois de transmiti-la com sucesso a todas as réplicas usando uma implementação do Paxos [39, 40]. A sincronização das *threads* do servidor é feita pelo uso de um escalonador determinístico DMT [47] a nível de sistema operacional. Adicionalmente, o CRANE ainda introduz um intervalo de tempo artificial entre as requisições para garantir a execução de lotes de comandos em tempos lógicos determinísticos.

Percebemos como pontos fracos dessa abordagem, a sobrecarga sobre o servidor principal, a sobrecarga adicionada pelo escalonador DMT (em média de 12,7% do desempenho do sistema), o acoplamento com o sistema operacional da máquina e a complexidade envolvida com essa modificação.

REX

O sistema Rex [8] utiliza uma estratégia do tipo execute-concorde-siga. Nele, um servidor denominado primário recebe as requisições do cliente e as processa em paralelo com diversas *threads*. Enquanto executa, o servidor primário registra as requisições e suas relações de dependência com base nas variáveis compartilhadas acessadas (bloqueadas e desbloqueadas) por cada *thread*. Periodicamente o servidor primário envia um lote desses registros coletados para consenso com o conjunto de réplicas. As réplicas recebem

os registros e reproduzem a execução do servidor primário, respeitando a ordem parcial dos comandos, seguindo a causalidade nas operações de bloqueio e desbloqueio. O não determinismo causado pela concorrência é resolvido pela repetição das decisões tomadas pelo servidor primário.

Enquanto que nas técnicas do tipo concorde-execute de RMEP, vistas anteriormente, as instâncias de consenso são independentes entre si, no Rex elas não são, pois os lotes de registros de execução são encadeados e cada um depende da execução de seu antecessor para que possa executar. A presença de um servidor principal com muito mais responsabilidades que os outros também pode levar a situação de sobrecarga nesse servidor, além do tempo adicional de resposta a falhas no servidor primário, típico dos sistemas de replicação passiva (Seção 2.5). Afora essas limitações, a própria sincronização dos registros de execução pode resultar em alto consumo de largura de banda da rede e sobrecarga de desempenho.

3.3.2 Algoritmos otimistas

Este tipo de solução é caracterizado pela existência de duas etapas distintas, uma etapa de paralelização e uma etapa de validação da consistência. O Eve [6], o Storyboard [48] e o opt-SMR [12], descritos a seguir, são exemplos destas soluções.

Eve: Execute-Verifique

No sistema Eve [6], a réplica principal recebe lotes de requisições dos clientes e repassa para execução nas outras réplicas. Em cada réplica um módulo, chamado misturador, particiona deterministicamente o lote original em micro-lotes de requisições paralelas. Esses micro-lotes são executados em sequência, porém operações do mesmo micro-lote são executadas em paralelo. No final do processamento as réplicas calculam um *token* com base no estado corrente e nas respostas das requisições, em seguida enviam esse *token* para o estágio de verificação. Na verificação as réplicas entram em consenso sobre o *token* calculado, a fim de verificar se houve alguma violação de consistência. No caso de haver uma violação de consistência, as réplicas desfazem o processamento do lote e o reexecutam de forma determinística e sequencial.

O misturador no Eve pode restringir o desempenho da execução, pois o conteúdo de todas as requisições deve ser examinado por ele antes de poder ser executado. Além disso, a réplica primária pode ficar sobrecarregada com o trabalho extra. O estágio de verificação é outro ponto de sincronização que, além do misturador e da réplica primária, pode ameaçar a escalabilidade dessa abordagem. Embora as inconsistências sejam

raras, o impacto no desempenho da estratégia adotada para restabelecer a consistência é significativo.

Em 2019, Aksoy e Kapritsos [49] revisitaram o Eve, estendendo-o com quatro técnicas que permitiram seu uso num ambiente onde há interação entre serviços. A plataforma desenvolvida se chama **Aegen** e os resultados apresentados no artigo mostram uma melhora no desempenho em relação ao **Eve** quando aplicado no contexto de micro-serviços, por exemplo.

Storyboard

O Storyboard [48] funciona no modelo concorde-execute. Ao receber um lote de requisições, a réplica executa um componente de predição que possui conhecimentos sobre o funcionamento da aplicação. Esse componente inspeciona as requisições e prevê a ordem dos bloqueios das variáveis compartilhadas durante a execução do lote. O preditor funciona de tal forma que não há divergências nas previsões de diferentes réplicas para o mesmo lote de requisições. A previsão sobre os bloqueios é enviada juntamente com os comandos para o componente de execução, chamado Storyboard. Enquanto as previsões estiverem corretas, as requisições podem ser executadas em paralelo, respeitando a ordem dos bloqueios antevista. Se o Storyboard detectar que o caminho de execução de uma requisição não corresponde à previsão feita pelo preditor, a réplica interromperá o processamento da requisição e usará um acordo (consenso) para recalcular o caminho de execução do comando, em cooperação com as outras réplicas.

OPT-SMR

Uma técnica otimista para aumentar a concorrência de execução do escalonamento estático também foi proposta [12]. Nesse trabalho foram apresentados os conceitos de interdependência estática e dinâmica entre comandos. As interdependências estáticas são aquelas que podem ser determinadas sem conhecimento sobre o estado do serviço, e as dinâmicas, aquelas que não podem ser determinadas sem o conhecimento sobre o estado do serviço. Como no escalonamento estático, o cliente não tem conhecimento sobre os valores das variáveis de estado internas das réplicas do serviço. Enquanto que no escalonamento estático os comandos dinamicamente interdependentes são mapeados como sequenciais, em opt-SMR eles são marcados otimisticamente como paralelos, ou seja, independentes entre si. Para garantir a linearizabilidade do sistema, no momento da execução, na réplica, a abordagem otimista realiza uma verificação de segurança em todos os comandos. Se for detectado que a execução de um comando causará conflito com a execução de outros comandos em paralelo, considera-se que a hipótese otimista falhou e o comando conflituoso será reordenado entre as réplicas utilizando uma abordagem mais conservadora. O

opt-SMR apresenta as mesmas limitações do escalonamento estático (Seção 3.2.4), com o adendo de que a melhora ou piora no desempenho depende em grande parte de quão aderente está o conjunto de requisições à sua decisão especulativa.

3.4 Considerações Finais

Neste capítulo vimos os trabalhos relacionados e o estado da arte em técnicas de Replicação Máquina de Estados Paralela (RMEP). A Tabela 3.1 apresenta um resumo das abordagens discutidas neste capítulo.

<table>
<tr><th>Algoritmo</th><th>Abordagem</th><th>Independente da semântica da aplicação</th></tr>
<tr><td>LSA</td><td>Líder-Seguidores</td><td rowspan="3">Sim</td></tr>
<tr><td>CRANE</td><td>DMT</td></tr>
<tr><td>REX</td><td>Execute-Concorde-Siga</td></tr>
<tr><td>EVE</td><td>Execute-Verifique</td><td rowspan="6">Não</td></tr>
<tr><td>StoryBoard</td><td rowspan="5">Concorde-Execute</td></tr>
<tr><td>OPT-SMR</td></tr>
<tr><td>Escalonamento Tardio</td></tr>
<tr><td>Escalonamento Antecipado</td></tr>
<tr><td>Escalonamento Estático</td></tr>
</table>

Tabela 3.1: Resumo dos trabalhos relacionados.

Dentre as técnicas apresentadas, as abordagens de escalonamento tardio e antecipado tiveram seus algoritmos apresentados detalhadamente e um exemplo de execução foi apresentado para cada uma delas. Também foi discutida a corretude dessas abordagens. No Capítulo 4, será apresentado o modelo de escalonamento híbrido para RMEP proposto neste trabalho.

Capítulo 4

Escalonamento Híbrido

Este capítulo apresenta o protocolo de escalonamento híbrido elaborado neste trabalho. São expostos o seu modelo de sistema, algoritmos e detalhes de implementação, juntamente com uma discussão sobre a sua corretude e um exemplo de execução para fixar o entendimento.

4.1 Introdução e Motivação

Replicação Máquina de Estados (RME) é uma maneira conceitualmente simples, mas eficaz, de projetar serviços que devem resistir a falhas [1, 2]. A RME fornece aos clientes a abstração de um serviço altamente disponível enquanto oculta a existência de múltiplas réplicas trabalhando em conjunto, no que é conhecido como consistência forte ou linearizabilidade [3, 50]. Na RME clássica, a linearizabilidade (Seção 2.6) é obtida fazendo com que as réplicas executem os comandos na mesma ordem, de forma sequencial e determinística[1]. Para tanto, normalmente usa-se o *multicast* atômico (Seção 2.3), um protocolo de comunicação em grupo que ordena os comandos enviados pelos clientes, garantindo a entrega dos mesmos comandos em todas as réplicas na mesma ordem total. Como as réplicas partem do mesmo estado inicial e executam os mesmos comandos na mesma sequência, então elas produzirão as mesmas mudanças de estado e respostas.

Apesar de seu uso difundido (por exemplo, [4, 5]), a RME clássica faz mau uso dos recursos disponíveis em computadores com vários núcleos de processamento (*multi-core*), devido à execução sequencial de comandos. A fim de abordar esta limitação de desempenho importante, várias técnicas foram propostas para acomodar a execução paralela de requisições em uma RME (por exemplo, [10, 14, 15, 11, 6, 12, 51]). A maioria dessas soluções é baseada em uma observação inicial sobre RME: embora a execução paralela de

[1]Conforme já explicado, comandos determinísticos são aqueles cujos resultados só dependem deles mesmos e do estado da aplicação,

comandos possa resultar em não determinismo, comandos independentes podem ser executados simultaneamente sem violações de consistência [1]. Lembre-se que dois comandos são *independentes* (ou *não conflitantes*) se eles não compartilham variáveis do estado do sistema ou apenas leem as variáveis compartilhadas; inversamente, dois comandos são *dependentes* (ou *conflitantes*) se eles acessam ao menos uma variável em comum e pelo menos um dos comandos muda o valor da variável compartilhada.

Nesse trabalho duas soluções para Replicação Máquina de Estados Paralela (RMEP) foram estudadas detalhadamente e aqui propomos uma nova abordagem que combina as vantagens de ambas. Embora o desempenho de uma RMEP seja limitado pela manutenção do determinismo e sensível à combinação entre comandos dependentes e independentes na carga de trabalho, diferentes abordagens têm sido sugeridas na literatura para executar comandos independentes simultaneamente, resultando em melhorias substanciais no desempenho. As duas técnicas selecionadas utilizam a replicação ativa e o conhecimento da semântica da aplicação para funcionar. Contudo, elas podem ser diferenciadas pela forma como as interdependências dos comandos são identificadas e como os comandos são escalonados para execução.

Como apresentado no Capítulo 3, no escalonamento tardio a carga de trabalho é naturalmente balanceada entre as *threads* executoras, mas o custo necessário para manter o grafo de dependências torna o paralelizador o gargalo do sistema [14]. De fato, apesar do escalonamento tardio com COS *LockFree* representar um avanço importante, a dependência de um paralelizador único continuou impondo um limite no desempenho da abordagem que é alcançado quando há um esvaziamento persistente do COS. Ou seja, quando o conjunto das *threads* executoras consegue consumir e executar os comandos tão rápido, ou mais, que o paralelizador consegue inseri-los no grafo. Dessa forma, a adição de poder de processamento ao conjunto dos executores não promoverá ganhos de desempenho visto que o paralelizador atingiu o seu limite. Por outro lado, no escalonamento antecipado o trabalho do classificador é simples e rápido, porém sincronizações adicionais são necessárias para execução de comandos conflitantes, o que diminui o seu desempenho abruptamente quando conflitos são adicionados nas cargas de trabalho [15]. Além disso, as sincronizações necessárias para tratar de comandos conflitantes inserem bolhas de inexecução de requisições, mesmo havendo requisições nas filas dos executores [16].

Estas características são mostradas em nossos experimentos (Capítulo 5). Na ausência de comandos conflitantes (ou seja, quaisquer dois comandos podem ser executados simultaneamente), o escalonamento antecipado supera amplamente o escalonamento tardio em diversas configurações. Isso ocorre porque o classificador pode atribuir eficientemente comandos às *threads*, de acordo com o mapeamento de classes, enquanto o paralelizador torna-se um gargalo no desempenho do escalonamento tardio. O escalonamento tardio,

no entanto, pode lidar com comandos conflitantes de forma mais eficiente do que o escalonamento antecipado, mesmo quando a porcentagem de comandos conflitantes na carga de trabalho é baixa (ou seja, 5%), uma vez que o custo de sincronizar os executores no escalonamento antecipado é alto.

Observando as limitações e as vantagens destas duas abordagens de escalonamento, esta trabalho propõe um modelo de escalonamento que faça uma boa gestão de conflitos e utilização dos recursos computacionais, como no escalonamento tardio, e cujo desempenho possa ser escalado com a adição de mais poder de processamento paralelo. Apresentamos então a técnica de **escalonamento híbrido** que se baseia nas vantagens do escalonamento tardio e antecipado. A ideia principal é particionar o estado do serviço, usar um classificador inspirado no escalonamento antecipado para encaminhar comandos para execução nas partições corretas, onde um paralelizador inspirado no escalonamento tardio e específico da partição inclui o comando em uma partição do COS, que é estendido para lidar com um sistema particionado. Finalmente, *Threads* de execução recuperam os comandos de sua partição do COS e os executam.

Um aspecto central na abordagem híbrida é como lidar com um comando c_{cs} multi-particionado sem comprometer a ausência de bloqueios nas operações do grafo (DAG) que implementa o COS. Nestes casos, cada paralelizador que trata o comando c_{cs} insere no grafo as arestas de dependências que ligam c_{cs} aos comandos pendentes em sua partição do DAG, independente de em qual partição c_{cs} será inserido. Observe que o escalonamento de c_{cs} pode inserir arestas que conectam duas partições. Uma avaliação de desempenho mostra que, em um serviço particionado, o escalonamento híbrido tem um desempenho semelhante ao escalonamento antecipado em uma carga de trabalho sem conflitos e supera as técnicas tardia e antecipada em até 3× em uma carga de trabalho com comandos conflitantes.

4.2 Modelo de Sistema

Assumimos um sistema distribuído (Seção 2.4) composto de processos interconectados, onde há um conjunto potencialmente ilimitado de processos clientes e um conjunto limitado de n processos servidores (réplicas). O sistema é assíncrono: não há limite para atrasos de mensagens e velocidades relativas de processo. Assumimos o modelo de falha por colapso, excluindo o comportamento malicioso e arbitrário. Um processo é *correto* se não falha, ou *defeituoso* caso contrário. Podem existir até f réplicas com defeito, de n réplicas, onde $n = 2f + 1$.

Os processos se comunicam por passagem de mensagem, usando comunicação um para um ou um para muitos. A comunicação um-para-um usa as primitivas $send(m)$ e

receive(m), onde m é uma mensagem. Se um remetente enviar uma mensagem vezes suficientes, um destinatário correto eventualmente receberá a mensagem. A comunicação um-para-muitos usa transmissão do tipo *multicast* atômico, definida pelas primitivas *broadcast*(m) e *deliver*(m), onde m é uma mensagem. A difusão atômica garante o seguinte propriedades [21, 52] [2]:

- *Validade*: se um processo correto transmite uma mensagem m, então m será eventualmente entregue.
- *Acordo uniforme*: se um processo entrega uma mensagem m, então todos os processos corretos eventualmente entregam m.
- *Integridade uniforme*: para qualquer mensagem m, todo processo entrega m no máximo uma vez, e, somente se, m foi transmitida anteriormente por um processo.
- *Ordem Total Uniforme*: Se dois processos corretos p e q entregam as mensagens m e m', então p entrega m antes de m', se, e somente se, q entrega m antes de m'.

A Replicação Máquina de Estados (RME) fornece *linearizabilidade* [3], uma forma de consistência forte (Seção 2.6). Os algoritmos de escalonamento discutidos exploram a concorrência entre comandos. Novamente, seja C o conjunto de comandos possíveis e $\#_C \subseteq C \times C$ a relação de conflito entre os comandos. Se $\{c_i, c_j\} \in \#_C$, então os comandos c_i e c_j conflitam e as réplicas devem serializar sua execução; caso contrário, as réplicas podem executar c_i e c_j simultaneamente.

4.3 H-SMR: escalonamento híbrido

Nesta seção, apresentamos o escalonamento híbrido que aproveita as vantagens das abordagens anteriores, ou seja, a simplicidade e rapidez do classificador do modelo antecipado, bem como a distribuição de carga e a sincronização dos executores através de uma implementação do COS baseada em grafo livre de bloqueios do modelo tardio.

4.3.1 Visão Geral

A ideia geral é permitir que vários paralelizadores insiram simultaneamente requisições no COS, removendo o gargalo do modelo tardio. Para isso, o estado da aplicação é particionado e o COS é formado por vários subgrafos (Figura 4.1), um para cada partição,

[2] A difusão atômica precisa de suposições de sincronia adicionais para ser implementada [53, 54]. Essas suposições não são usadas explicitamente pelo protocolo proposto neste trabalho.

sendo que requisições endereçadas a mais de uma partição criam um nó conectando os subgrafos correspondentes.

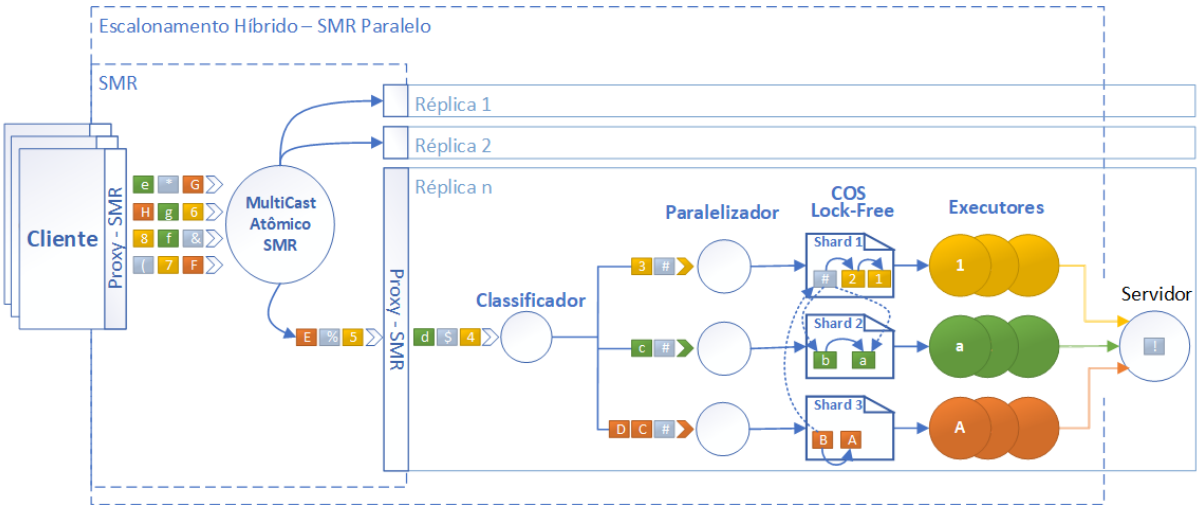


Figura 4.1: Escalonamento Híbrido: esquema de uma réplica.

Uma classe sequencial é associada a cada partição e mapeada para um paralelizador diferente. Uma vez que todas as inserções em uma mesma partição devem ser sequenciais, não faz sentido usar mais de um paralelizador por partição. Além disso, como uma requisição pode acessar quaisquer partições da aplicação, são criadas classes para todas as combinações possíveis de partições e os respectivos paralelizadores são atrelados a cada classe, permitindo a sincronização na inserção destas requisições. Deste modo, o cliente deve incluir o identificador da classe em cada requisição e o classificador recebe as requisições e as distribui entre os paralelizadores, de acordo com suas classes. Os paralelizadores inserem as requisições paralelamente no COS, i.e., nos respectivos subgrafos. Por fim, um conjunto de executores acessa o COS para execução das requisições.

O escalonamento híbrido adota o seguinte modelo de execução nas réplicas:

- $m + p + 1$ *threads*: um classificador, p paralelizadores e m executores.
- Cada paralelizador tem uma fila de escalonamento separada e remove requisições desta fila em ordem FIFO.
- Cada partição possui um conjunto de executores e um paralelizador associado, os quais acessam o subgrafo de dependências da partição para inserir (paralelizador) e obter/remover (executores) requisições.
- O classificador entrega as requisições na ordem total e encaminha cada requisição r para uma ou mais filas de escalonamento:

a. Caso r acesse apenas uma partição s, r depende das requisições anteriores endereçadas a s e é incluída na fila do paralelizador correspondente a s.

b. Caso r acesse mais de uma partição, r depende das requisições anteriores endereçadas a estas partições e é incluída nas filas correspondentes. Neste caso, todos os paralelizadores envolvidos em r devem incluir as dependências de r em suas respectivas partições, sendo que um deles também insere a requisição em seu subgrafo. Uma requisição só é considerada inserida no COS quando todos os paralelizadores envolvidos computarem suas dependências.

- Cada executor seleciona uma requisição que esteja pronta para execução no subgrafo correspondente, marcando-a como "em execução" e, após seu processamento, remove-a (logicamente) do subgrafo.

4.3.2 Protocolo

Agora apresentamos o protocolo de escalonamento híbrido com mais detalhes, enfatizando as principais diferenças e semelhanças em relação aos algoritmos anteriores. O algoritmo 6 apresenta os tipos de estruturas e variáveis usadas. O conjunto C de identificadores de classes de requisições tem a mesma definição do escalonamento antecipado (Algoritmo 4). Adicionamos um conjunto de identificadores de partições S e um mapeamento de classes para subconjuntos de partições (*shards*) $CtoS$, semelhante ao mapeamento de classes para *threads* do escalonamento antecipado. Cada partição tem uma fila de entrada e um COS associado, esse COS é uma versão do COS livre de bloqueios (Algoritmo 2) que foi estendida para lidar com comandos multi-particionados. Existem semáforos de contagem por partição para registrar o número de comandos prontos e o espaço livre para novos comandos. A definição de requisição foi reutilizada. O nó do COS estendido, agora chamado ***HyNode*** (*Hybrid Node*), é uma extensão do nó usado no COS *LockFree* adicionando novos atributos: $\boldsymbol{s_{id}}$ que armazena o identificador da partição que armazenará o nó; e $\boldsymbol{rem_s}$ que armazena o número de paralelizadores que ainda não processaram este nó durante a inserção. Cada *HyNode* n_{hy} também possui duas matrizes de conjuntos de nós, uma para nós que dependem do nó n_{hy} e outra para nós dos quais o nó n_{hy} depende. Cada paralelizador acessa apenas sua posição nas matrizes, evitando condições de corrida nesses conjuntos. A variável *shards* (linha 22) representa todas as partições em uma réplica.

Sempre que uma requisição é entregue, o classificador identifica as partições envolvidos na requisição (Algoritmo 7, linha 3) e escolhe uma delas para armazenar o nó (linha 4). Ele então cria um nó híbrido com esses atributos e o coloca na fila de entrada de todas as partições envolvidas.

Algoritmo 6 Escalonamento Híbrido: tipos de dados e variáveis

1: **Tipos de dados:**
2: $Shard : \{$
3: $queue \leftarrow \emptyset$, *{fila de entrada}*
4: $cos \leftarrow \langle N : \bot, R : \emptyset \rangle$, *{lock-free COS estendido - Alg. 10}*
5: $ready \leftarrow new\ Semaphore(0)$ *{controle dos nós prontos}*
6: $space \leftarrow new\ Semaphore(maxSize/|S|)$ *{controle do espaço disponível}*
7: $\}$
8: $Req : \{...\}$ *{definido no Algoritmo 4}*
9: $Node : \{...\}$ *{definido no Algoritmo 3}*
10: $HyNode\ extends\ Node : \{$ *{especialização do Node}*
11: $s_{id} : S$, *{identificador da partição responsável}*
12: $rem_s : int$ *{contador de paralelizadores remanescentes}*
13: with $nxt : HyNodeRef$
14: with $depOn[]$: array of sets of $HyNodeRef$, one per shard
15: with $depMe[]$: array of sets of $HyNodeRef$, one per shard
16: $\}$
17: $HyNodeRef$ **atomic** reference to $HyNode$
18: **Variáveis:**
19: C *{definido no Algoritmo 4}*
20: S *{conjunto de identificadores das partições}*
21: $CtoS : C \rightarrow \mathcal{P}(S)$ *{uma classe mapeia para um subconjunto das partições}*
22: $shards[S]$ *{uma partição por identificador de partição}*

Algoritmo 7 Escalonamento híbrido: classificador (1 por réplica)

1: **Classificador executa assim:**
2: **onDeliver**$(req : Request)$:
3: $shards_{ids} \leftarrow CtoS(req.classId)$ *{identificação das partições envolvidas}*
4: $ss_{id} \leftarrow selectShard(req.classId)$ *{seleção da partição responsável}*
5: $rem_s \leftarrow shards_{ids}.size$ *{número de paralelizadores envolvidos}*
6: $node \leftarrow createNode(req.c, ss_{id}, rem_s)$ *{COS HybridNode}*
7: **for all** $s \in shards_{ids}$ **do** *{para cada partição envolvida}*
8: $shards[s].queue.fifoPut(node)$ *{coloca na fila de entrada da partição}*
9: **Funções auxiliares:**
10: **selectShard**$(class_{id} : C) : S$
11: **return** shard s escolhido em round-robin dentre $CtoS(class_{id})$
12: **createNode**$(c : Command, s_{id} : int, rem_s : int)$: HyNodeRef
13: **return reference to new** $HyNode\{c, \bot, \emptyset, \emptyset, \bot, s_{id}, rem_s\}$

Em seguida, a requisição é tratada pelo paralelizador em cada partição (Algoritmo 8). O paralelizador da partição segue lendo a fila de entrada em ordem FIFO, em busca do

próximo nó a agendar (linha 3). Se o paralelizador pertencer à partição responsável pelo nó (linha 4), aloca espaço para armazenar o nó (linha 5) e cria as dependências (linha 6); caso contrário, ele apenas cria as dependências (linha 8). Observe que o identificador da partição é usado na operação de inserção para indicar em qual subgrafo o nó deve ser incluído.

Algoritmo 8 Escalonamento híbrido: paralelizador (1 por partição)

```
1: Paralelizador da partição s_id executa assim:
2:   while true do                                   {laço infinito}
3:     node ← shards[s_id].queue.fifoPull()          {aguarda próximo nó disponível na fila de entrada}
4:     if node.s_id = s_id then                      {se este é o paralelizador responsável...}
5:       shards[s_id].space.down()                   {...garanta o espaço para inserção...}
6:       shards[s_id].cos.insert(node, s_id, true)   {...do nó e suas dependências}
7:     else                                          {caso contrário...}
8:       shards[s_id].cos.insert(node, s_id, false)  {...insira só as dependências}
```

Antes de nos aprofundarmos nos detalhes das operações do COS livre de bloqueios estendido para o escalonamento híbrido, descrevemos como as *threads* de execução processam os comandos (Algoritmo 9). Este algoritmo adapta o Algoritmo 1 associando cada *thread* executora a uma partição s_{id} para que processe suas requisições. Sempre que houver um nó livre de dependências nessa partição (linha 3), ele é recuperado (linha 4), executado (linha 5) e logicamente removido do COS (linha 6), liberando espaço para novos nós (linha 7).

Algorithm 9 Escalonamento híbrido: executor (vários por partição)

```
1: Cada thread executora da partição s_id executa assim:
2:   while true do                                   {laço infinito}
3:     shards[s_id].ready.down()                     {aguarda a sinalização de nó pronto}
4:     node ← shards[s_id].cos.get()                 {recupera um nó livre para execução}
5:     execute(node.c)                               {executa o comando do nó}
6:     shards[s_id].cos.remove(node)                 {marca o nó como logicamente removido}
7:     shards[s_id].space.up()                       {libera espaço para novos nós}
```

O Algoritmo 10 apresenta a implementação do COS estendido para lidar com um serviço particionado. Cada nó pertence a uma partição responsável que foi selecionado na linha 4 do Algoritmo 7 e está relacionado a todas as outras partições envolvidas em sua classe. Os nós que pertencem à partição são armazenados em N (linha 2), e os nós relacionados são armazenados em R (linha 3). Cada partição é responsável por alocar espaço, incluir, executar e excluir nós em N. Durante o tratamento de nós em N, dependências para outras partições devem ser respeitadas. Para cada partição, os nós relacionados

representam nós cujas classes foram mapeadas para esta partição, mas cuja partição responsável é outra. Consequentemente, estes nós são armazenados no N de outra partição, mas podem ter dependências com os nós desta partição e, portanto, devem ser verificados durante uma inserção. O conjunto R é manipulado apenas pelo paralelizador da partição, portanto também não apresenta condições de corrida.

A operação ***insert*** tem como argumento um *flag* que informa se o nó deve ser incluído em N (linha 9) ou em R (linha 11). Depois que o nó é processado em todos os subgrafos em que está envolvido (linha 12), Ele muda seu status para espera (wtg) e é testado quanto a sua prontidão (linhas 13-14). A operação ***remove*** marca o nó como logicamente removido (linha 16) e avalia se os nós dependentes tornaram-se executáveis. Finalmente, ***get*** é a mesma operação do Algoritmo 2. Ele retorna o primeiro, e mais antigo, nó pronto no subgrafo.

As funções *insertDeps* e *insertRelatedDeps* inserem as dependências relacionadas aos nós em N e R, respectivamente. Nessas funções, os nós removidos logicamente são removidos fisicamente das estruturas. A função *removeDeps* remove as arestas de entrada para um nó removido, permitindo sua coleta pelo coletor de lixo e evitando vazamentos de memória. A função *bind* é responsável por incluir as arestas que representam uma dependência entre dois nós. Semelhante ao escalonamento tardio, a função *testReady* verifica se um nó está pronto para ser executado.

4.4 Discussão da corretude

O escalonamento híbrido combina as principais ideias do escalonamento antecipado com as do escalonamento tardio, adicionando alguns refinamentos. Os comandos são entregues a cada réplica na mesma ordem total pelo *multicast* atômico. O classificador inspeciona sequencialmente os comandos e, de acordo com suas classes, os coloca na fila de entrada de um ou mais paralelizadores. Pelas propriedades discutidas no escalonamento antecipado, temos que a ordem de entrega é respeitada em todas as filas dos paralelizadores.

Cada classe com conflitos externos representa uma possível combinação de partições envolvidas no comando. Um comando de partição única pertence a uma classe que mapeia para o paralelizador que representa apenas essa partição, enquanto um comando multi-particionado pertence a uma classe que mapeia para os paralelizadores das partições envolvidas.

O paralelizador de uma partição recebe comandos de sua fila de entrada, ao mesmo tempo que preserva a ordem de entrega e calcula conflitos contra comandos pendentes na sua partição. Isso garante que nunca haverá ciclos de dependência dentro ou entre as

Algoritmo 10 COS Livre de bloqueios (Lock-Free) estendido

1: **Variáveis:**
2: $N : HyNodeRef,\ initially\ \perp$ *{Lista de HyNodes no subgrafo}*
3: $R : set\ of\ HyNode,\ initially\ \emptyset$ *{Conjunto de HyNodes relacionados}*

4: **Interface:**
5: **insert**$(n_n : HyNodeRef, s_{id} : int, insert_{node} : boolean)$
6: $insertRelatedDeps(n_n, s_{id})$ *{constrói as dependências com R}*
7: $n \leftarrow insertDeps(n_n, s_{id})$ *{constrói as dependências com N}*
8: **if** $insert_{node}$ **then** *{insere o nó neste subgrafo...}*
9: **if** $n = \perp$ **then** $N \leftarrow n_n$ **else** $n.nxt \leftarrow n_n$
10: **else** *{...ou no conjunto de nós relacionados}*
11: $R \leftarrow R \cup \{n_n\}$
12: **if** $decrementAndGet(n_n.rem_s) = 0$ **then** *{se este é o último paralelizador...}*
13: $n_n.st \leftarrow wtg$ *{...o nó passa para wtg...}*
14: $testReady(n_n)$ *{...e é testado quanto a sua prontidão}*

15: **remove**$(n :\ HyNodeRef)$ *{assumindo que n existe e tem n.st = exe}*
16: $n.st \leftarrow rmd$ *{remove n logicamente}*
17: **for all** $s_{id} \in S$ **do** *{para todas as partições}*
18: **for all** $n_i \in n.depMe[s_{id}]$ **do** *{para todos os nós que dependem de n}*
19: $testReady(n_i)$ *{verifica se n_i está pronto}*

20: **get**$() : HyNodeRef$ *{definido no Algoritmo 2}*

21: **Funções auxiliares:**
22: **insertRelatedDeps**$(n_n : NodeRef, s_{id} : int)$
23: **for all** $n \in R$ **do** *{para cada nó n em R}*
24: **if** $n.st = rmd$ **then** *{se n foi logicamente removido...}*
25: $removeDeps(n, s_{id})$ *{...remove suas dependências...}*
26: $R \leftarrow R \setminus n$ *{...e remove n de R}*
27: **else if** $conflict(n_n, n)$ **then** *{caso contrário, se existe conflito com n_n...}*
28: $bind(n_n, n, s_{id})$ *{...insere a dependência entre n e n_n}*

29: **insertDeps**$(n_n : NodeRef, s_{id} : int) : NodeRef$
30: $n' \leftarrow\ n \leftarrow N$
31: **while** $n' \neq \perp$ **do** *{para cada nó n' em N, por ordem de chegada}*
32: **if** $n'.st = rmd$ **then** *{se n' foi logicamente removido}*
33: $removeDeps(n', s_{id})$ *{remove suas dependências}*
34: **if** $!compareAndSet(N, n, n'.nxt)$ **then** *{se não é o primeiro nó}*
35: **atomic** { $n.nxt \leftarrow n'.nxt$ } *{remove n' do subgrafo}*
36: **else if** $conflict(n_n, n')$ **then** *{caso contrário, se há um conflito...}*
37: $bind(n_n, n', s_{id})$ *{...insere uma dependência}*
38: $n \leftarrow n';\ n' \leftarrow n'.nxt$ *{vai para o próximo}*
39: **return** n *{retorna o último nó em N}*

40: **removeDeps**$(n : NodeRef, s_{id} : int)$
41: **for all** $n_i \in n.depMe[s_{id}]$ **do** *{para cada dependência}*
42: $n_i.depOn[s_{id}] \leftarrow n_i.depOn[s_{id}] \setminus \{n\}$ *{remova-a}*

43: **bind**$(n_{new}, n_{old} : NodeRef, s_{id} : int)$
44: $n_{old}.depMe[s_{id}] \leftarrow n_{old}.depMe[s_{id}] \cup \{n_{new}\}$ *{n_{new} depende...}*
45: $n_{new}.depOn[s_{id}] \leftarrow n_{new}.depOn[s_{id}] \cup \{n_{old}\}$ *{...de n_{old}}*

46: **testReady**$(n : NodeRef)$
47: **if** $\{n_i \in \{\exists s_{id} \in S : n_i \in n.depOn[s_{id}] \wedge n_i.st \neq rmd\}\} = \emptyset$ **then** *{se n não tem dependências...}*
48: **if** $compareAndSet(n.st, wtg, rdy)$ **then** *{...e se consegue mudar n de wtg para rdy...}*
49: $shards[n.s_{id}].ready.up()$ *{...sinalize um novo nó pronto em s_{id}}*

50: **decrementAndGet**$(value : int) : int$
51: **return atomic** $\{value \leftarrow value - 1;\ value\}$

52: **compareAndSet**$(a, b, c) : boolean$ *{definido em Algoritmo 3}*

53: **conflict**$(n_i, n_j : Node) : boolean$ *{definido em Algoritmo 3}*

partições - este último porque a ordem total é preservada em todas as filas de entrada das partições.

Comandos multi-particionados só são considerados completamente escalonados quando todos paralelizadores envolvidos já calcularam suas dependências para os comandos pendentes em suas respectivas partições. Isso é garantido pelo contador de paralelizadores envolvidos remanescentes do nó (rem_s) que é atomicamente decrementado sempre que um paralelizador completa sua análise de dependências. Portanto, um nó nunca será considerado para execução antes de todos os paralelizadores envolvidos o processarem e suas dependências serem totalmente representadas.

Um comando é considerado para execução apenas se estiver livre de dependências. Isso é garantido por *get*, que leva apenas comandos prontos, e por *testReady*, que muda os comandos de esperando para pronto quando suas dependências são resolvidas. A função *testReady* é disparada sempre que um novo nó é inserido ou um nó é logicamente removido, assim ela verifica se o novo nó está livre de dependências ou se a remoção de um nó liberou qualquer outro nó para execução, respectivamente.

Os argumentos acima mostram que os comandos dependentes são executados de acordo com a ordem total, enquanto comandos independentes de partições diferentes ou da mesma partição são executados simultaneamente. Devido às dependências acíclicas, conforme discutido acima, considerando todas as partições, em todos os momentos, sustenta-se que existe sempre um comando mais antigo que não depende de nenhum anterior e que pode ser executado, ou não há qualquer comando no grafo. Quando este comando é executado, ele é logicamente removido e as dependências são resolvidas. Consequentemente, ele irá liberar indutivamente outros comandos, resultando em uma estrutura de dependências com a mesma propriedade acima, garantindo o progresso.

Complementando, cada comando tem uma partição responsável associada. Os executores associados à partição responsável considerarão seus comandos para execução sempre que estiverem prontos, garantindo que comandos prontos em todas as partições sejam executados.

Em relação ao acesso simultâneo sem bloqueio ao gráfico de dependência, a segurança em cada partição tem os mesmos argumentos da Seção 3.1.3. Além disso, durante a inserção de um nó multi-particionado, os paralelizadores das partições envolvidas calculam simultaneamente as dependências do nó de entrada para diferentes conjuntos de nós. Isso é feito adicionando referências para outros nós aos atributos de nó *depOn* e $depMe$ (em posições diferentes dos *arrays*). Esses atributos são construídos com conjuntos diferentes para cada partição. Uma vez que cada paralelizador atualiza um conjunto diferente, a operação é tanto segura quanto livre de problemas de sincronização entre paralelizadores.

Quando pronto, o nó é recuperado (*get*) por uma *thread* executora pertencente à partição responsável pelo nó. Essa *thread* executa e remove logicamente o nó. Durante a remoção física, todos os nós em $depMe$ são visitados para atualizar seus respectivos conjuntos *depOn*, pelo paralelizador de cada partição. Quando executores de diferentes partições simultaneamente removem nós que afetam (liberam dependências) um nó comum, o conjunto *depOn* do nó comum será acessado simultaneamente. Novamente, devido à existência de uma lista de dependências por partição em cada nó, isso é garantido como seguro e livre de sincronização entre *threads* executoras de diferentes partições.

4.5 Exemplo de execução

Para facilitar o entendimento vamos apresentar um exemplo de execução descrevendo a jornada de 4 comandos de uma aplicação fictícia no escalonamento híbrido. A aplicação é um sistema imaginário bi-particionado com operações de leitura e escrita locais (em partições individuais) e de escritas globais (em todas as partições). Em sistemas assim é comum que as operações locais de uma partição não conflitem com as operações locais de outra partição. No nosso sistema hipotético as escritas locais conflitam com outras escritas e leituras locais, ou seja, da mesma partição. As leituras locais não conflitam entre si e as escritas globais conflitam com todas as operações.

O escalonamento híbrido cria uma classe para cada partição, que é mapeada para o seu paralelizador correspondente, e ainda uma classe para cada combinação de partições, que é mapeada para os paralelizadores correspondentes (Seção 4.3.1). Neste exemplo, teremos as classes C_{S1} e C_{S2} para as operações endereçadas às partições 1 e 2, respectivamente. Além disso, teremos a classe C_{S12} para as operações que acessam as duas partições. No nosso exemplo consideramos também que existem quatro *threads* de execução distribuídos entre as duas partições, os executores 1 e 2 pertencem à partição 1 e os executores 3 e 4 pertencem à partição 2.

O mapeamento de classes para partições (*shards*) (CtoS), utilizado pelo classificador, fica assim:

- $C_{S1} \rightarrow Shard_1$ (as requisições são encaminhadas para a fila do paralelizador da partição 1);
- $C_{S2} \rightarrow Shard_2$ (as requisições são encaminhadas para a fila do paralelizador da partição 2);
- $C_{S12} \rightarrow Shard_1 \wedge Shard_2$ (as requisições são encaminhadas para as filas de ambos os paralelizadores).

Vamos acompanhar 4 operações específicas $O_{R1} \in C_{S1}$, $O_{R2} \in C_{S2}$, $O_{W1} \in C_{S1}$ e $O_{Wg} \in C_{S12}$ que representam respectivamente, uma leitura na partição 1, uma leitura na

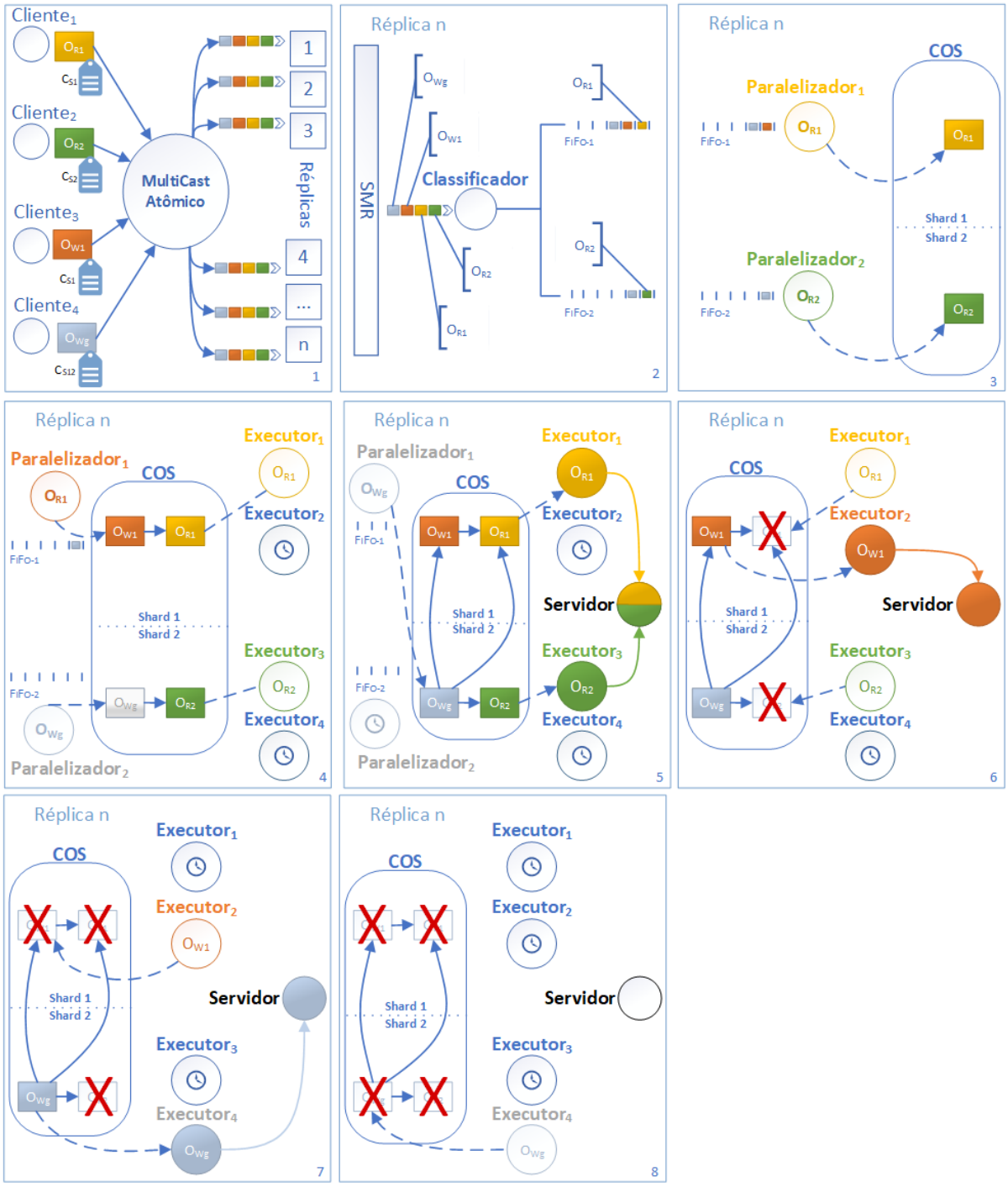


Figura 4.2: Exemplo de funcionamento do escalonamento híbrido.

partição 2, uma escrita na partição 1 e uma escrita global. O quadro 1 da Figura 4.2 apresenta as quatro operações sendo enviadas por quatro processos clientes distintos. Os quadros de 2 a 8 da Figura 4.2 representam as estruturas internas de uma réplica.

A difusão das operações é realizada por uma biblioteca de *multicast* atômico. A

primitiva de comunicação é síncrona e bloqueante, ou seja, a *thread* do cliente que realiza a transmissão fica bloqueada aguardando a resposta do serviço remoto. A biblioteca de *multicast* atômico ordenará as requisições e as entregará em todas as réplicas do serviço na mesma ordem total.

Perceba que, no quadro 1, assim como no escalonamento antecipado, antes de entregar os comandos à camada de transmissão e ordenação, cada cliente rotula seu comando com o identificador da classe de requisições adequada. Esse rótulo é o que determina quais partições o classificador deve considerar ao tratar cada requisição.

No quadro 2 podemos ver que o classificador colocou o comando O_{R1} e O_{W1} na fila de entrada do paralelizador da partição 1, o comando O_{R2} na fila do paralelizador da partição 2 e o comando O_{Wg} nas duas filas.

O quadro 3 apresenta os paralelizadores 1 e 2 consumindo os comandos de suas filas e inserindo-os em suas respectivas partições do COS. Cada comando que chega a um paralelizador possui a indicação da sua partição responsável (Algoritmo 8 linha 4), devido a essa indicação foi possível eliminar a sincronização por meio de barreiras. Ao alcançar o comando, o paralelizador identifica se é o responsável por sua inserção no COS ou só pelo cálculo de parte de suas dependências.

No quadro 4, os paralelizadores continuam construindo seus subgrafos de dependência enquanto os executores 1 e 3 já foram despertados e seguem varrendo o grafo em busca dos comandos prontos. Observe que o comando O_{Wg} encontra-se parcialmente inserido, dependendo exclusivamente do comando O_{R2}. A escrita global pertence às duas partições e até o momento retratado na imagem só o paralelizador 2 a havia tratado, por isso não vemos as relações entre O_{Wg} e a partição 1.

No quadro 5, os executores 1 e 3 processam os comandos O_{R1} e O_{R2} ao mesmo tempo. Enquanto o paralelizador 1 finalmente alcança o comando O_{Wg} e cria suas dependências para os comandos presentes na partição 1. O paralelizador 2 encontra-se bloqueado, aguardando novos comandos serem inseridos em sua fila de entrada.

No quadro 6, as operações O_{R1} e O_{R2} são finalizadas e marcadas como logicamente removidas nos respectivos subgrafos, liberando a operação O_{W1} para execução por não possuir mais dependências não processadas. O executor 2 é despertado e varre a sua partição do COS em busca do novo comando pronto, que ao ser encontrado é executado imediatamente.

No quadro 7, a operação O_{W1} é finalizada e logicamente removida. Essa ação, por sua vez, libera a execução da operação O_{Wg}. O executor 4 é despertado e varre a sua partição do COS em busca do comando disponível, que ao ser encontrado é executado imediatamente.

Por fim, o quadro 8 apresenta o executor 4 marcando a operação O_{Wg} como logicamente

removida, logo após sua execução. Neste exemplo, os quatro comandos acompanhados permaneceram no grafo de dependências, mesmo estando logicamente removidos. A remoção física dos comandos de cada partição do COS é realizada pelo respectivo paralelizador no ato da inserção de novos comandos.

4.6 Otimizações

A primeira otimização importante que introduzimos no escalonamento híbrido foi a mudança da criação de nós do classificador (Algoritmo 7) para os paralelizadores (Algoritmo 8) em comandos de partição única. Isso é possível porque há apenas uma partição envolvida, portanto não há outros paralelizadores compartilhando informações desse nó. Também evitamos a execução atômica da linha 12 no algoritmo 10 e simplesmente executamos as linhas 13 e 14 porque a instrução if sempre retorna verdadeiro nesses casos. Outra observação importante sobre o escalonamento híbrido é que optamos por atribuir cada executor a uma partição em vez de usar executores globais. Essa abordagem apresentou um melhor desempenho, pois menos executores compartilham estruturas comuns (principalmente semáforos) e já conhecem a partição onde devem procurar um comando livre.

4.7 Conclusões

Neste capítulo foi apresentado o modelo de sistema, os algoritmos e detalhes de implementação do protocolo de escalonamento híbrido para Replicação Máquina de Estados Paralela elaborado neste trabalho. Além disso, foi apresentada uma discussão sobre a sua corretude e um exemplo de execução. No Capítulo 5 serão expostos os resultados dos testes experimentais realizados sobre uma RMEP com H-SMR.

Capítulo 5

Experimentos e Estudo de Caso

Este capítulo apresenta os resultados de uma avaliação experimental dos protocolos de Replicação Máquina de Estados clássica e Replicação Máquina de Estados Paralela com escalonamento tardio, antecipado e híbrido. Posteriormente foi realizado um estudo de caso onde aplicamos a RMEP com escalonamento híbrido na construção de um protótipo de sistema de pagamento inspirado na SmartCoin [41], que é uma aplicação de moeda virtual construída sobre o a BFT-SMaRt [55].

5.1 Experimentos

Os 4 protocolos selecionados foram exercitados de 4 formas diferentes com o objetivo de mostrar: (1) as vantagens e desvantagens das abordagens tardia e antecipada (Seção 5.1.4); (2) como o escalonamento híbrido contorna esses problemas de desempenho, aproveitando as melhores características de cada uma dessas técnicas em um sistema multi-particionado (Seção 5.1.5); (3) o desempenho dos protocolos com cargas de trabalho distorcidas/desbalanceadas (Seção 5.1.6); e (4) o desempenho das estruturas de dados sozinhas, sem integração em um *framework* RME (Seção 5.1.7).

5.1.1 Implementação

Os algoritmos que compõe o escalonamento híbrido nas réplicas descritos nas seções anteriores foram implementados na linguagem de programação Java, tendo como base a implementação *LockFree* proposta em Escobar et al [14] e disponível em `https://github.com/parallel-SMR/library`. Adicionalmente, a solução híbrida foi integrada a biblioteca de replicação de código aberto Bft-SMaRt [55, 56]. Escrita em Java, a biblioteca Bft-SMaRt implementa uma solução robusta de ordenação para Replicação Máquina de Estados (RME) tolerante a falhas bizantinas.

Usamos uma implementação eficiente e sem bloqueios do problema do consumidor único e produtor único [57] para as filas FIFO usadas nas abordagens antecipada e híbrida. Também configuramos o tamanho máximo do grafo de dependência para 150 posições nas abordagens tardia e híbrida.

5.1.2 Ambiente

Implementamos todas as técnicas de escalonamento usando a BFT-SMART [55], que pode configurada para usar protocolos otimizados para tolerar apenas falhas por colapso ou falhas bizantinas. Em todos os experimentos dessa seção, configuramos BFT-SMART para tolerar falhas por colapso. O protocolo de *broadcast* atômico da BFT-SMART executa uma sequência de instâncias de consenso, onde cada instância ordena um lote de comandos. Para aumentar o desempenho do protocolo de ordenação do BFT-SMART, implementamos interfaces para permitir que os clientes enviassem um lote de comandos a cada mensagem.

O ambiente experimental foi configurado com 7 máquinas conectadas a uma rede comutada de 1Gbps. As máquinas foram configuradas com o sistema operacional Ubuntu Linux 18.04 e uma máquina virtual Java de 64 bits, versão 10.0.2. O BFT-SMART foi configurado com 3 réplicas hospedadas em máquinas separadas (Nós Dell PowerEdge R815 equipados com quatro processadores AMD Opteron 6366HE de 16 núcleos rodando a 1,8 GHz com 128 GB de RAM) para tolerar o *crash* de até uma réplica. Utilizamos também 800 clientes distribuídos uniformemente em outras 4 máquinas (nós HP SE1102 equipados com dois processadores Intel Xeon L5420 quad-core rodando a 2,5 GHz com 8 GB de RAM). Os experimentos com as estruturas de dados (grafos) sozinhas, sem integração em uma RME (Seção 5.1.7), foram executados em uma das máquinas Dell PowerEdge.

5.1.3 Aplicação

Foram implementados serviços não particionados e multi-particionados (*multi-sharded*). Para o serviço não particionado, usamos um aplicativo de lista vinculada com operações para verificar se uma entrada i (ou seja, um inteiro) está na lista (*contains*) e para incluir uma entrada j na lista (*add*), representando um serviço de leitores e escritores. A operação $contains(i)$ retorna *true* se a entrada i estiver na lista, caso contrário, retorna *false*. A operação $add(j)$ verifica se j está na lista, se estiver ela retorna *false*, se não estiver ela inclui j na lista e retorna *true*. Neste contexto, os comandos *contains* não entram em conflito uns com os outros, mas entram em conflito com os comandos *add*, que entram em conflito com todos os comandos.

Para o serviço multi-particionado, usamos um conjunto de aplicativos de listas vinculadas (um por partição) com operações de partição única e multi-partição. Para verificar se uma entrada está em um subconjunto de listas ($contains_S(i)$ executa $contains(i)$ nas listas associadas a cada partição $k \in S$) e para incluir uma entrada em um subconjunto de listas ($add_S(i)$ executa $add(i)$ nas listas associadas a cada partição $k \in S$). Observe que $|S| = 1$ para comandos de partição única. Neste caso, $add_{S'}$ entra em conflito com cada $contains_{S''}$ e com cada $add_{S''}$ se $S' \cap S'' \neq \emptyset$.

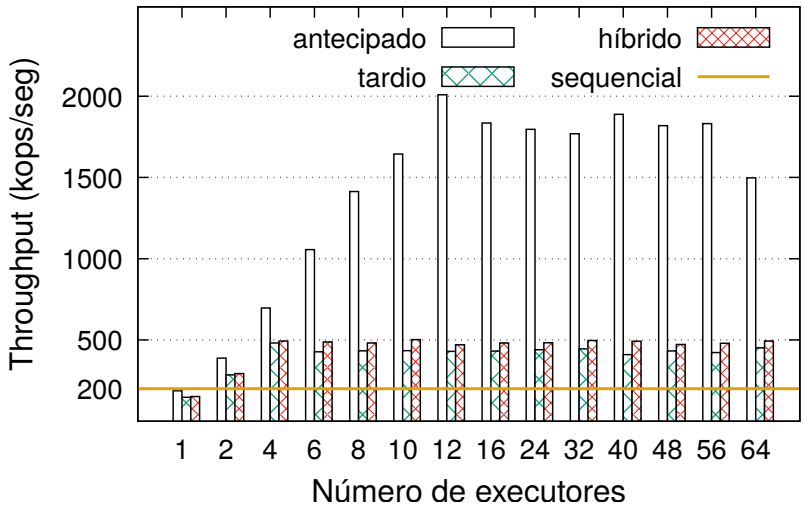
A seguir, nos referimos às operações que verificam se uma entrada está em uma ou mais listas como operações de *leitura* e às operações que incluem uma entrada em uma ou mais listas como operações de *escrita*. Cada lista foi inicializada com $1k$ ou $10k$ entradas em cada réplica (variando de 0 até *tamanho da lista - 1*), representando operações com diferentes custos de execução. O argumento inteiro usado nas operações de *leitura* e *escrita* foi sempre uma posição escolhida aleatoriamente na lista.

Nos experimentos, medimos a vazão do sistema nos servidores e a latência de cada comando nos clientes. Nos experimentos com as estruturas de dados sozinhas, medimos apenas a vazão geral obtida pelas *threads* de execução, uma vez que não faz sentido calcular a latência neste caso. Uma fase de aquecimento precedeu cada experimento.

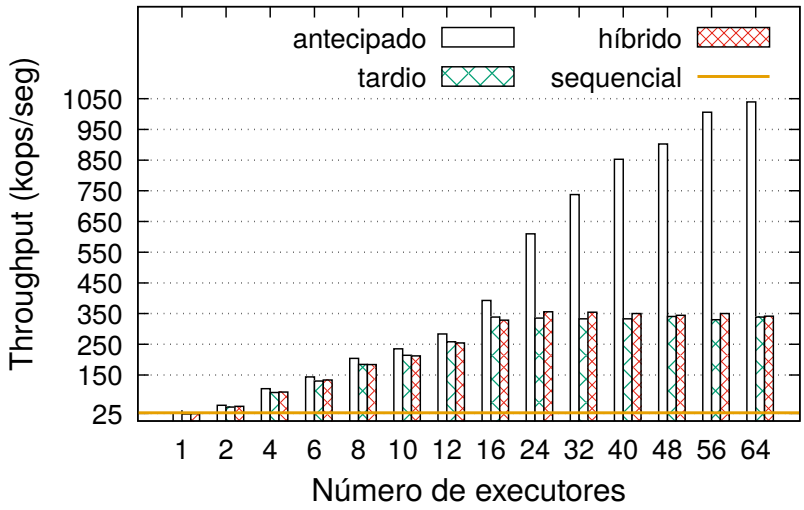
5.1.4 Sistemas não particionados

O primeiro conjunto de experimentos considera apenas leituras e escritas em uma única partição. A Figura 5.1 apresenta a vazão para uma carga só com operações de leitura aplicada à RME clássica (*sequential*) e a três RMEPs com abordagens de escalonamento tardia, antecipada e híbrida, respectivamente. É perceptível o destaque no desempenho do escalonamento antecipado nessa carga de trabalho sem conflitos. Isso ocorre pelo seu baixo custo de escalonamento no servidor, quando comparado com as abordagens que usam operações com grafos. O desempenho da RME clássica mantém-se constante em todos os cenários, independentemente da quantidade de *threads* de execução disponíveis, pois essa abordagem executa tudo de forma sequencial em uma só *thread*. A RME clássica no entanto muda de 200 mil operações por segundo (*ops/sec*) para 25 mil *ops/sec* quando o custo da operação é aumentado (i.e., aumentando o tamanho da lista de 1k para 10k posições).

A Figura 5.2 também apresenta a vazão das 4 abordagens de RME, só que para uma carga de trabalho mista, composta por 5% de escritas e 95% de leituras, distribuída uniformemente entre os clientes. Perceba que o desempenho do escalonamento antecipado diminui abruptamente quando há conflitos, além de piorar com a adição de *threads* executoras. Isso acontece porque o processamento no classificador é rápido, mas os executores precisam sincronizar entre si para executar comandos conflitantes.



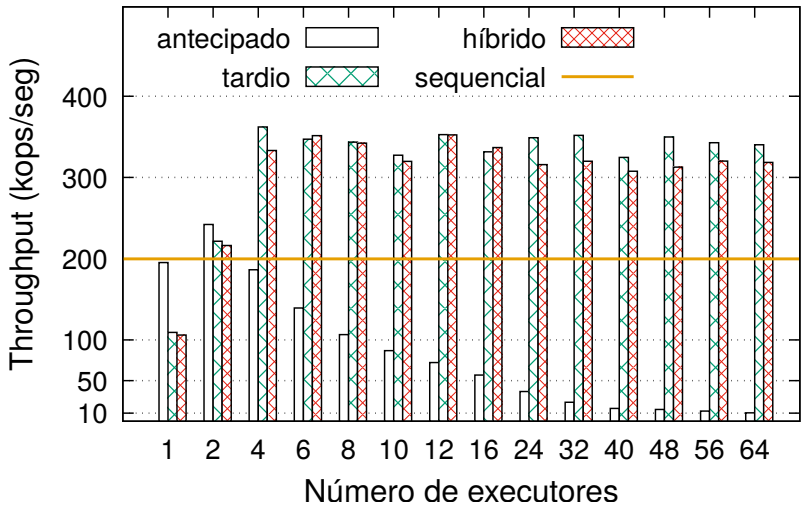
(a) lista com 1k posições.



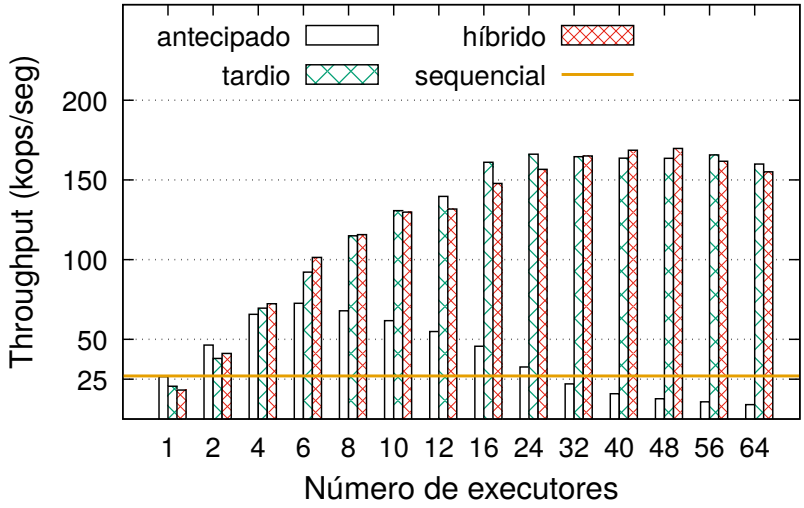
(b) lista com 10k posições.

Figura 5.1: Vazão para uma carga somente com leituras e diferentes números de executores.

Seguindo no mesmo estudo, a Figura 5.3 apresenta os resultados para uma carga de trabalho mista, composta por 10% de escritas e 90% de leituras, distribuída uniformemente entre os clientes. Perceba que em todos os cenários das figuras 5.1, 5.2 e 5.3 o escalonamento híbrido tem um desempenho semelhante ao tardio, pois em sistemas com apenas uma partição, ambas as abordagens funcionam de maneira semelhante. Pode se observar também que o desempenho das abordagens que rastreiam as dependências por meio de um grafo sem bloqueios alcança um limite superior que não é ultrapassado com a adição de mais *threads* de execução. Isso se deve ao paralelizador ser a *thread* responsável por manter o grafo, inserindo e removendo comandos (lembre-se de que as *threads* execu-



(a) lista com 1k posições.



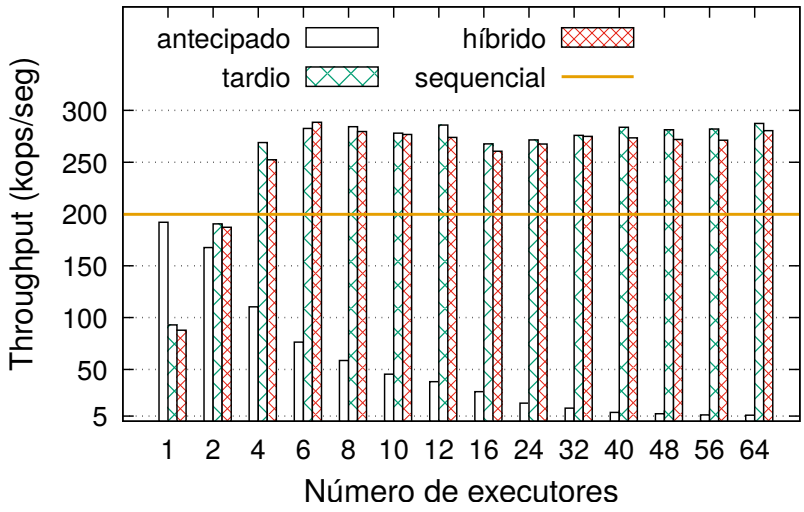
(b) lista com 10k posições.

Figura 5.2: Vazão para 5% de escritas e diferentes números de executores.

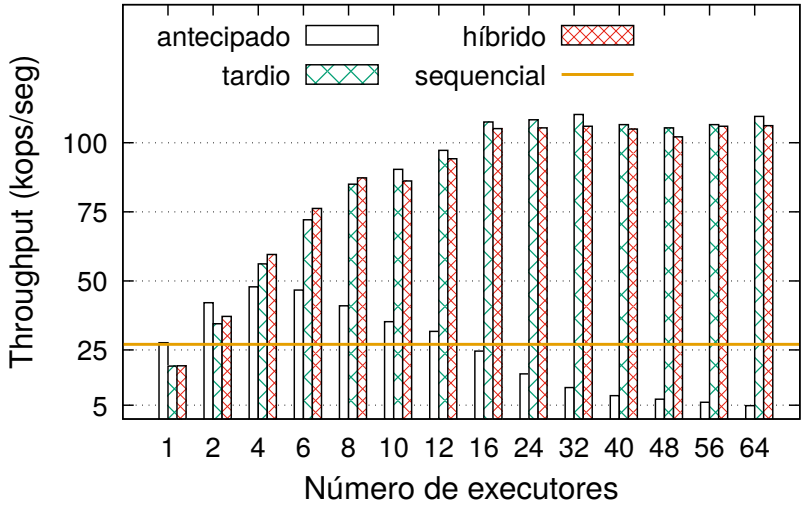
toras apenas marcam o comando como removido logicamente). Embora essas abordagens equilibrem melhor a carga de trabalho entre os executores, o paralelizador se torna um gargalo e o desempenho não aumenta com executoras adicionais.

5.1.5 Sistemas multi-particionados

O primeiro conjunto de experimentos mostrou até agora que o escalonamento tardio e híbrido apresentam desempenhos semelhantes em um sistema de partição única, e o escalonamento antecipado teve um desempenho melhor em cargas de trabalho sem conflitos. Uma questão natural é como essas abordagens funcionam em um sistema multi-



(a) lista com 1k posições.



(b) lista com 10k posições.

Figura 5.3: Vazão para 10% de escritas e diferentes números de executores.

particionado, uma vez que os escalonadores antecipado e híbrido tentam tirar vantagem dessa característica, enquanto o escalonador tardio não está ciente do particionamento. A Figura 5.4 apresenta a vazão das 4 abordagens de RME, para uma carga de trabalho apenas com operações de leitura de partição única; em um sistema multi-particionado considerando diferentes números de partições e *threads* executoras. As operações de partição única são distribuídas uniformemente entre as partições e todas as operações são distribuídas uniformemente entre os clientes. Para as abordagens antecipada e híbrida, o número de *threads* executoras diz respeito à quantidade de executores por partição, enquanto no escalonamento tardio esse número representa o total de executores. O tra-

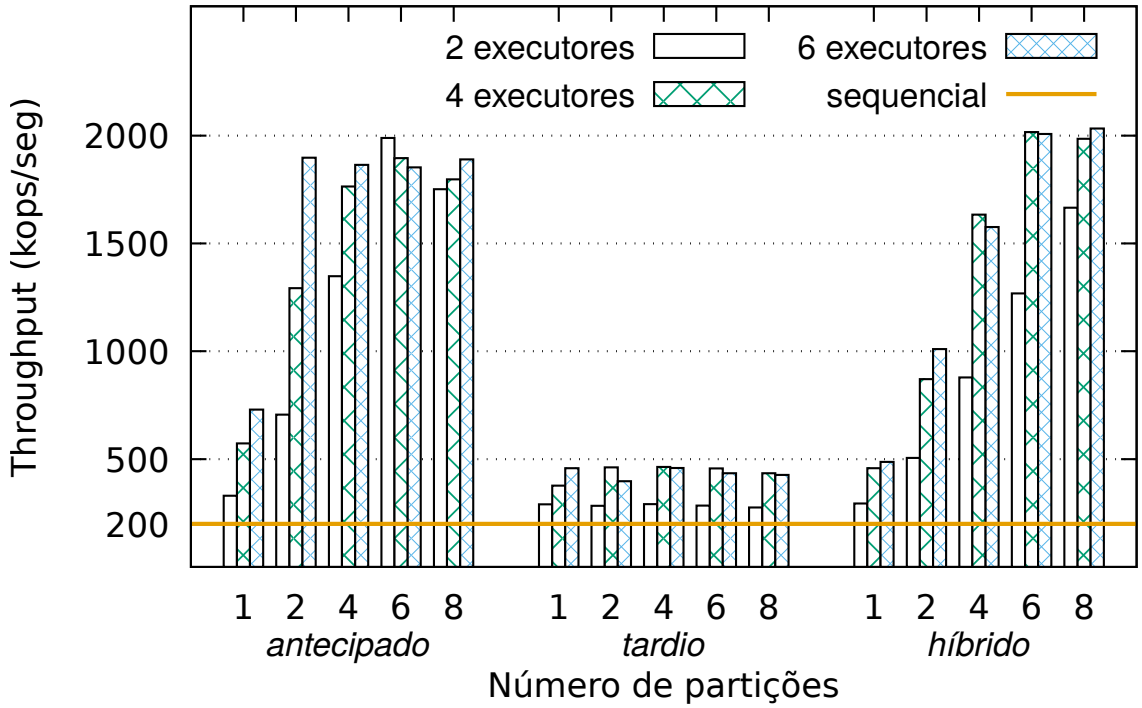


Figura 5.4: Vazão para uma carga somente com leituras locais para sistemas configurados com diferentes números de partições e *threads* de execução (lista de 1k entradas). Para as abordagens antecipada e híbrida, o número de executoras se refere à quantidade de executoras por partição, enquanto no escalonamento tardio esse número representa o total de executoras.

tamento diferenciado para o escalonamento tardio se deve ao fato dele ter atingido seu teto de desempenho com 6 executores e a adição de mais *threads* não ter influenciado o resultado.

A Figura 5.5 apresenta o mesmo estudo, para uma carga de trabalho mista composta por 5% de escritas e 95% de leituras, das quais 1% são operações multi-partição e 99% são operações de partição única; Sendo que as operações de partição única são distribuídas uniformemente entre as partições e todas as operações são distribuídas uniformemente entre os clientes. Além disso, 10% das operações multi-partição envolvem todas as partições, enquanto as outras são endereçadas a duas partições escolhidas aleatoriamente.

Continuamos o estudo com a Figura 5.6 que apresenta os resultados para uma carga de trabalho mista composta de 10% de escritas e 90% de leituras, das quais 5% são operações multi-partição e 95% são operações de partição única. Onde operações de partição única são distribuídas uniformemente entre as partições e todas as operações são distribuídas uniformemente entre os clientes. Além disso, 10% das operações multi-

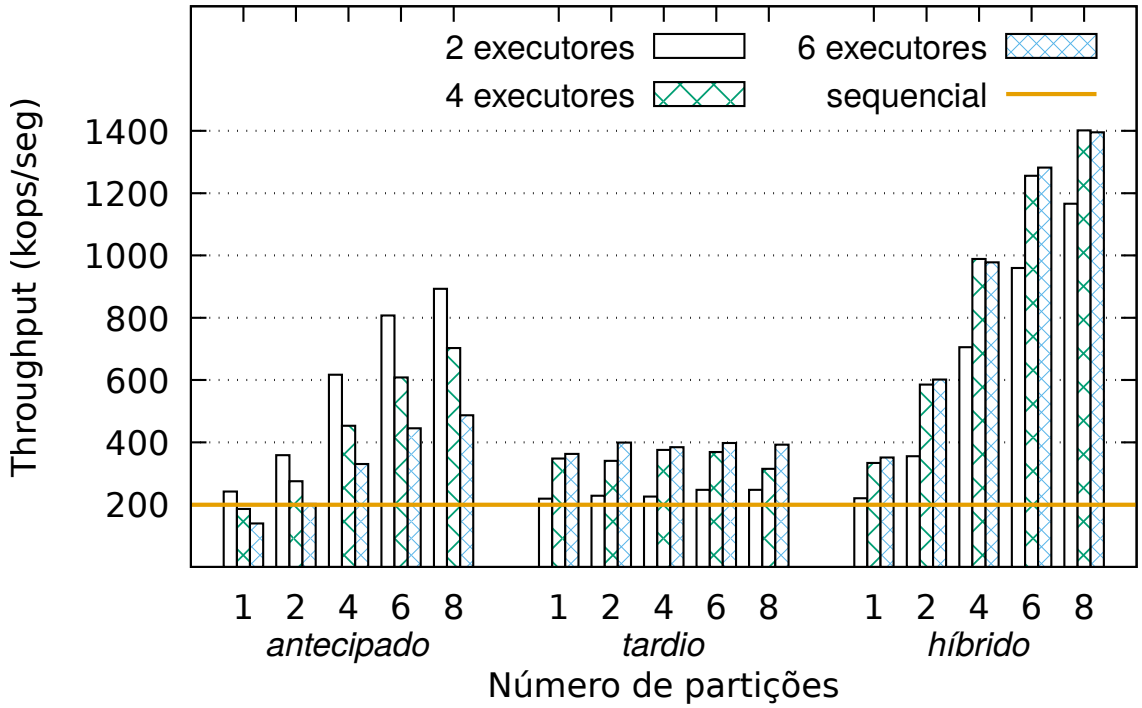


Figura 5.5: Vazão para 5% de operações de escrita, 1% de operações globais para sistemas configurados com diferentes números de partições e *threads* de execução (lista de 1k entradas). Para as abordagens antecipada e híbrida, o número de executoras se refere à quantidade de executoras por partição, enquanto no escalonamento tardio esse número representa o total de executoras.

partição envolvem todas as partições, enquanto as outras são endereçadas a duas partições escolhidas aleatoriamente.

Como o escalonamento híbrido usa um paralelizador por partição para inserir (e remover) comandos em (de) um subgrafo, seu desempenho é dimensionado com o número de partições. Ele também atinge o pico de vazão do escalonamento antecipado na carga de trabalho apenas com comandos de leitura, quando configurado com 6 ou mais partições. O escalonamento antecipado não escala com mais de 2 executores por partição quando existem comandos conflitantes e o desempenho do escalonamento tardio é limitado pela *thread* que mantém o grafo. Observe que o escalonamento tardio foi executado com menos *threads* do que as outras técnicas, visto que o paralelizador já se torna um gargalo de desempenho com poucas threads executoras.

A Figura 5.7 mostra os resultados de latência versus vazão para as configurações com melhor desempenho com 8 partições e a carga de trabalho com 5% global e 10% de escritas (Figura 5.6).

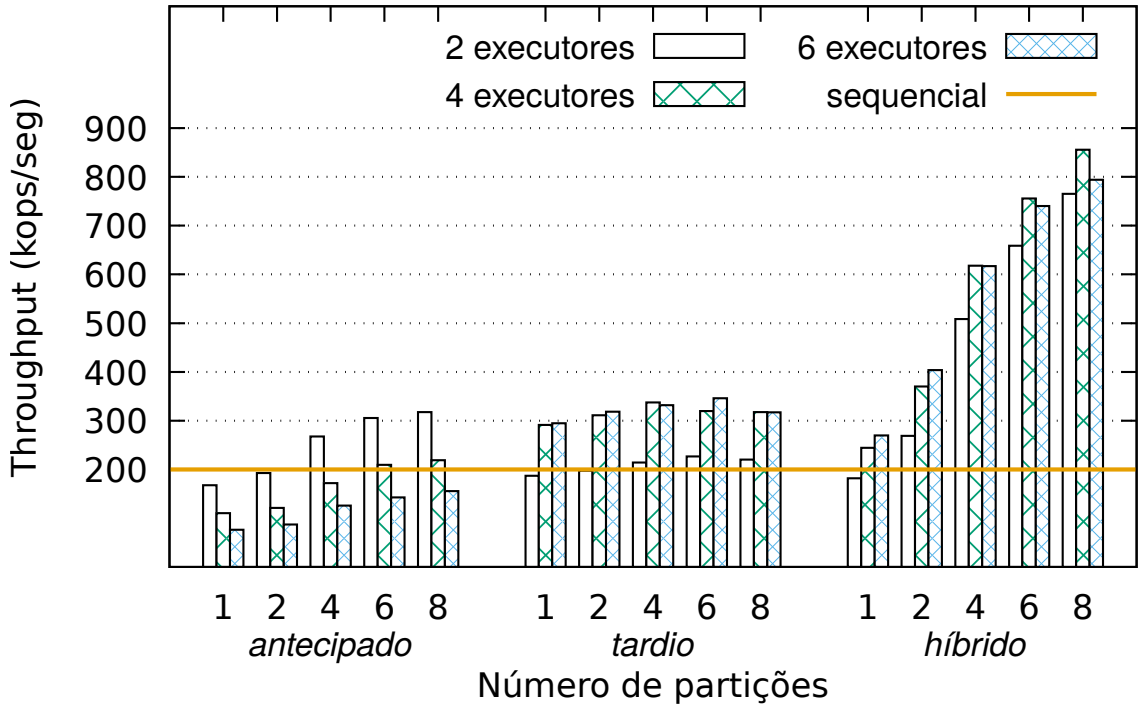


Figura 5.6: Vazão para 10% de operações de escrita, 5% de operações globais para sistemas configurados com diferentes números de partições e *threads* de execução (lista de 1k entradas). Para as abordagens antecipada e híbrida, o número de executoras se refere à quantidade de executoras por partição, enquanto no escalonamento tardio esse número representa o total de executoras.

Nas abordagens de Replicação Máquina de Estados Paralela (RMEP) estudadas, todos os comandos têm latência semelhante porque têm custos de execução semelhantes e a sincronização das escritas impacta o desempenho das leituras ordenadas após cada escrita. Obviamente, o mesmo comportamento ocorre na Replicação Máquina de Estados (RME) clássica. Consequentemente, a Figura 5.7 apresenta a latência média considerando todas as operações. É possível observar que todas as abordagens apresentaram características semelhantes de latência até próximo à saturação do sistema e, a partir deste ponto, a latência aumenta abruptamente. Como o mesmo comportamento ocorre para as demais configurações e cargas de trabalho, apresentamos apenas esses casos. O mesmo comportamento é relatado em trabalhos anteriores sobre RME (e.g., [14, 15, 55]).

5.1.6 Cargas de trabalho distorcidas

A Figura 5.8 apresenta a vazão de um sistema com 4 partições considerando as mesmas cargas de trabalho apresentadas na seção anterior (balanceado) e também para os casos

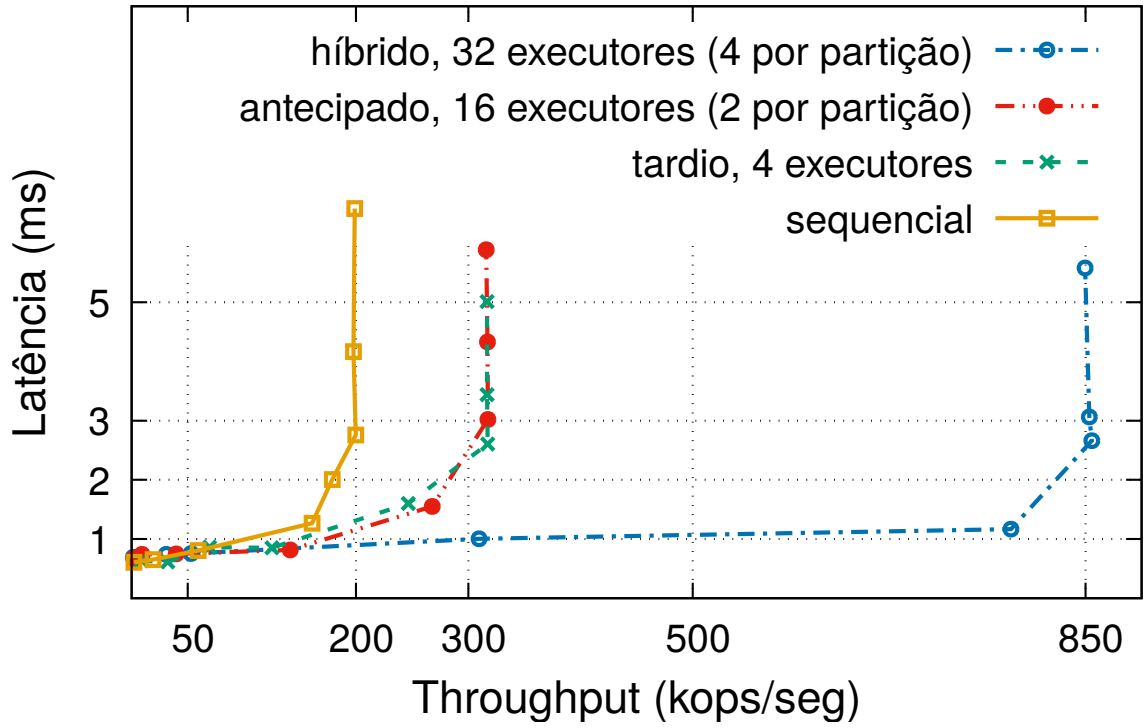


Figura 5.7: Latência versus vazão para um sistema com 8 partições e uma carga de trabalho composta por 10% das escritas e 5% de operações globais (lista de 1k entradas).

em que uma partição recebe 50% das operações de partição única enquanto o restante é uniformemente distribuído pelas outras três partições (enviesado). Para cada técnica, usamos a configuração que em geral apresentou melhor desempenho para cargas de trabalho balanceadas (figuras 5.4, 5.5 e 5.6).

O escalonamento tardio apresentou desempenho semelhante para cargas de trabalho distorcidas e balanceadas, uma vez que não faz distinção entre a existência ou não de partições. O desempenho do escalonamento antecipado e híbrido diminui em cargas de trabalho distorcidas, pois enquanto algumas partições estão sobrecarregadas, outras têm menos trabalho para processar. No entanto, o escalonamento híbrido ainda supera os outros por uma grande margem.

5.1.7 Desempenho da estrutura de dados

Esta seção relata o desempenho das estruturas de dados sozinhas (ou seja, sem integração em uma RME). Consideramos uma única réplica, onde uma *thread* percorre a lista de requisições pré-criadas (para poupar os tempos de criação) e entrega para escalonamento. Executamos os experimentos para uma carga de trabalho somente com leituras e sem ope-

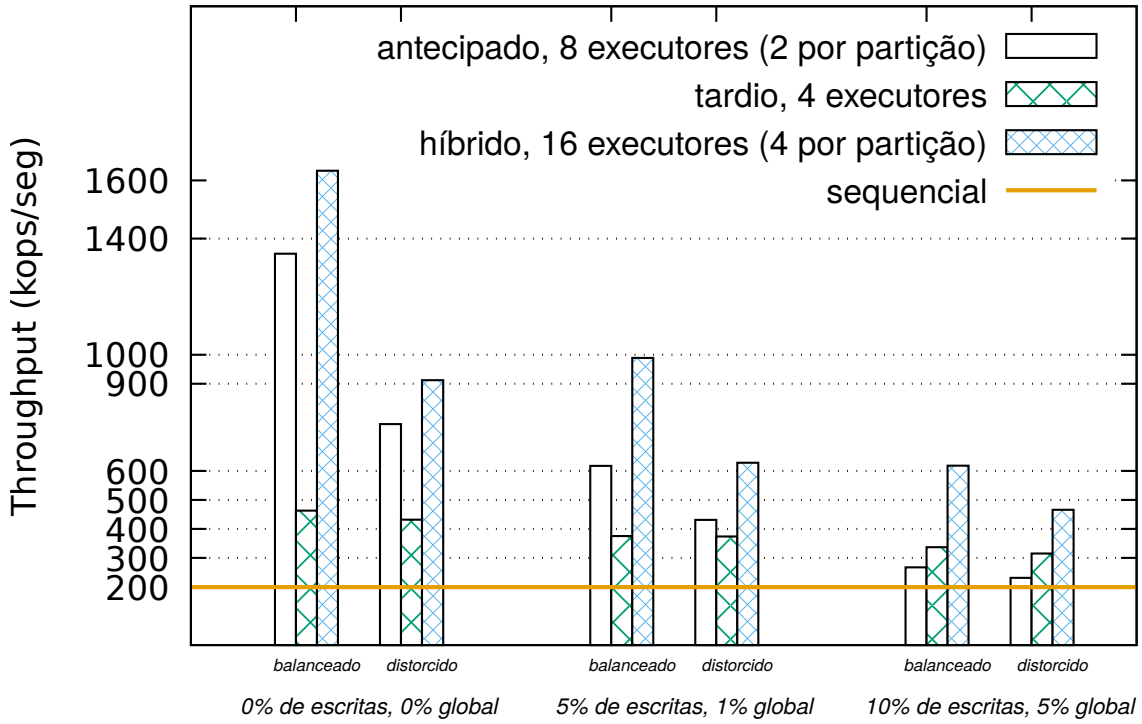


Figura 5.8: Vazão para um sistema com 4 partições considerando cargas de trabalho balanceadas e distorcidas (lista de 1k entradas).

rações globais, pois é a que mais desafia o RMEP por prover o maior grau de paralelismo possível.

Os resultados apresentados na Figura 5.9 relatam um desempenho semelhante para as técnicas quando integradas na estrutura de uma RME (Figura 5.4). Isso significa que o desempenho geral é limitado pelas sincronizações dentro de cada réplica, bem como pelo tempo exigido para criar objetos (por exemplo, nós a serem inseridos no gráfico de dependência e requisições serializadas nas mensagens recebidas a serem entregues nas camadas superiores).

5.2 Estudo de Caso

Replicação Máquina de Estados (RME) e *blockchains* possuem um objetivo em comum que é o de manter a consistência no estado de um serviço replicado. RMEs tolerantes a falhas bizantinas (BFT) [58] e [59] são particularmente relevantes no contexto de *blockchains* permissionados [60], por suportarem que uma fração das réplicas seja controlada por um adversário. Estes protocolos também são usados em *blockchains* não permissionados, onde

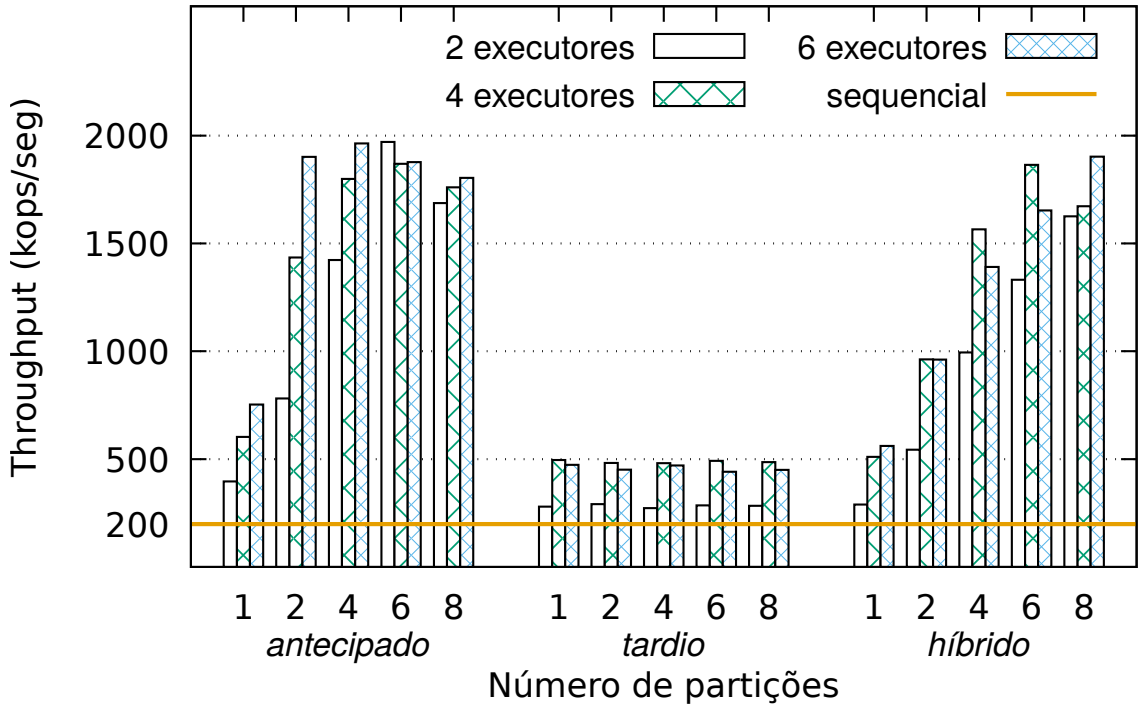


Figura 5.9: Vazão para a carga de trabalho composta somente de leituras e sem operações globais em sistemas configurados com diferentes números de partições e threads executoras (lista de 1k entradas). Para as abordagens antecipada e híbridas, o número de executoras se refere à quantidade de threads executoras por partição, enquanto no escalonamento tardio esse número representa o total de executoras.

um conjunto de réplicas é escolhido para formar um comitê que executa os protocolos de replicação (exemplo, [61, 62]).

Apesar dos protocolos para RMEs BFT e *blockchains* compartilharem este objetivo, existem importantes diferenças entre estas duas abordagens [41]. Primeiramente, enquanto que as RMEs mantêm um *log* de requisições executadas para sincronização de estados, por exemplo após a recuperação de uma réplica, nos *blockchains* este *log* precisa (1) ser escrito em memória estável (disco) para garantir durabilidade, (2) incluir o resultado da execução das requisições/transações para permitir auditoria, e (3) precisa ser auto-verificável por qualquer outra parte. Além disso, outra diferença é que em *blockchains*, ao contrário de RMEs, o conjunto de réplicas geralmente sofre reconfigurações sem o auxílio de partes confiáveis. Finalmente, um trabalho recente [41] demonstrou que utilizar uma RME clássica como um bloco de construção na implementação de *blockchains* resulta em um sistema com limitações de desempenho, principalmente devido a execução sequencial das transações, que envolvem custosas operações criptográficas (i.e., verificação de assinaturas digitais).

Nesta seção, apresentamos um estudo de caso que utiliza uma solução para RMEP na implementação de *blockchains*. Por meio da implementação de um protótipo de sistema de pagamento e a realização de uma série de experimentos, este estudo de caso mostra que, usando RMEs paralelas, a limitação de desempenho descrita em Bessani et al [41] pode ser minimizada, i.e., o desempenho do sistema é aumentado substancialmente. Note que muitos trabalhos recentes focaram na escalabilidade do protocolo de consenso subjacente (exemplo, [63], [64] e [65]), mas no presente estudo estamos particularmente interessados na camada de execução das operações.

Para este estudo de caso, utilizamos um protótipo de sistema de pagamento instantâneo implementado através de moedas digitais que suporta transações para a transferência de valores entre contas de usuários, consulta de saldos, câmbio entre diferentes moedas, dentre outras. Os resultados experimentais mostram que o desempenho do sistema é substancialmente maior através da execução paralela de um percentual destas operações (por exemplo, aquelas envolvendo contas distintas).

5.2.1 Aplicação

ParallelCoin é um serviço simples de carteira digital que gerencia múltiplas moedas digitais baseadas no modelo UTXO (*Unspent Transaction Output*) introduzido no *Bitcoin* [29]. Neste modelo, cada objeto (UTXO) de uma moeda também é chamado de *token* e representa uma certa quantidade dessa moeda possuída por um usuário. Os usuários possuem um par de chaves pública-privada, os *tokens* estão sempre atrelados a uma chave pública e o dono da chave privada correspondente é o único que poderá movimentá-los. Existem quatro tipos de operações suportadas pelo ParallelCoin:

- *Emissão*: as emissões criam um novo *token* para o emissor, na moeda informada e só podem ser realizadas por um grupo seleto de usuários (emissores) previamente configurados.
- *Transferência*: as transferências consomem um conjunto de *tokens* do mesmo usuário e criam outro conjunto de *tokens* na mesma moeda, porém de outros usuários.
- *Câmbio*: os câmbios consomem um conjunto de *tokens* do mesmo usuário e criam outro conjunto de *tokens* em outras moedas para outros usuários.
- *Consulta*: uma consulta retorna os saldo de *tokens* do remetente nas moedas solicitadas.

A operação de consulta é composta pela chave pública do usuário e a lista de moedas nas quais deseja consultar seu saldo. As transferências requerem a chave pública do

remetente, os identificadores dos *tokens* que serão consumidos e um conjunto de pares de chave-valor, cada um contendo uma chave pública de um usuário e a quantidade de moedas que receberá. A operação de câmbio é uma especialização da operação de transferência, sendo que no câmbio, além do par chave-valor, o remetente deve informar a moeda de cada novo *token* que será criado. A taxa de câmbio é configurável para cada par de moedas.

Todas as requisições precisam ser assinadas para garantir sua autenticidade e, assim, comprovar a propriedade dos fundos afetados. Os *tokens* são identificados unicamente por um identificador composto pelo *hash* da operação que os criou e a sua posição na lista dos *tokens* criados nessa operação. O sistema só executará as operações que estejam corretamente formadas e assinadas pelo dono dos *tokens* consumidos ou consultados. Novos *tokens* surgem na saída de cada operação (exceto as consultas) e permanecem ativos até serem usados como entrada de outra operação posterior, quando são consumidos. Cada *token* só pode ser consumido uma única vez, para evitar o problema do gasto-duplo [29]. As réplicas devem garantir essa propriedade.

5.2.2 Modelo de concorrência.

O estado do sistema, mantido pelas réplicas do serviço, é composto pela lista de emissores autorizados e um conjunto de tabelas de *tokens* atribuídos a cada usuários, sendo uma tabela para cada moeda gerenciada. Desta forma, o estado do sistema é dividido em partições, sendo que cada partição armazena as informações relacionadas a uma moeda específica. O serviço do ParallelCoin não precisa se preocupar com problemas de concorrência, uma vez que a camada subjacente de RME paralela só permitirá a execução paralela de operações que não possuem conflitos.

Seguindo o modelo de RME paralela adotado (Seção 4), os clientes indicam quais moedas (partições) serão acessadas em sua requisição e o classificador despacha as requisições para os paralelizadores correspondentes, sendo que existe um paralelizador para cada moeda. Estes paralelizadores inserem as requisições em seus respectivos subgrafos. Quando as requisições que envolvem mais de uma moeda (partição), como o câmbio, os paralelizadores das partições envolvidas inserem as dependências que ligam a nova requisição ao seu subgrafo, mas só um deles insere também a requisição, conforme já descrito. Por fim um conjunto de executores buscam requisições no grafo de dependências para execução.

As operações de emissão acessam apenas uma partição (moeda) e criam os *tokens* correspondentes na conta especificada. Da mesma forma, as operações de transferência também acessam apenas uma partição para consumir (remetente) e criar (destinatário) *tokens*. Como estas duas operações modificam o estado da partição, as mesmas conflitam com qualquer outra operação dentro da mesma partição desde que envolva algum usuário

em comum. Note que transferências entre diferentes usuários não são conflitantes pois alteram diferentes entradas na tabela de *tokens*.

Câmbios são operações globais que acessam mais de uma partição, i.e., as partições relacionadas com as moedas usadas na operação de câmbio. Estas operações alteram o estado e, com isso, conflitam com qualquer outra operação que acessa pelo menos uma das partições e que possuem pelo menos um usuário em comum. Por fim, as consultas funcionam tanto como operações internas em uma partição, em uma única moeda, quanto como operações globais, em mais de uma moeda/partição. Duas consultas nunca conflitam, mas conflitam com qualquer uma das outras operações que alteram o estado da(s) partição(ões) acessada(s), conforme anteriormente descrito.

Algoritmo 11 Definição de conflitos

```
1:  Estrutura da Transação:
2:    Transaction : {
3:      type : {Consulta, Emissao, Transferencia, Cambio},   {tipo de transação}
4:      users : list of public keys,                         {usuários afetados pela transação}
5:      partitions : list of coins,                          {partições acessadas pela transação}
6:      operation : array of bytes,                          {dados da operação a ser realizada pela transação}
7:      signature : array of bytes                           {assinatura digital do remetente}
8:    }

9:  isDependent(Transaction t1, Transaction t2) : boolean
10:   if t1.type = Consulta and t2.type = Consulta then
11:     return false
12:   else
13:     return (t1.users ∩ t2.users ≠ ∅) ∧ (t1.partitions ∩ t2.partitions ≠ ∅)
```

O Algoritmo 11 apresenta a função de definição de conflitos entre duas transações. Esta função deve ser fornecida para a camada de RME paralela, possibilitando que a mesma controle os acessos ao estado da aplicação. A estrutura $Transaction$ possui todos os dados relacionados com a transação, como o tipo de transação, os usuários afetados pela transação, as partições/moedas que a transação acessa, os dados da operação a ser realizada e uma assinatura digital para garantir a autenticidade da transação. A função $isDependent(t_1, t_2)$ recebe duas transações como parâmetro e retornará verdadeiro caso elas conflitem, caso contrário retornará falso.

5.2.3 Implementação

O ParallelCoin foi implementado em JAVA, replicado usando o BFT-SMaRt [55] e o algoritmo de escalonamento híbrido descrito no Capítulo 4. Os clientes geram operações/transações assinadas, rotulam suas operações com a classe adequada e as en-

viam para o protocolo de multicast atômico do BFT-SMaRt. Este protocolo executa um consenso bizantino para ordenar um lote de operações por vez. Portanto, cada réplica entrega os mesmos lotes de requisições na mesma ordem total. Todas as implementações desenvolvidas estão livremente disponíveis no seguinte repositório: `https://github.com/aldenioburgos/library/tree/hibrid`.

5.2.4 Ambiente experimental

Esta seção apresenta os experimentos realizados com o objetivo de analisar o desempenho do ParallelCoin. Para fins de comparação, também implementamos uma versão sequencial do sistema, que é replicada da forma tradicional sobre o BFT-SMaRt, i.e., todas as operações são executadas de forma sequencial.

O ambiente experimental foi configurado com 8 máquinas conectadas a uma rede comutada de 10Gbps no Emulab [66]. As máquinas foram configuradas com o sistema operacional Ubuntu Linux 20.04 e uma máquina virtual Java de 64 bits versão 15.0.2. O Bft-SMaRt foi configurado com 4 réplicas hospedadas em máquinas separadas (Dell PowerEdge R820 equipados com quatro processadores Intel Xeon E5-4620 com 8 núcleos e 16 threads executando a 2,2 GHz com 128 GB de RAM) para tolerar a falha bizantina de até uma réplica. Adicionalmente, 2400 clientes foram distribuídos uniformemente em outras 4 máquinas (Dell PowerEdge R430 equipados com dois processadores Intel E5-2630v3 de 8 núcleos e 16 threads executando a 2,4 GHz com 64 GB de RAM).

5.2.5 Cargas de trabalho e configurações

Antes de cada experimento, o sistema foi pré-carregado com 2400 usuários (chaves públicas), e cada usuário com 10 *tokens* de valor 1 em cada uma das moedas. A taxa de câmbio entre as moedas foi configurada como 1 para todos os pares de moedas. Nos testes de desempenho, após cada transferência ou câmbio o sistema retorna para seu estado inicial, assim os *tokens* consumidos voltam para o estado ativo e os novos *tokens* gerados são descartados. Essa configuração permitiu que cada *thread* cliente executasse um número potencialmente ilimitado de requisições só com seus 10 *tokens* iniciais. O algoritmo de *hash* utilizado foi o *SHA256*, enquanto que o algoritmo de assinatura digital escolhido foi o algoritmo de assinatura digital de curvas elípticas (*ECDSA*) com o algoritmo de *hash* *SHA256*.

Dois conjuntos de experimentos foram executados, e os resultados da média da vazão obtida nos servidores são reportados nas figuras 5.10 e 5.11. O primeiro conjunto é composto por 90% de consultas e 10% de transferências, sendo que todas as operações envolvem apenas uma partição (moeda) por vez. No segundo, temos 90% de consultas e

10% de movimentações (transferências ou câmbios), sendo que 5% das movimentações são câmbios envolvendo duas moedas e o mesmo percentual das consultas também envolve duas moedas. A cada requisição, a *thread* cliente sorteia (seguindo uma distribuição uniforme) o tipo da operação, o usuário receptor e as moedas envolvidas, cria uma nova operação e assina digitalmente antes de enviar para o servidor. Todas as movimentações envolveram exatamente dois usuários, o pagador e o recebedor. Conforme já comentado, as operações envolveram no máximo duas moedas.

5.2.6 Resultados

A Figura 5.10 apresenta a vazão obtida tanto pela execução sequencial quanto por diversas combinações de número de partições/moedas e *threads* executores na solução que permite execuções paralelas. Perceba que na configuração com apenas uma partição e um executor por partição também temos uma execução sequencial, enquanto que em todos os demais casos temos execuções paralelas. É possível identificar com clareza que o desempenho da RME paralela supera por uma larga margem o da RME sequencial. Enquanto que a vazão na RME sequencial manteve-se em aproximadamente $4k$ operações por segundo, a solução paralela escalou com o número de partições e conseguiu atingir um valor superior a $55k$ operações por segundo na configuração com 8 partições e 4 executores por partição (32 no total). De fato, as soluções para RME paralela conseguem aumentar o desempenho na medida que a carga de trabalho oferece mais oportunidade de paralelismo (por exemplo, aumentando o número de partições/moedas) ou mais *threads* são disponibilizadas para executar a mesma carga de trabalho.

A Figura 5.11 mostra resultados similares para a segunda carga de trabalho. Como esta carga de trabalho contém operações que acessam mais de uma partição, não faz sentido executar o caso com apenas uma partição, que é idêntico ao apresentado na Figura 5.10. O desempenho da RME sequencial continua limitado pela velocidade de processamento de uma única *thread* e novamente manteve-se próximo de $4k$ operações por segundo. Por outro lado, a implementação com RME paralela apresentou um desempenho largamente superior, aproximando-se de $30k$ operações por segundo em algumas configurações.

5.3 Considerações finais

Os experimentos apresentados neste capítulo demonstraram a superioridade do desempenho da abordagem híbrida sobre suas antecessoras em diversas cargas de trabalho. Também podemos perceber que as estruturas adicionais do escalonamento híbrido não causam aumentos significativos na latência do sistema. Além disso, por meio de uma aplicação de moeda digital, Bessani et al [41] mostrou que quando usamos RMEs tradicionais como um

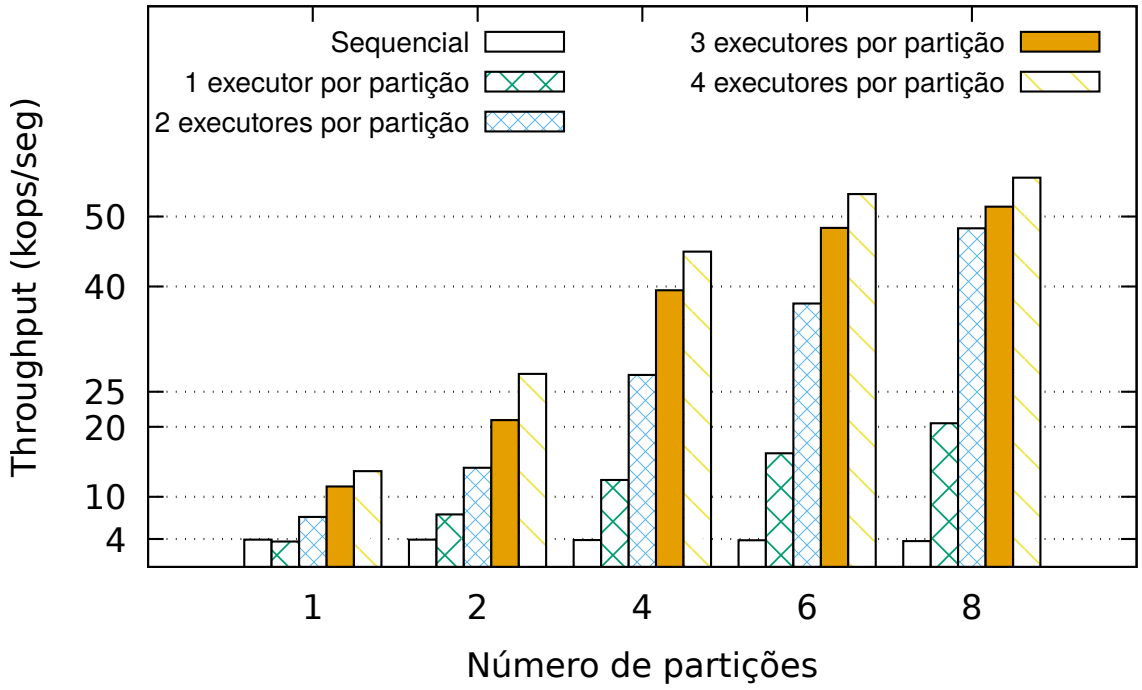


Figura 5.10: Vazão para diferentes números de partições e executores para uma carga de trabalho com 90% de consultas e 10% de transferências.

bloco de construção na implementação de blockchains, seu modelo de execução sequencial afeta significativamente o desempenho do sistema. O estudo de caso aqui apresentado utilizou uma aplicação de moeda digital, chamada de ParallelCoin, para demostrar que RMEs paralelas conseguem minimizar as limitações de desempenho anteriormente identificadas, pois permitem que uma parte das transações seja executada em paralelo nas réplicas, seguindo um modelo de concorrência elaborado para as operações definidas na aplicação. Experimentos realizados com as implementações desenvolvidas mostram que o ParallelCoin possui um desempenho de até 13.75× o da solução sequencial, quando consideramos cargas de trabalho com operações que acessam apenas uma partição/moeda. Já para cargas de trabalho com operações endereçadas a mais de uma partição/moeda, o desempenho do ParallelCoin é até 7× maior do que a solução com apenas execuções sequenciais.

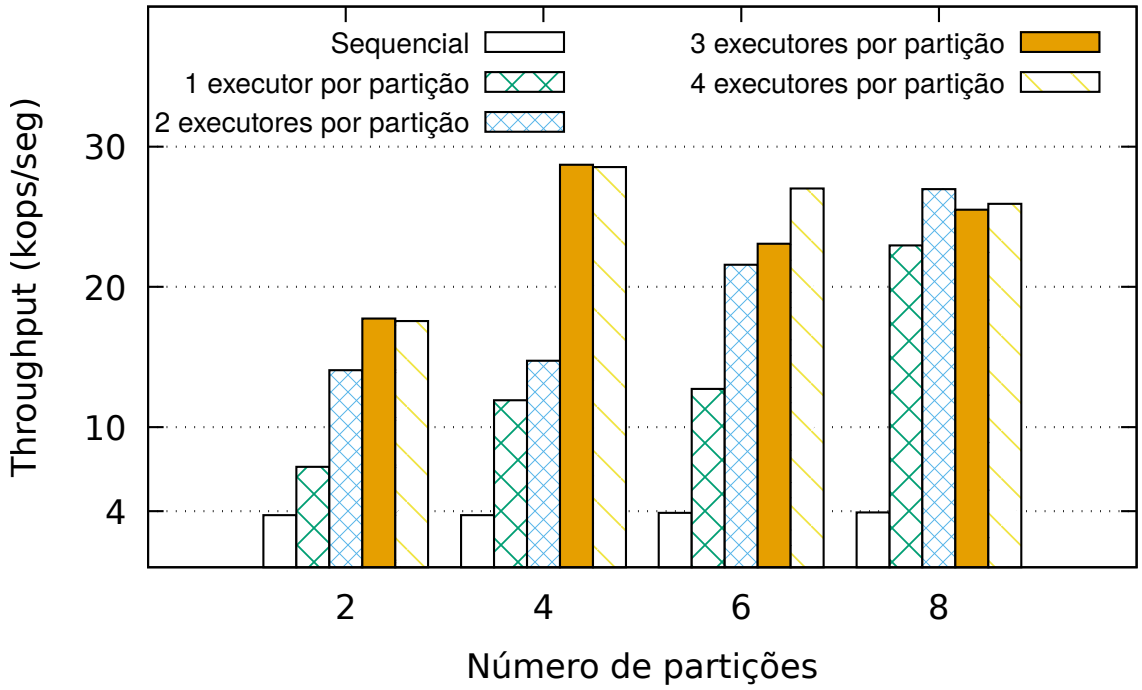


Figura 5.11: Vazão para diferentes números de partições e executores para uma carga de trabalho com 90% de consultas e 10% de movimentações (transferências/câmbios), sendo que 5% de todas as operações envolvem duas moedas/partições e 95% só uma moeda/partição.

Capítulo 6

Conclusão e Trabalhos Futuros

O presente capítulo conclui esta dissertação. Primeiramente, uma visão geral sobre o trabalho é apresentada. Após isso, os objetivos desta dissertação são relembrados e as contribuições deste trabalho são abordadas. Por fim, algumas perspectivas de trabalhos futuros são expostas.

6.1 Visão Geral do Trabalho

Nesse trabalho foi desenvolvido e apresentado o H-SMR, um protocolo de escalonamento híbrido para Replicação Máquina de Estados Paralela (RMEP), inspirado na combinação de dois protocolos de RMEP anteriores. Foi demonstrado em ambiente experimental que o H-SMR superou em desempenho a RME clássica em praticamente todos os cenários estudados e a seus dois antecessores, com grande margem, em sistemas multi-particionados com cargas de trabalho mistas (com leituras e escritas), empatando com um ou com outro em alguns cenários específicos. O Capítulo 1 fez uma contextualização do tema, com a descrição do problema e a relevância do trabalho proposto, listando também os objetivos almejados. O Capítulo 2 revisou conceitos necessários para a compreensão do protocolo H-SMR. O Capítulo 3 descreveu os protocolos considerados como o estado da arte em Replicação Máquina de Estados Paralela, e de maneira mais aprofundada os protocolos com escalonamento antecipado e escalonamento tardio que serviram de base para a idealização do escalonamento híbrido. O Capítulo 4 detalhou o H-SMR, nossa proposta de Replicação Máquina de Estados Paralela com escalonamento híbrido, expondo seu modelo de sistema e algoritmos. O Capítulo 5 apresentou os resultado dos testes experimentais, comparando a estratégia proposta de escalonamento com o escalonamento tardio e antecipado, bem como com o RME clássico, considerando diversas configurações e cargas de trabalho. Ainda no mesmo capítulo foi realizado um estudo de caso com um protótipo de sistema de pagamentos (moeda virtual), por meio do qual foi possível

demonstrar, em ambiente experimental, que uma RMEP consegue minimizar as limitações de desempenho identificadas em Bessani et al [41]. Por fim, o capítulo atual conclui o trabalho elencando suas principais contribuições e apontando a direção para trabalhos futuros.

6.2 Revisão dos Objetivos e Contribuições da Dissertação

Os objetivos desta dissertação, enunciados na Seção 1.1, são aqui relembrados, juntamente com a indicação do capítulo que apresenta o estudo que visa cumprir cada objetivo. Além disso, as principais contribuições desta dissertação, que vão ao encontro do cumprimento destes objetivos, são abordadas.

1. **Proposta de um protocolo para escalonamento híbrido em RMEP, visando remover as limitações impostas pelas soluções anteriores de escalonamento antecipado ou tardio.**

   O Capítulo 4 discute o projeto e desenvolvimento do H-SMR, um protocolo de Replicação Máquina de Estados Paralela com escalonamento híbrido, que se sobressaiu as soluções de escalonamento antecipado ou tardio, nos teste experimentais, em condições onde estas soluções manifestam suas limitações.

2. **Implementação deste protocolo e integração com uma biblioteca de RME.**

   O protocolo H-SMR foi implementado, conforme o Capítulo 4, e posteriormente integrado a biblioteca de RME de código aberto Bft-SMaRt [55, 56]. Os fontes do H-SMR, já integrados com o Bft-SMaRt, estão disponíveis em `https://github.com/aldenioburgos/library/tree/hibrid`.

3. **Implementação de uma aplicação (para *microbenchmark*) e análise de desempenho da solução proposta, comparando-a com o estado da arte dos protocolos antecipado e tardio.**

   Foram implementadas e utilizadas para *microbenchmark* duas aplicações de lista encadeada com operações para verificar se uma entrada i (ou seja, um inteiro) está na lista (*contains*) e para incluir uma entrada j na lista (*add*), representando um serviço de leitores e escritores. Uma das aplicações tratou com uma lista única (sistema não particionado) e a outra com múltiplas listas separadas (sistema multi-particionado). Essas aplicações encontram-se descritas na Seção 5.1.3. As seções de

5.1.4 a 5.1.7 apresentaram os resultados, e as análises sobre os resultados, dos experimentos realizados com os protocolos de RME clássica e RMEP com escalonamento antecipado, tardio e híbrido.

4. **Realizar um estudo de caso a fim de analisar como a solução para RMEP proposta poderia melhorar uma limitação de desempenho de *blockchains* privadas**.

   Também no Capítulo 5 foi realizado um estudo de caso onde o H-SMR foi aplicado ao ParallelCoin, um protótipo de sistema de pagamentos multi-moedas, inspirado no SMaRtCoin [41], que é uma aplicação de moeda digital. Os experimentos realizados com o ParallelCoin mostraram que o uso do H-SMR no lugar de uma RME clássica como bloco de construção na implementação de um *blockchain*, como o SMaRtChain, pode atenuar suas limitações de desempenho, quando usado em aplicações como o SMaRtCoin e o ParallelCoin, devido ao aumento na paralelização de suas custosas operações criptográficas (i.e., verificação de assinaturas digitais).

Uma parte das contribuições listadas acima foram publicadas em [67, 68, 69].

6.3 Trabalhos Futuros

Os estudos realizados nesta dissertação certamente não finalizam a discussão sobre a Replicação Máquina de Estados Paralela. Além do mais, este assunto é muito amplo e dificilmente as opções de pesquisa que o envolvem se esgotarão. Deste modo, alguns trabalhos relacionados com esta dissertação poderão ser explorados no futuro.

O primeiro destes trabalhos é a realização de testes mais amplos, onde análises mais significativas a respeito dos protocolos apresentados nesta dissertação seriam apuradas. Os experimentos apresentados no Capítulo 5 poderiam ser executados em um *testbed* melhor, com um número maior de computadores, o que possibilitaria uma análise mais precisa do desempenho e escalabilidade dos protocolos.

Outro trabalho interessante que não foi explorado nesta dissertação é a realização de estudos no sentido de incorporar características do escalonamento estático (Seção 3.2.4) ao H-SMR, afim de avaliar a escalabilidade do protocolo resultante. Neste caso, o classificador seria removido do servidor e vários grupos de multicast atômico seriam utilizados na ordenação das requisições.

Referências

[1] Schneider, Fred B: *Implementing fault-tolerant services using the state machine approach: A tutorial*. ACM Computing Surveys (CSUR), 22(4):299–319, 1990. 1, 2, 11, 12, 18, 28, 42, 43

[2] Lamport, Leslie: *Time, clocks, and the ordering of events in a distributed system*. Communications of the ACM, 21(7):558–565, 1978. 1, 8, 42

[3] Herlihy, Maurice P e Jeannette M Wing: *Linearizability: A correctness condition for concurrent objects*. ACM Transactions on Programming Languages and Systems (TOPLAS), 12(3):463–492, 1990. 1, 12, 42, 45

[4] Burrows, Mike: *The chubby lock service for loosely-coupled distributed systems*. Em *Proceedings of the 7th symposium on Operating systems design and implementation*, páginas 335–350, 2006. 1, 42

[5] Hunt, Patrick, Mahadev Konar, Flavio Paiva Junqueira e Benjamin Reed: *Zookeeper: Wait-free coordination for internet-scale systems*. Em *USENIX annual technical conference*, volume 8, 2010. 1, 42

[6] Kapritsos, Manos, Yang Wang, Vivien Quema, Allen Clement, Lorenzo Alvisi e Mike Dahlin: *All about eve: execute-verify replication for multi-core servers*. Em *Presented as part of the 10th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 12)*, páginas 237–250, 2012. 2, 13, 39, 42

[7] Budhiraja, Navin, Keith Marzullo, Fred B Schneider e Sam Toueg: *The primary-backup approach*. Distributed systems, 2:199–216, 1993. 2, 11

[8] Guo, Zhenyu, Chuntao Hong, Mao Yang, Dong Zhou, Lidong Zhou e Li Zhuang: *Rex: Replication at the speed of multi-core*. Em *Proceedings of the Ninth European Conference on Computer Systems*, páginas 1–14, 2014. 2, 37, 38

[9] Cui, Heming, Rui Gu, Cheng Liu, Tianyu Chen e Junfeng Yang: *Paxos made transparent*. Em *Proceedings of the 25th Symposium on Operating Systems Principles*, páginas 105–120, 2015. 2, 37, 38

[10] Kotla, Ramakrishna e Michael Dahlin: *High throughput byzantine fault tolerance*. Em *International Conference on Dependable Systems and Networks, 2004*, páginas 575–584. IEEE, 2004. 2, 13, 18, 19, 20, 21, 42

[11] Marandi, Parisa Jalili, Carlos Eduardo Bezerra e Fernando Pedone: *Rethinking state-machine replication for parallelism.* Em *2014 IEEE 34th International Conference on Distributed Computing Systems*, páginas 368–377. IEEE, 2014. 2, 12, 13, 36, 42

[12] Marandi, Parisa Jalili e Fernando Pedone: *Optimistic parallel state-machine replication.* Em *2014 IEEE 33rd International Symposium on Reliable Distributed Systems*, páginas 57–66. IEEE, 2014. 2, 13, 39, 40, 42

[13] Alchieri, Eduardo, Fernando Dotti, Odorico M Mendizabal e Fernando Pedone: *Reconfiguring parallel state machine replication.* Em *2017 IEEE 36th Symposium on Reliable Distributed Systems (SRDS)*, páginas 104–113. IEEE, 2017. 2, 28

[14] Escobar, Ian Aragon, Eduardo Alchieri, Fernando Luís Dotti e Fernando Pedone: *Boosting concurrency in parallel state machine replication.* Em *Proceedings of the 20th International Middleware Conference*, páginas 228–240, 2019. 2, 3, 20, 21, 22, 42, 43, 57, 65

[15] Alchieri, Eduardo, Fernando Dotti e Fernando Pedone: *Early scheduling in parallel state machine replication.* Em *Proceedings of the ACM Symposium on Cloud Computing*, páginas 82–94, 2018. 2, 3, 13, 28, 30, 42, 43, 65

[16] Batista, Elia, Eduardo Alchieri, Fernando Dotti e Fernando Pedone: *Resource utilization analysis of early scheduling in parallel state machine replication.* Em *2019 9th Latin-American Symposium on Dependable Computing (LADC)*, páginas 1–10. IEEE, 2019. 3, 43

[17] Avizienis, Algirdas, J C Laprie, Brian Randell e Carl Landwehr: *Basic concepts and taxonomy of dependable and secure computing.* IEEE transactions on dependable and secure computing, 1(1):11–33, 2004. 5, 6

[18] Tanenbaum, AS e H Bos: *Sistemas operacionais modernos. 4a edição*, 2016. 5

[19] Coulouris, George F, Jean Dollimore e Tim Kindberg: *Distributed systems: concepts and design.* pearson education, 2005. 5, 6, 7, 8, 9, 10

[20] Hadzilacos, Vassos e Sam Toueg: *A modular approach to fault-tolerant broadcasts and related problems.* Relatório Técnico, Cornell University, 1994. 7

[21] Défago, Xavier, André Schiper e Péter Urbán: *Total order broadcast and multicast algorithms: Taxonomy and survey.* ACM Computing Surveys (CSUR), 36(4):372–421, 2004. 8, 45

[22] Dwork, Cynthia, Nancy Lynch e Larry Stockmeyer: *Consensus in the presence of partial synchrony.* Journal of the ACM (JACM), 35(2):288–323, 1988. 9

[23] Défago, Xavier: *Agreement-related problems: From semi-passive replication to totally ordered broadcast.* 2000. 10, 11

[24] Schneider, Fred B: *Replication management using the state-machine approach, distributed systems.* 1993. 11

[25] Poledna, Stefan: *Replica determinism in distributed real-time systems: A brief survey.* Real-Time Systems, 6(3):289–316, 1994. 11

[26] Pedone, Fernando e André Schiper: *Generic broadcast.* Em *International Symposium on Distributed Computing*, páginas 94–106. Springer, 1999. 13

[27] Toueg, Sam: *Thrifty generic broadcast".* Em *Distributed Computing: 14th International Conference, DISC 2000 Toledo, Spain, October 4-6, 2000 Proceedings*, página 268. Springer, 2003. 13

[28] Lamport, Leslie: *Generalized consensus and paxos.* 2005. 13

[29] Nakamoto, Satoshi e A Bitcoin: *A peer-to-peer electronic cash system.* Bitcoin.–URL: https://bitcoin. org/bitcoin. pdf, 2008. 13, 14, 15, 16, 69, 70

[30] Bashir, Imran: *Mastering blockchain.* Packt Publishing Ltd, 2017. 14

[31] *Global bitcoin nodes distribution.* `https://bitnodes.earn.com/dashboard/?days=730`. 14

[32] Wood, Gavin *et al.*: *Ethereum: A secure decentralised generalised transaction ledger.* Ethereum project yellow paper, 151(2014):1–32, 2014. 15

[33] Baliga, Arati, I Subhod, Pandurang Kamat e Siddhartha Chatterjee: *Performance evaluation of the quorum blockchain platform.* arXiv preprint arXiv:1809.03421, 2018. 15

[34] Greenspan, Gideon: *Multichain private blockchain-white paper.* URl: http://www. multichain. com/download/MultiChain-White-Paper. pdf, 2015. 15

[35] Cachin, Christian *et al.*: *Architecture of the hyperledger blockchain fabric.* Em *Workshop on distributed cryptocurrencies and consensus ledgers*, volume 310, 2016. 15

[36] Buchman, Ethan: *Tendermint: Byzantine fault tolerance in the age of blockchains.* Tese de Doutoramento, 2016. 15

[37] Payment, Committee on e Settlement Systems: *A glossary of terms used in payments and settlement systems.* 2003. 15

[38] Burgos, A e B Batavia: *O meio circulante na era digital.* Banco Central do Brasil, 2018. 15

[39] Lamport, Leslie: *The part-time parliament.* ACM Transactions on Computer Systems (TOCS), 16(2):133–169, 1998. 16, 38

[40] Lamport, Leslie *et al.*: *Paxos made simple.* ACM Sigact News, 32(4):18–25, 2001. 16, 38

[41] Bessani, Alysson, Eduardo Alchieri, João Sousa, André Oliveira e Fernando Pedone: *From byzantine replication to blockchain: Consensus is only the beginning.* arXiv preprint arXiv:2004.14527, 2020. 16, 57, 68, 69, 73, 77, 78

[42] Van Renesse, Robbert, Nicolas Schiper e Fred B Schneider: *Vive la différence: Paxos vs. viewstamped replication vs. zab*. IEEE Transactions on Dependable and Secure Computing, 12(4):472–484, 2014. 18, 28

[43] Herlihy, Maurice: *A methodology for implementing highly concurrent data structures*. Em *Proceedings of the second ACM SIGPLAN symposium on Principles & practice of parallel programming*, páginas 197–206, 1990. 23

[44] Alchieri, Eduardo, Fernando Dotti, Parisa Marandi, Odorico Mendizabal e Fernando Pedone: *Boosting state machine replication with concurrent execution*. Em *2018 Eighth Latin-American Symposium on Dependable Computing (LADC)*, páginas 77–86. IEEE, 2018. 36

[45] Basile, Claudio, Zbigniew Kalbarczyk e Ravishankar K Iyer: *Active replication of multithreaded applications*. IEEE transactions on parallel and distributed systems, 17(5):448–465, 2006. 37

[46] Reiser, Hans P, Jorg Domaschka, Franz J Hauck, Rudiger Kapitza e Wolfgang Schroder-Preikschat: *Consistent replication of multithreaded distributed objects*. Em *2006 25th IEEE Symposium on Reliable Distributed Systems (SRDS'06)*, páginas 257–266. IEEE, 2006. 38

[47] Olszewski, Marek, Jason Ansel e Saman Amarasinghe: *Kendo: efficient deterministic multithreading in software*. Em *Proceedings of the 14th international conference on Architectural support for programming languages and operating systems*, páginas 97–108, 2009. 38

[48] Kapitza, Rüdiger, Matthias Schunter, Christian Cachin, Klaus Stengel e Tobias Distler: *Storyboard: Optimistic deterministic multithreading*. 2010. 39, 40

[49] Aksoy, Remzi Can e Manos Kapritsos: *Aegean: Replication beyond the client-server model*. Em *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, páginas 385–398, 2019. 40

[50] Attiya, Hagit e Jennifer Welch: *Distributed computing: fundamentals, simulations, and advanced topics*, volume 19. John Wiley & Sons, 2004. 42

[51] Mendizabal, Odorico M, Rudá ST De Moura, Fernando Luís Dotti e Fernando Pedone: *Efficient and deterministic scheduling for parallel state machine replication*. Em *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, páginas 748–757. IEEE, 2017. 42

[52] Hadzilacos, Vassos e Sam Toueg: *Fault-tolerant broadcasts and related problems*. Em *Distributed systems (2nd Ed.)*, páginas 97–145. 1993. 45

[53] Chandra, Tushar Deepak e Sam Toueg: *Unreliable failure detectors for reliable distributed systems*. Journal of the ACM (JACM), 43(2):225–267, 1996. 45

[54] Fischer, Michael J, Nancy A Lynch e Michael S Paterson: *Impossibility of distributed consensus with one faulty process*. Journal of the ACM (JACM), 32(2):374–382, 1985. 45

[55] Bessani, Alysson, João Sousa e Eduardo EP Alchieri: *State machine replication for the masses with bft-smart*. Em *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, páginas 355–362. IEEE, 2014. 57, 58, 65, 71, 77

[56] *Bftsmart*. `http://bft-smart.github.io/library/`. 57, 77

[57] Maffione, Vincenzo, Giuseppe Lettieri e Luigi Rizzo: *Cache-aware design of general-purpose single-producer-single-consumer queues: Cache-aware single-producer-single-consumer queues*. Software: Practice and Experience, 49, dezembro 2018. 58

[58] Castro, Miguel e Barbara Liskov: *Practical Byzantine fault tolerance*. Em *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation*, páginas 173–186. USENIX Association, fevereiro 1999, ISBN 1-880446-39-1. 67

[59] Castro, Miguel e Barbara Liskov: *Practical byzantine fault tolerance and proactive recovery*. ACM Transactions on Computer Systems (TOCS), 20(4):398–461, 2002. 67

[60] Cachin, Christian e Marko Vukolić: *Blockchain consensus protocols in the wild*. arXiv preprint arXiv:1707.01873, 2017. 67

[61] Pass, Rafael e Elaine Shi: *Hybrid consensus: Efficient consensus in the permissionless model*. Em *31st International Symposium on Distributed Computing (DISC 2017)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017. 68

[62] Abraham, Ittai, Dahlia Malkhi, Kartik Nayak, Ling Ren e Alexander Spiegelman: *Solida: A blockchain protocol based on reconfigurable byzantine consensus*. arXiv preprint arXiv:1612.02916, 2016. 68

[63] Gueta, Guy Golan, Ittai Abraham, Shelly Grossman, Dahlia Malkhi, Benny Pinkas, Michael Reiter, Dragos Adrian Seredinschi, Orr Tamir e Alin Tomescu: *Sbft: a scalable and decentralized trust infrastructure*. Em *2019 49th Annual IEEE/IFIP international conference on dependable systems and networks (DSN)*, páginas 568–580. IEEE, 2019. 69

[64] Liu, Jian, Wenting Li, Ghassan O Karame e N Asokan: *Scalable byzantine consensus via hardware-assisted secret sharing*. IEEE Transactions on Computers, 68(1):139–151, 2019. 69

[65] Guerraoui, Rachid, Jad Hamza, Dragos Adrian Seredinschi e Marko Vukolic: *Can 100 machines agree?* arXiv preprint arXiv:1911.07966, 2019. 69

[66] White, Brian, 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*. Em *Symposium on Operating Systems Design and Implementation*, 2002. 72

[67] Burgos, Aldênio, Eduardo Alchieri, Fernando Dotti e Fernando Pedone: *Replicação máquina de estados paralelas com escalonamento híbrido.* Em *Anais do XXXIX Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos*, páginas 476–489. SBC, 2021. 78

[68] Burgos, Aldênio e Eduardo Alchieri: *Um estudo sobre o uso de replicação máquina de estados paralelas na implementação de blockchains.* Em *Anais do XXII Workshop de Testes e Tolerância a Falhas*, páginas 57–70. SBC, 2021. 78

[69] Burgos, Aldênio, Eduardo Alchieri e Fernando Dotti: *On the performance of using parallel statemachine replication to implement blockchains.* Em *2021 X Latin-American Symposium on Dependable Computing (LADC)*. IEEE, 2021. 78