UNIVERSIDADE FEDERAL DO RIO GRANDE - FURG
CENTRO DE CIÊNCIAS COMPUTACIONAIS
CURSO DE ENGENHARIA DE COMPUTAÇÃO

Projeto de Graduação em Engenharia de Computação

Log Escalável e Fracamente Acoplado para Recuperação em Replicação Máquina de Estados

Luiz Gustavo C. Xavier

Orientador: Prof. Dr. Odorico Machado Mendizabal

Rio Grande, 2019

Banca examinadora:

Prof. Dr. Pedro de Botelho Marcos

Prof. Dr. Rodrigo Andrade de Bem

Simplicity is prerequisite for reliability.
— EDSGER W. DIJKSTRA

RESUMO

XAVIER, Luiz Gustavo C.. ***Log* Escalável e Fracamente Acoplado para Recuperação em Replicação Máquina de Estados**. 2019. 52 f. Projeto de Graduação – Engenharia de Computação. Universidade Federal do Rio Grande - FURG, Rio Grande.

A técnica de Replicação Máquina de Estados (SMR) é amplamente utilizada para fornecer tolerância a falhas em aplicações distribuídas. O suporte para implementações SMR em infraestruturas compartilhadas tem sido difundido na literatura, popularizando sua adoção. No entanto, ainda existem aspectos não triviais que desenvolvedores precisam lidar para implementar serviços confiáveis. Neste trabalho, foi identificada a necessidade de protocolos de recuperação para proporcionar uma maior disponibilidade de serviços replicados, e proposta uma abordagem para simplificar o desenvolvimento do registro de comandos em arquivos *log*, facilitando o compartilhamento de recursos em infraestruturas compartilhadas e aliviando a sobrecarga de registro em *log* das réplicas em execução. A ideia central é desacoplar a execução do serviço do *log* e oferecer suas funcionalidades como um serviço acoplável à implementações de SMR. Além da simplicidade no desenvolvimento de SMR, experimentos demonstram que tal abordagem não penaliza o desempenho dos serviços replicados e que um serviço de *log* pode ser escalado para atender várias aplicações concorrentes.

Palavras-chave: Sistemas Distribuídos, Tolerância à Falhas, Replicação Máquina de Estados, Recuperação por *Log*.

ABSTRACT

XAVIER, Luiz Gustavo C.. **Scalable and Decoupled Log for Recovery in State Machine Replication**. 2019. 52 f. Projeto de Graduação – Engenharia de Computação. Universidade Federal do Rio Grande - FURG, Rio Grande.

State Machine Replication (SMR) is a widely used technique to provide fault tolerance in distributed applications. Particularly, the development of SMR on shared infrastructures has received considerable attention from both researchers and practicioners. However, there are still non-trivial aspects that developers have to handle to build and deploy their dependable services. In this document, it is identified the needs for recovery to keep fault-tolerance levels, and proposed an approach to simplify the development of logging, improving resource sharing in shared infrastructures, and alleviating replica's logging overhead. The central idea is to decouple service execution from logging and offer logging functionality as a service attachable to SMR deployments. Beyond the added simplicity to deploy an SMR application, experiments show that this approach does not penalize performance of replicated services, and that a logging service can scale to look to several concurrent applications.

Keywords: Distributed Systems, Fault Tolerance, State Machine Replication, Log Recovery.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

BFT	*Byzantine Fault Tolerance* – Tolerância a Falhas Bizantinas
CFT	*Crash Fault Tolerance* – Tolerância a Falhas por Colapso
DAG	*Directed Acyclic Graph* – Grafo Acíclico Dirigido
I/O	*Input/Output* – Entrada e Saída
MTBF	*Mean Time Between Failures* – Tempo Médio entre Falhas
P-SMR	*Parallel State Machine Replication* – Replicação Máquina de Estados Paralela
RPC	*Remote Procedure Call* – Chamada de Procedimento Remoto
SMR	*State Machine Replication* – Replicação Máquina de Estados

SUMÁRIO

1 INTRODUÇÃO

No mundo globalizado do século XXI é difícil negar a afirmação popular de que "Estamos todos conectados". Sistemas distribuídos integram o cotidiano de milhões de pessoas em suas tarefas mais rotineiras, estando exemplificados em aplicações de comércio eletrônico, conteúdo educativo *online* e entretenimento (Coulouris et al., 2011). Sistemas críticos representam uma categoria especial de aplicações distribuída, em que a maximização da disponibilidade é tida como principal objetivo, uma vez que a ocorrência de qualquer intervalo de indisponibilidade pode comprometer significativamente a qualidade do serviço. Neste sentido, torna-se crucial o desenvolvimento de sistemas capazes de prover segurança no funcionamento em ambientes providos de falhas[1]. Ao considerar as mais diversas aplicações em ambientes virtualizados mapeadas atualmente (JoSEP et al., 2010), torna-se necessário adequar soluções conhecidas na literatura para o ambiente compartilhado, onde há maior escassez de recursos, alto grau de concorrência, e preocupações maiores com a segurança. Apesar de algumas demandas de confiabilidade serem sutilmente satisfeitas por parte dos provedores de infraestruturas descentralizadas, abstraindo-se detalhes minuciosos de implementações à desenvolvedores desta classe de aplicações (Chaczko et al., 2011), é importante ressaltar a relevância de contribuições que estudem confiabilidade em ambiente compartilhado, considerando o extenso número de falhas identificadas em tais ambientes (Jhawar and Piuri, 2017).

A técnica de replicação é conhecida como uma importante estratégia para prover tolerância a falhas em sistemas distribuídos, de modo a aumentar a disponibilidade dos serviços. Mais especificamente, a abordagem de replicação ativa por máquina de estados (SMR, do inglês *State Machine Replication*) (Lamport, 1978; Schneider, 1990) permite ao sistema alcançar intervalos ainda maiores de disponibilidade quando comparada à replicação por primário/*backup* (Budhiraja et al., 1993), devido a não evidenciar a presença de períodos de indisponibilidade na ocorrência de falhas do nodo primário do sistema. Porém, ambas as abordagens requerem a implementação de mecanismos de recuperação, capazes de resgatar a réplica ao seu estado de execução normal após

[1] Os termos *fault*, *error* e *failure* são frequentemente traduzidos de formas diferentes na língua portuguesa. Portanto, para esclarecimento da nomenclatura utilizada neste trabalho, tais termos são traduzidos como *falha*, *erro* e *defeito*, respectivamente.

a ocorrência de falhas. A recuperação por *logging* consiste no registro de comandos executados pela aplicação em arquivos persistentes, para serem inteiramente reprocessados a fim de alcançar o estado mais atual de funcionamento. Há também a maneira de recuperação por *checkpoints*, que tem por objetivo minimizar o custo da recuperação com processamento de extensos arquivos de *logs*. Tal técnica produz estágios de recuperação representando estados consistentes do sistema . Com a realização efetiva de cada *checkpoint* é permitida a eliminação de comandos prévios a este estado no *log* de recuperação. Na prática, as estratégias de *logging* e *checkpointing* são combinadas para o desenvolvimento de protocolos de recuperação. A primeira é menos custosa para a réplica em execução, porém o crescimento indefinido do *log* eventualmente levaria a uma escassez de recursos. Com a segunda estratégia, o *log* pode ser encurtado, mas a criação de um *checkpoint* causa períodos de queda na vazão do serviço. Como forma de minimizar custos com estratégias de *logging* e *checkpointing*, alguns trabalhos apresentam alternativas para realizar o processo de recuperação de maneiras menos intrusivas, minimizando a sobrecarga na execução normal da aplicação (Clement et al., 2009; Bessani et al., 2013), ou permitindo que uma réplica em recuperação possa atender parcialmente novas requisições (Mendizabal et al., 2017b).

1.1 Justificativa

Aplicações de alta vazão demandam alta disponibilidade do serviço, uma vez que curtos períodos de indisponibilidade podem comprometer negativamente com a qualidade do serviço. Como no ano de 2016, em que foi registrado durante o *Amazon Prime Day* uma vazão média de 636 itens sendo comercializados por segundo nos servidores da empresa multi varejista Amazon (Popomaronis, 2016). Adotando-se um preço médio de aproximadamente 30 dólares por item, é possível concluir diretamente que um mero segundo de indisponibilidade neste intervalo representaria a empresa um prejuízo na ordem de 19.080 dólares.

Uma maneira prática de elevar níveis de disponibilidade para esta classe de aplicações é com a implementação de técnicas de tolerância a falhas, de modo a prover maior segurança no funcionamento correto. Porém, deve-se ressaltar a importância na implementação de técnicas menos intrusivas, uma vez que uma degradação na vazão observada em decorrência de tais abordagens também compromete diretamente com o desempenho, reduzindo o faturamento no caso da aplicação mencionada.

1.2 Objetivos

O presente trabalho tem como objetivo principal facilitar o desenvolvimento de aplicações tolerantes a falhas ao propor um serviço de *log* desacoplado para

implementações de replicação máquina de estados. Este serviço é especialmente atrativo para arquiteturas baseadas em serviços, podendo executar em infraestruturas compartilhadas, tais como arquiteturas de micro-serviços ou ambientes de computação em nuvem. Sendo assim, alguns objetivos específicos são identificados:

1. *Usabilidade*: Oferecer uma interface transparente para a integração deste serviço de *logging* à diferentes aplicações SMR. O objetivo é garantir que poucas modificações sejam necessárias para associar uma aplicação ao serviço de *logging*, de modo que o desenvolvedor não precise se preocupar com particularidades referentes à inserção e obtenção de comandos em dispositivos de armazenamento persistente.

2. *Escalabilidade*: Permitir a adoção do serviço de *logging* por aplicações distintas, mantendo níveis satisfatórios de desempenho com o aumento de aplicações utilizando este serviço. Este objetivo cresce em importância ao vislumbrar o uso deste serviço em infraestruturas compartilhadas, onde um número dinâmico, e possivelmente elevado, de aplicações podem se beneficiar do serviço de *logging*.

3. *Confiabilidade*: Garantir a confiabilidade em protocolos de recuperação baseados no serviço de *logging* proposto. Desta forma, o serviço de *logging* não pode ferir a correção do modelo de replicação máquina e estados, além de prover o isolamento do *log* de diferentes aplicações.

Como contribuições deste trabalho, destacam-se a implementação de um processo *logger*, que dispõe de uma simples API (*Application Programming Interface*) para comunicação com diferentes aplicações; e a concepção da técnica de desacoplamento do *log*, contemplando diferentes estudos a respeito de seu desempenho e escalabilidade.

1.3 Metodologia

O trabalho teve início com um estudo da literatura existente na área de tolerância a falhas em sistemas distribuídos. Em especial, foram aprofundados os estudos nos temas de: replicação máquinas de estado, protocolos de consenso, e estratégias de recuperação eficiente. Com esta investigação foi possível identificar a relevância de contribuições nesta área do conhecimento, identificando-se um nicho de contribuição na recuperação de estado em arquiteturas de serviços compartilhados. Após discussões e esboços iniciais, foi originada a proposta de pesquisar a viabilidade no desacoplamento do *log*.

Com a concepção da proposta, iniciou-se a implementação de sistemas replicados com a técnica de SMR com o propósito experimental. Tais aplicações de teste tiveram como propósito avaliar métricas importantes para o desempenho da solução proposta, como a avaliação da sobrecarga gerada na execução normal da aplicação, a escalabilidade do serviço, e efeitos gerados na recuperação. Estes experimentos foram executados em um

ambiente de testes colaborativo e distribuído, onde uma análise comparativa das diferentes técnicas implementadas com a abordagem tradicional de recuperação em SMR pode ser conduzida de forma mais assertiva, levando em consideração importantes variáveis como a latência observada na comunicação de entidades distribuída por diferentes cargas de trabalho. A documentação do projeto e a escrita da dissertação seguiram de maneira paralela às etapas mencionadas.

1.4 Organização do Trabalho

O restante do trabalho é estruturado como segue. No Capítulo 2 são apresentados trabalhos relacionados. O Capítulo 3 apresenta uma conceituação a respeito de replicação por máquina de estados e mecanismos de recuperação de estado. No Capítulo 4 é apresentada a técnica de *log* desacoplado, que permite uma execução compartilhada de um mesmo processo *logger* por diferentes aplicações de forma escalável. O Capítulo 5 apresenta detalhes a respeito da implementação do serviço de *logging*, cobrindo a implementação de SMR e as aplicações de testes desenvolvidas. No Capítulo 6 são apresentados experimentos que exploram a utilização do processo *logger* em diferentes casos uso, considerando ambientes centralizados e compartilhados. Por fim, o Capítulo 7 conclui este trabalho e enuncia trabalhos futuros.

2 TRABALHOS RELACIONADOS

Este capítulo expõe trabalhos relacionados em duas categorias distintas. A Seção 2.1 apresenta trabalhos relacionados com o tema de tolerância a falhas em arquiteturas de serviços compartilhados, como ambientes de computação em nuvem. Estes trabalhos proveem abstrações distintas quanto à usabilidade de suas soluções, mas compartilham do mesmo objetivo de entregar segurança no funcionamento de aplicações nestes ambientes. A Seção 2.2 contempla contribuições relacionadas à recuperação de estado, apresentando trabalhos que exploram otimizações no processo de recuperação diminuindo a sobrecarga na execução, ou reduzindo o tempo de recuperação.

2.1 Confiabilidade em Ambientes Compartilhados

2.1.1 *Koordinator*

A técnica de conteinerização, ou virtualização baseada em contêineres, é amplamente utilizada em ambientes de computação em nuvem (Soltesz et al., 2007). Enquanto a técnica de virtualização tradicional consiste na execução de diferentes *máquinas virtuais*, cada uma com seu próprios sistemas operacional, sobre uma aplicação de gerenciamento denominado *hypervisor*; na conteinerização diferentes *contêineres* são executados sobre um mesmo sistema operacional, compartilhando um mesmo *kernel*. Esta abordagem traz alguns benefícios frente à virtualização, como melhor desempenho na execução de aplicações e melhor aproveitamento de recursos físicos, ao diminuir consideravelmente o uso de memória e armazenamento. Sistemas orquestradores de contêineres como Kubernetes (Burns et al., 2016) possibilitam o gerenciamento de diferentes contêineres agrupados em um *cluster*. Com o uso destes, é possível configurar esquemas de replicação para oferecer balanceamento de carga (*i.e.*, ao distribuir requisições à diferentes réplicas), e tolerância à falhas através de redundância. Esta abordagem de replicação não implementa a técnica de replicação máquina de estados. Ao ocorrer a falha de uma réplica no sistema, uma nova réplica é lançada a partir de uma nova imagem, gerando a perda total ou corrupção do estado do sistema (Netto et al., 2018).

Em (Netto et al., 2018) é apresentada uma solução para o desenvolvimento de

aplicações SMR em *clusters* Kubernetes, denominada *Koordinator*. Em SMR (vide Capítulo 3), é necessário assegurar que mensagens concorrentes encaminhadas por diferentes clientes sejam processadas por réplicas do sistema seguindo uma mesma ordenação total. Esta garantia é usualmente implementada através da utilização de algoritmos de consenso, porém sua utilização impõe diversas modificações no código das aplicações. Com o uso de *Koordinator*, torna-se possível alcançar a coordenação de diferentes réplicas sem que seja necessário modificar suas implementações, uma vez que esta abordagem é entregue como um serviço, sendo fracamente acoplada à processos de aplicações clientes.

Koordinator é executado como uma camada intermediária entre clientes e réplicas, sendo distribuído em três contêineres fisicamente distribuídos para possibilitar consenso e tolerância à falhas. Clientes enviam requisições à um *Proxy*, que é responsável por encaminhá-las à uma das réplicas do serviço de coordenação. Este é responsável por propor a requisição aos demais participantes do grupo de consenso. Após atingir o acordo, um dos participantes do protocolo encaminha a requisição às réplicas da aplicação, que podem ser executadas sobre uma mesma infraestrutura física dos contêineres do serviço.

A utilização da solução proposta por *Koordinator* é restrita somente para algoritmos de consenso baseados em líderes, uma vez que a comunicação entre o serviço de coordenação e as réplicas da aplicação é de responsabilidade de um único participante do protocolo, exercendo o papel de líder. Essa limitação é baseada em duas garantias fundamentais oferecidas por tais protocolos de consenso: (i) somente o líder é responsável por propor comandos aos demais participantes, sendo garantido que o mesmo sempre possuirá o estado mais atual de comandos propostos; e (ii) é garantido que nunca haverá mais do que um único participante exercendo o papel de líder no sistema. Ao considerar a extensa densidade de mensagem aos quais estes participantes (*i.e.*, líderes) estão submetidos, é de se esperar que a solução proposta adicione uma maior sobrecarga à este cenário, ao encarregar a estes processos a tarefa de entrega de comandos às réplicas da aplicação. Para minimizar este efeito, o trabalho propõe um modelo de consistência eventual em operações de leitura. Experimentos apontam o efeito de menor sobrecarga com a adição desta última abordagem.

O *Koordinator* implementa SMR em uma infraestrutura típica para serviços em computação em nuvem. Dessa forma, a disponibilidade é aumentada devido à replicação, além da garantia de consistência forte entre as réplicas. No entanto, o trabalho não aborda diretamente a recuperação de réplicas faltosas ou a inserção de novas réplicas em tempo de execução, o que torna-o um possível cliente do serviço de *logging* desacoplado proposto nesta monografia. Com modificações pontuais, réplicas do *Koordinator* poderiam usufruir do registro e recuperação de estado desacoplados oferecidos por este serviço.

2.1.2 *Reliable Runtime Replication Library*

Em (Pereira et al., 2019) é apresentada a biblioteca *R3Lib* (Pereira et al., 2019), que tem por objetivo facilitar o desenvolvimento de aplicações SMR abstraindo-se a utilização de protocolos de acordo. Com sua utilização é possível converter o código fonte de aplicações distribuídas em sistemas replicados, através da inserção de anotações e pela implementação de *interfaces* na aplicação cliente. Em sua arquitetura, a biblioteca faz uso de uma *Camada de Consenso Única*, implementada de modo à compartilhar uma mesma instância de consenso entre diferentes aplicações. Este compartilhamento impõe um melhor aproveitamento de recursos computacionais.

Esta abstração é oferecida com a implementação de dois módulos principais: o *Proxy de Consenso*, capaz de difundir requisições impostas por clientes através da API para a instância de consenso única, e retornar sua respectivas respostas computadas; e *Consensus Delivery*, que realiza a entrega de comandos propostos para a camada de consenso às diferentes réplicas da aplicação. Um ponto importante nesta implementação modular é de que se torna possível a utilização de diferentes protocolos de consenso, uma vez que basta implementar os módulos mencionados para se adequar aos protocolos desejados.

Experimentos conduzidos demonstram uma degradação na vazão ao comparar uma mesma aplicação seguindo a abordagem de SMR tradicional com sua versão utilizando as abstrações oferecidas pela biblioteca. Entretanto, é constatado que ao executar diferentes aplicações em uma mesma infraestrutura, a vazão total representada pelo somatório das vazões observadas por cada uma diferentes aplicações é superior com a utilização da biblioteca. Isso deve-ser pelo melhor reaproveitando de recursos ocasionado pelo compartilhamento do protocolo de acordo.

Assim como o *R3Lib*, o serviço de *logging* proposto neste trabalho é projetado para ser compartilhado entre múltiplos serviços independentes. Este compartilhamento de serviços é uma demanda atual, especialmente pelo uso frequente de infraestruturas compartilhadas para prover serviços na Internet (JoSEP et al., 2010). Um aspecto positivo destas abordagens é a identificação de melhor desempenho com o acréscimo de clientes do serviço compartilhado, seja ele o serviço de consenso no caso do *R3Lib*, ou o serviço de *logging*, como observado em (Pereira et al., 2019) e no Capítulo 6, respectivamente. No Capítulo 5 é discutida em maiores detalhes a similaridade de ambos trabalhos em caráter de implementação.

2.2 Recuperação Eficiente

2.2.1 *UpRight Cluster Services*

UpRight (Clement et al., 2009) caracteriza-se como uma biblioteca para replicação tolerante a falhas, que procura separar detalhes de implementação da aplicação dos de-

talhes do protocolo de replicação. Com sua utilização é possível desenvolver aplicações tolerantes à falhas bizantinas (*Byzantine Fault Tolerance* – BFT) de maneira transparente, sem o conhecimento de detalhes a respeito do protocolo de difusão atômica; ou então converter implementações tolerantes à falhas por colapso (*Crash Fault Tolerance* – CFT) em BFT.

Em relação à utilização da biblioteca, o autor de uma aplicação deve conhecer a interface disponibilizada, detalhes sobre como a aplicação processa requisições, e como ela atualiza seu estado para recuperação, podendo ser escolhidas três abordagens distintas de *checkpoint* disponibilizadas pela biblioteca:

- ***Stop and Copy*:** Maneira tradicional onde a evolução do processo da aplicação é pausada momentaneamente para a realização da captura de estado.
- ***Helper Process*:** Realiza o *checkpoint* de maneira assíncrona ao dividir a aplicação em dois processos independentes em cada réplica. O primeiro é denominado *Primary*, e este caracteriza-se como a própria aplicação sem o protocolo de recuperação por *checkpoint*. O segundo é chamado *Helper*, e nele é omitido o encaminhamento de respostas à clientes, e sua execução é pausada para realização de *checkpoints* assim como *Stop and Copy*.
- ***Copy-On-Write*:** Implementa a técnica de cópia-na-escrita, que possibilita a execução concorrente do processo de *checkpoint* com a execução normal com tanto que haja disponibilidade de memória para criação de cópias temporárias.

UpRight se assemelha ao trabalho proposto no comportamento de execução concorrente da técnica de *helper process* para recuperação de estado, a qual tem como objetivo reduzir a sobrecarga na aplicação assim como o uso do processo de *logging* desacoplado. Os trabalhos diferenciam-se quanto ao objetivo principal da abstração oferecida. Enquanto o *UpRight* visa facilitar o desenvolvimento de sistemas BFT, neste trabalho pretende-se, através do desacoplamento do processo de *logging* da aplicação, reduzir o impacto na vazão causado pelo gerenciamento do *log*.

2.2.2 *Parallel Logging*, *Sequential Checkpointing* e *Collaborative State Transfer*

Em (Bessani et al., 2013), os autores destacam que abordagens utilizadas para durabilidade em aplicações SMR, como *logging*, *checkpoint* e transferência de estado, podem impactar significativamente no desempenho destas aplicações. Os autores apresentam três técnicas que procuram minimizar esta degradação de desempenho, sem o uso de recursos adicionais de *hardware*.

Parallel Logging explora a redução da sobrecarga gerada pelo processo de escrita de novos comandos ao *log*. Essa abordagem beneficia-se da maior largura de banda de discos rígidos ao realizar a escrita de novos comandos em lotes. Para aplicação, se faz

necessária a entrega de grupos de comandos pelo protocolo de difusão atômica para que todo conjunto possa ser escrito em uma única chamada de sistema. A técnica também explora a execução paralela do processo de escrita de novos comandos em disco com o processamento de novas requisições. Porém, como uma resposta só pode ser retornada a um cliente após sua requisição ser processada e registrada no *log*, a eficiência deste procedimento depende diretamente da relação entre o tempo de processamento de um lote de comandos com a latência imposta pela operação de escrita destes registros.

A técnica de *Sequential Checkpointing* propõe a realização de processos de captura de estado de maneira sequencial e assíncrona entre o grupo de réplicas do serviço. A maneira tradicional prevê a realização do *checkpoint* de maneira síncrona, onde todas réplicas efetuam o processo em um mesmo intervalo, considerando critérios como número de comandos executados ou tempo de execução desde o último *checkpoint*. Esta abordagem visa eliminar os períodos de total indisponibilidade do serviço observados na maneira tradicional de captura de estado.

Collaborative State Transfer é um protocolo de transferência de estado que visa dispersar a tarefa de transmissão de *logs* e *checkpoints* de maneira equilibrada entre as réplicas do serviço. Com esta abordagem, partes do *checkpoint* e do *log* podem ser transmitidas à réplica em recuperação por diferentes réplicas do serviço, aliviando o custo total na transferência de estado.

Assim como o trabalho proposto, Bessani *et al.* (Bessani et al., 2013) apresentam propostas para reduzir custos relacionados à durabilidade e protocolos de recuperação. Dessa forma, estratégias como *parallel logging* poderiam ser aplicadas no serviço de *logging* desacoplado proposto. Além disso, protocolos de recuperação podem combinar a estratégia de *logging* desacoplado com *sequential checkpoint* e *collaborative state transfer*.

2.2.3 *Speedy Recovery* e *On-demand State Recover*

Em (Mendizabal et al., 2017b), são apresentadas duas alternativas para recuperação de estado na Replicação Máquina de Estados Paralela (P-SMR, *Parallel State Machine Replication*). Este modelo de replicação é uma extensão do SMR tradicional, que permite a execução paralela de comandos independentes para alcançar alta vazão.

Speedy Recovery é um protocolo de recuperação que explora a semântica de comandos utilizados na aplicação para mapear eventuais dependências entre comandos, reduzindo consideravelmente o tempo de recuperação ao explorar a execução paralela de comandos independentes. Esta identificação de dependência é realizada combinando-se três estratégias: interpretação de comandos em lotes, em que eventuais dependências são consideradas ao analisar a aplicação de todo um grupo de comandos na máquina de estados; rápida detecção de conflitos, onde cada conjunto de comandos agrupados possui uma assinatura própria que representa todas variáveis impactadas pela execução do lote; e menor sobrecarga no tratamento de dependências, uma vez que a detecção de dependência entre

lotes de comandos é feita por meio de comparações entre mapas de bits, demonstrando-se menos custosa do que a administração de um grafo acíclico dirigido (DAG – *Directed Acyclic Graph*), usado em outras implementações de P-SMR (Kotla and Dahlin, 2004).

Em aplicações que realizam o salvamento de *checkpoints* para recuperação de estado, o processamento do *log* de comandos só pode ser realizado após a instauração do novo estado. A abordagem de *On-demand State Recover* trata o estado de *checkpoint* como uma coleção de segmentos, que são instalados somente quando são requisitados para o processamento de um novo comando. Deste modo, o *log* não precisa ser completamente transferido e processado no momento da recuperação, sendo esta postergação do processamento de partes do *log* benéficos para o desempenho do sistema.

O trabalho proposto se assemelha a *Speedy Recovery* no propósito de redução do tempo de recuperação de uma réplica falha. Porém, ao invés de apresentar maneiras de agilizar o processamento do *log* de comandos informado pelo protocolo de difusão atômica, este trabalho propõe a construção e manutenção do *log* de recuperação por um processo independente.

2.2.4 *Full*, *View*, e *EpochSS*

Algoritmos de recuperação que exploram características do protocolo de acordo para restaurar réplicas de maneiras mais eficientes são apresentados em (Kończak et al., 2019). Adotando-se Paxos (Lamport, 1998) como mecanismo de consenso, os autores propõe três variações do algoritmo. A primeira destas, denominada *FullSS*, prevê que escritas frequentes em armazenamento persistente sejam realizadas durante intervalos de execução estável, o que garante a robustez da recuperação mesmo em cenários de falhas catastróficas. *ViewSS* impõe mudanças menos intrusivas na implementação de Paxos tradicional. Neste algoritmo, uma fase adicional de recuperação é executada à nível do algoritmo de consenso, o que garante uma sobrecarga menor na execução da aplicação, e oferece um desempenho similar ao modelo tradicional de recuperação. A abordagem *EpochSS* estipula uma maneira de monitorar o número de vezes que um processo foi reiniciado devido à recuperação. Este controle permite uma recuperação mais eficiente por parte do procolo, uma vez que mensagens redundantes não são retransmitidas pelos grupo de réplicas ativas.

As três abordagens de redução de custo de recuperação apresentadas em (Kończak et al., 2019) são dependentes do protocolo de consenso, efetuando alterações no funcionamento interno do Paxos. Diferentemente, a proposta apresentada neste trabalho é independente de protocolo, portanto não exige modificações no subsistema de comunicação confiável entre as réplicas.

3 REPLICAÇÃO MÁQUINA DE ESTADOS

A técnica de replicação é uma estratégia conhecida para prover alta disponibilidade e tolerância à falhas em sistemas distribuídos. Especificamente, a abordagem de replicação por máquina de estados (*State Machine Replication* – SMR) (Lamport, 1978; Schneider, 1990) possibilita uma maior disponibilidade e escalabilidade do serviço ao tratar o grupo de servidores como um conjunto de réplicas idênticas, que eventualmente refletem os mesmos estados, sem a necessidade de coordenação explícita entre as réplicas. Em SMR, as réplicas são máquinas de estado que desempenham papéis equivalentes e são organizadas como um grupo (Coulouris et al., 2011). Do ponto de vista de cada réplica, não há uma distinção de papéis como se faz presente na replicação passiva, de modo que cada réplica opera como um processo independente sempre respondendo à requisições impostas por clientes.

Apesar de simples, esta poderosa abstração é responsável por garantir o comportamento correto de sistemas replicados mesmo em cenários falhos. Ao considerar um serviço como uma máquina de estados, define-se que cada alteração de estado seja decorrente do processamento de requisições impostas por clientes, denominadas *comandos*. A execução de comandos impõe a tomada de novas transições ao sistema, que podem alterar o estado do serviço e retornar um resultado. De acordo com (Schneider, 1990), para a implementação de uma máquina de estados tolerante à falhas é necessário assegurar que as diferentes réplicas do sistema sempre tomem as mesmas transições de estado produzindo as mesmas saídas para cada comando executado. Esta garantia pode ser enunciada através de três propriedades:

- **Acordo:** Todas réplicas não falhas devem receber os mesmos comandos;
- **Ordem:** Todos réplicas não falhas devem processar os comandos recebidos na mesma ordem.
- **Determinismo:** A execução de comandos deve ser determinística, isto é, diferentes réplicas sempre efetuam uma mesma transição ao executarem comandos iguais partindo de um mesmo estado.

Em resumo, é necessário garantir que todas as réplicas recebam a mesma sequência de comandos, para que tomem as mesmas transições e assim, eventualmente atinjam os mesmos estados, progredindo de forma consistente. Uma vez que todas réplicas comecem a execução em um mesmo estado inicial, e recebam uma mesma sequência de comandos, deve ser assegurado que estas *executem* todos os comandos entregues. Esta propriedade visa a segurança a falhas acidentais (*safety*), preservando o funcionamento correto do sistema. Um critério importante que sintetiza a correção da técnica de SMR é a propriedade de *linearizability* (Herlihy and Wing, 1990). Um sistema é dito "linearizável"caso seja possível reordenar os comandos de clientes de maneira a preservar a ordem definida previamente pela aplicação, respeitando sua semântica de comandos. Sistemas linearizáveis são também considerados fortemente consistentes, uma vez que cada estado é atingido após a computação definitiva do estado anterior, e *safety* é sempre garantida.

No contexto de SMR, a definição de consenso (Pease et al., 1980) está fortemente relacionada com as propriedade de *acordo* e *ordem* descritas em (Schneider, 1990). Sendo necessário assegurar que réplicas recebam a mesma sequência de comandos seguindo uma mesma ordem, é correto afirmar que a interação entre as entidades participantes apresenta-se como uma alternativa para se obter consenso. Neste sentido, o termo consenso é atribuído como a uniformidade na execução de um mesmo comando em um dado instante de tempo por diferentes réplicas do sistema. Assim, ao resolver uma sequência de rodadas de consenso é observada a execução de uma sequência uniforme de comandos respeitando uma ordenação total.

Acordo e consenso são duas propriedades indissociáveis (Hadzilacos and Toueg, 1994). Para fins de exemplo, considera-se um cenário onde há a presença de entidades pertencentes ao conjunto $E = \{e_1, e_2, \ldots, e_n\}$ que necessitam decidir opiniões em conjunto (e.g. um parlamento) e que comunicam-se através de troca de mensagens. Ao considerar a presença de duas entidades ($n = 2$), pode-se dizer que há acordo quanto estas *acordam* em uma decisão única após sua interação, resultando em uma uniformidade de opiniões. Ao adicionar participantes neste conjunto de entidades, as entidades primordiais e_1 e e_2 preservam o acordo previamente estabelecido. Neste exemplo, não será assegurado o consenso antes que as entidades interajam entrem si. O consenso será atingido quando houver a uniformidade de opiniões entre todo o conjunto de entidades.

Disponibilidade é um requisito fundamental em aplicações distribuídas, descrevendo a probabilidade do sistema estar funcionando corretamente em um determinado instante de tempo. A disponibilidade em uma aplicação SMR é garantida em cenários livres de falhas bizantinas desde que uma maioria de réplicas corretas esteja ativa. Esta exigência de um quórum dado pela maioria de réplicas corretas é um requisito do módulo de consenso (Turek and Shasha, 1992) e descreve o *fator de replicação* do serviço, dado pela equação $r = 2f + 1$, onde r representa o número de réplicas necessárias em um sistema para tolerar até f réplicas falhas.

A Figura 1 exemplifica a topologia de um grupo de réplicas em uma típica implementação de SMR. Pode-se observar diferentes clientes encaminhando requisições (1) a um conjunto de réplicas, onde as camadas sobrepostas representam clientes e réplicas distintas. Um módulo de consenso é responsável por assegurar a ordem total de entrega de mensagens para as réplicas, garantindo as propriedades de *acordo* e *ordem* (2), através da criação de um *log* persistente representando uma sequência única de comandos comum a todos servidores. A execução determinística destes comandos (3) e consequentemente a resposta ordenada às requisições impostas (4), faz com que todas réplicas eventualmente atinjam os mesmos estados, permitindo a evolução consistente da aplicação.

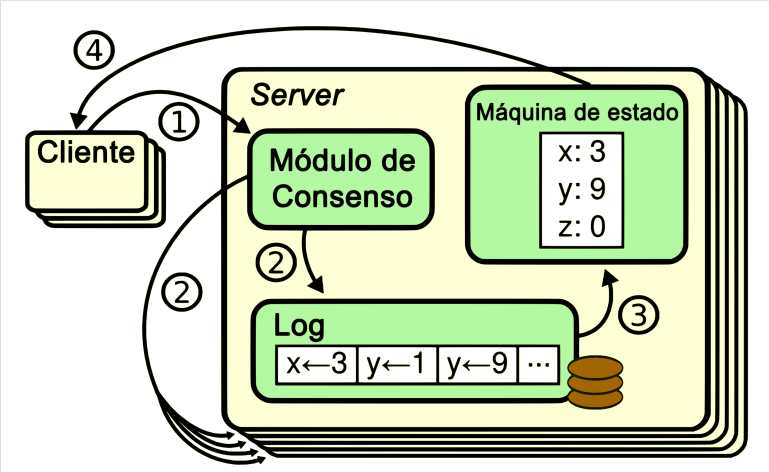

Figura 1: Exemplo replicação máquina de estados.
Adaptada de: (Ongaro and Ousterhout, 2014)

3.1 Recuperação de Estado em SMR

A replicação do sistema seguindo a abordagem SMR permite alcançar altos níveis de disponibilidade de serviço, porém para a aplicação retomar a execução normal após a ocorrência de falhas se faz necessário a implementação de técnicas de recuperação de estado. Nesta seção são apresentadas duas abordagens comuns para prover *durabilidade* em sistemas SMR, a escrita de comandos em um *log* de recuperação e a criação de estados de restauração denominados *checkpoints*. Ambas técnicas realizam o salvamento de seus dados na unidade de armazenamento persistente do sistema.

3.1.1 Recuperação por *Log*

A abordagem de *logging* caracteriza-se pelo registro de comandos em arquivos para serem reprocessados com o início do processo de recuperação, permitindo a uma réplica falha alcançar um estado mais atual, consistente com as demais réplicas. São conheci-

das três diferentes maneiras de implementar o processo de escrita de novos comandos (Elnozahy et al., 2002), sendo estas:

- ***Logging* Pessimista:** Respostas a clientes são encaminhadas após o registro de seus respectivos comando no *log*. Impõe uma sobrecarga maior na aplicação, porém evitam a criação de comandos órfãos.

  Comandos órfãos são aqueles que foram efetivados na máquina de estados da aplicação, porém não apresentam nenhum registro equivalente no *log* de recuperação.

- ***Logging* Otimista:** Um comando é registrado no *log* após o encaminhamento de sua resposta ao cliente. Representa um menor impacto no desempenho da aplicação se comparado à estratégia pessimista, porém permite a criação de comandos órfãos em razão de falhas, o que dificulta a recuperação;

- ***Logging* Causal:** Procuram mesclar uma menor sobrecarga na aplicação com a redução de registro de comandos órfãos, porém exigem um verificação de recuperação mais complexa.

Usualmente em aplicações SMR, o gerenciamento do *log* de comandos é de responsabilidade do protocolo de consenso utilizado (Lamport, 1998; Ongaro and Ousterhout, 2014), sendo de sua responsabilidade o registro de novas comandos e garantia de consistência de estado.

3.1.2 *Checkpointing*

O processo de *checkpointing* consiste na criação de estágios de recuperação, que podem ser criados ao definir uma contagem de comandos ou frequência de tempo de execução. Pela necessidade de realizar a leitura do espaço de endereçamento para capturar o estado da aplicação, o processo de *checkpointing* age de forma concorrente com a execução normal dessa, requerendo acesso exclusivo à região de memória a fim de que o estado capturado em sua execução caracterize-se como um estado consistente do sistema.

Em razão dessa imprescindibilidade de exclusão mútua, a realização de um *checkpoint* interrompe momentaneamente a execução da aplicação, evitado escritas subsequentes para garantir a consistência do estado (Bessani et al., 2013). Na Figura 2 é ilustrada a vazão de um serviço SMR e como esta se comporta no decorrer do seu tempo de execução. É possível observar que aproximadamente a partir do instante indicados por 120 segundos há uma clara redução na vazão da aplicação devido à necessidade de sincronização previamente mencionada. Este período de indisponibilidade do serviço, evidenciado pela vazão nula, é mais duradouro ao realizar armazenamento em dispositivos de memória secundária.

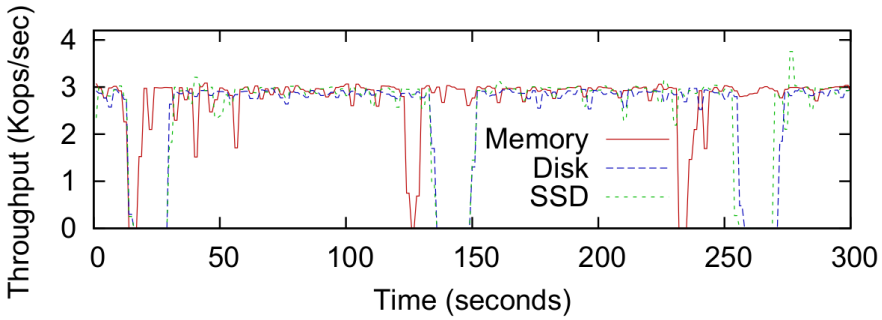

Figura 2: Redução de vazão ocasionada pelo processo de *checkpoint* ao considerar diferentes tecnologias de armazenamento.
Fonte: (Bessani et al., 2013)

Vale lembrar que o uso do protocolo de recuperação por *checkpoint* não exclui a necessidade do registro de comandos em *log*. Como salvamento de contexto é realizado em intervalos periódicos, o registro de comandos em arquivos de *log* se faz necessário para que os comandos efetivados entre a criação de dois *checkpoints* possam ser reprocessados pela aplicação.

4 DESACOPLAMENTO DO *LOG*

Neste trabalho é proposta a realização do procedimento de *logging* de maneira desacoplada à aplicação, oferecida à esta como um serviço. Com o desacoplamento, é possível realizar um compartilhamento mais eficiente de recursos físicos em estruturas compartilhadas, além de aliviar o processamento realizado pelas réplicas do sistema ao encarregar a tarefa de registro de novos comandos à processos especializados. Estes processos denominados *loggers* participam do serviço replicado, compartilhando do mesmo protocolo de acordo das réplicas da aplicação.

4.1 Modelo do Sistema

Adota-se um sistema com um número limitado de réplicas, representado pelo conjunto $R = \{r_1, r_2, \ldots, r_n\}$. Requisições geradas por um número ilimitado de clientes $C = \{c_1, c_2, \ldots\}$ são encaminhadas ao conjunto de réplicas, que acordam uma sequência única de comandos através de um protocolo de comunicação confiável capaz de assegurar *atomic broadcast* (Lamport, 1978). Tal protocolo garante que (i) se um processo correto transmite uma mensagem m, então haverá um intervalo i dentre um número infinito de possibilidades em que todas réplicas corretas entregarão (i, m); (ii) se uma réplica correta entrega (i, m), então eventualmente todas réplicas corretas irão receber (i, m); (iii) nenhuma réplica entrega (i, m') sendo $m \neq m'$; (iv) m é proposta por algum processo.

Adota-se um modelo de computação *parcialmente síncrono* (Dwork et al., 1988). Este modelo considera que o sistema comporta-se de maneira assíncrona até um instante desconhecido, onde passa a agir de maneira síncrona, respeitando limites no tempo de processamento e comunicação. É considerado a permanência do sistema neste período de estabilidade por tempo suficiente para a evolução da computação. Esta restrição é necessária uma vez que em (Fischer et al., 1982) é destacada a impossibilidade de se atingir consenso ao considerar um sistema totalmente assíncrono, já que é impossível distinguir o comportamento de um processo falho de um processo lento.

O sistema é equipado com um número limitado de *loggers*, descritos pelo conjunto $L = \{l_1, l_2, \ldots, l_n\}$. Processos *loggers* recebem requisições oriundas do mesmo pro-

tocolo de comunicação confiável ao qual as réplicas em R participam. É assumida a ocorrência de no máximo f falhas de um total de $n = 2f + 1$ servidores, e k falhas de um total de $n = k + 1$ *loggers*. Neste modelo não é assumida a ocorrência de falhas ocasionadas por comportamento bizantino (Lamport et al., 1982).

Réplicas falhas retomam ao estado de execução normal com o processamento de comandos contidos em um *log* armazenado em memória persistente. Todas as réplicas são capazes de realizar o procedimento de recuperação com o arquivo processado, uma vez que não é considerado corrupção deste com o colapso do sistema.

4.2 Correção do *Logger*

Arquivos de *log* são utilizados para prover um histórico de comandos para réplicas em recuperação. Usualmente, durante o processo de recuperação, uma réplica falha deve primeiramente buscar um estado de *checkpoint* armazenado localmente ou requisitar um estado à outra réplica do grupo; após a instalação do estado, dá-se início à obtenção e processamento do *log*, que acaba por aplicar todos comandos cujos efeitos não estão contidos no estado transferido. Ao utilizar a abordagem de *log* desacoplado, uma réplica em recuperação deve requisitar o arquivo de *log* à um dos processos *logger* em execução, sendo esta uma pequena alteração frente à maneira tradicional.

Ao desacoplar o *log* da aplicação, é importante considerar o fato de que podem ocorrer diferenças eventuais entre os estados observados por *loggers* e réplicas. Por exemplo, é possível que em aplicações de alta vazão com comandos que requerem um baixo custo de processamento, os *loggers* momentaneamente atrasem seu estado de computação quando comparado às réplicas do sistema. Embora esta situação possa acarretar atrasos na recuperação, isso não representa uma violação na correção do algoritmo, pois uma vez que *loggers* e réplicas recebem mensagens do mesmo protocolo de consenso, os *loggers* eventualmente possuirão o intervalo de comandos necessários para a réplica em recuperação. De outro modo, caso *loggers* estejam à frente no estado de computação, réplicas em recuperação irão receber de maneira imediata todos os comandos necessários para recuperação.

4.3 Considerações sobre desempenho

Quando o *log* é separado da aplicação, algumas considerações importantes de desempenho emergem devido ao desacoplamento em si, e as diferentes velocidades relativas das réplicas e dos processos *logger*.

Primeiro, com o desacoplamento as réplicas de serviço não precisam registrar cada comando entregue pelo protocolo de acordo, o que resulta em um menor uso do dispositivos de armazenamento secundário. Segundo, como uma maneira prática de contor-

nar a defasagem do *logger* frente à aplicação, é com a execução destes processos em máquinas especializadas para operações de entrada e saída (E/S – *Entrada, Saída*). Esta especialização pode ser ofertado tanto em nível de *software*, na forma de sistemas de arquivos otimizados ou com adoção de estratégia de *parallel logging* (Bessani et al., 2013), quanto em *hardware*, com o uso de dispositivos de armazenamento com alta largura de banda e dispositivos de armazenamento de estado sólido (SSD – *Solid-State Drive*). Desta forma, não somente arquivos maiores poderiam ser gerenciados pelos processos *loggers*, como também efeitos de transferência de estado e truncamento do *log* seriam menos perceptíveis à usuários do sistema. Como efeito colateral positivo, a redução da frequência na tomada de *checkpoints* resultaria diretamente em um maior desempenho observado nas réplicas, como discutido em (Mendizabal et al., 2016).

Um terceiro aspecto importante diz respeito às velocidades relativas dos processos de réplica e *logger*.

- Se *loggers* forem mais rápidos que as réplicas e estas momentaneamente não fornecerem taxa de transferência suficiente para atender às solicitações recebidas, elas ficarão para trás no estágio de computação. Isso não prejudicará a recuperação, conforme discutido na Seção 4.2. De fato, como isso representa um cenário de saturação nas réplicas, seria ainda pior se tivéssemos o caso tradicional, onde manter o *log* de comandos é responsabilidade das réplicas.

- Se as réplicas forem mais rápidas que os *loggers* e se eles não fornecerem taxa de transferência suficiente para registrar solicitações recebidas, os mesmos ficarão atrás das réplicas. Se essa saturação for momentânea, eventualmente o *logger* recuperará o atraso, também como discutido na Seção 4.2. Nesse caso, uma solicitação para recuperar um intervalo de *log* sofrerá alguma latência e, eventualmente, será atendida quando o *logger* tiver todos os comandos solicitados.

  Se a saturação do *logger* se tornar proeminente, será observado um aumento gradual no atraso dos comandos entregues aos processos *loggers* e um aumento no atraso na recuperação de réplicas, pois uma réplica em recuperação solicitaria comandos que ainda não foram armazenados pelo *logger*, portanto, é necessário aguardar até que este processo possa transferir todo o estado para recuperação. Para atenuar esses problemas, uma réplica pode truncar comandos de *log* sempre que possível, ou seja, logo após a criação de um *checkpoint*. Dessa forma, os processos *loggers* seriam solicitados a truncar comandos que ainda não haviam processado. Ao fazer isso, recebendo comandos atrás do ponto de truncamento, os *loggers* podem simplesmente descartar esses comandos e retomar o registro de comandos após o ponto de truncamento. Mesmo essa mitigação sendo possível, a configuração ideal é tal que *loggers* não fiquem atrás das réplicas de serviço.

5 IMPLEMENTAÇÃO

Para implementação da biblioteca de *log* proposta, o processo de *logging* desacoplado, e para as aplicações de teste foi utilizada a linguagem de programação Go [1]. A linguagem desenvolvida pela Google e tendo sua primeira versão lançada em 2012 traz o suporte a execução concorrente como uma de suas principais funcionalidades, através de fortes abstrações no controle de linhas de execução concorrentes, denominadas *goroutines*, e na comunicação destas por canais (Hoare, 1978). Sua utilização no desenvolvimento de aplicações distribuídas tem se tornado cada vez mais popular no decorrer dos anos (Dinh et al., 2018; Hashicorp, 2014), especialmente ao considerar a sua aplicação em arquiteturas de micro-serviços executando sobre ambientes virtualizados.

5.1 Implementando SMR

Ao implementar SMR, algumas limitações são imediatamente impostas sobre o sistema para manter a correção do modelo de replicação, de modo a satisfazer suas propriedades. Em (Bessani and Alchieri, 2014) os autores apontam duas limitações principais, sendo estas:

- Para garantir o determinismo na execução de comandos, o sistema deve ser em princípio *single threaded*; aplicando comandos de maneira sequencial de uma única *thread* de execução, nunca lançando diferentes *threads* para serem executadas de forma concorrente. Implementações práticas de máquinas de estado paralelas (Kapritsos et al., 2012; Mendizabal et al., 2017a; Alchieri et al., 2018) implementam diferentes soluções para esquivar desta forte restrição, de modo a preservar a execução determinística de comandos, porém sua explicação foge deste escopo.

- Formalmente, máquinas de estado não podem iniciar conexões com clientes, uma vez que são limitadas a somente processarem requisições. Sendo assim, clientes que requerem atualizações constantes devem ativamente requisitar ao serviço replicado notificações sobre possíveis mudanças de estado, uma vez que estes só tem

[1] Mais informações a respeito da linguagem como guias, documentação, e passos de instalação podem ser encontradas em: https://golang.org/

conhecimento dos comandos originados por si próprio. Essa restrições impõe uma densidade maior de mensagens sobre a infraestrutura de rede, além de aumentar a carga de requisições imposta ao sistema, o que pode comprometer desempenho.

Conforme definido no Capítulo 3, é fundamental assegurar a entrega de mensagens seguindo uma ordenação total para todas as réplicas corretas do sistema. Neste sentido, são conhecidas duas soluções práticas para a resolução da ordenação total em implementações SMR:

1. Utilização de algoritmos de consenso (Lamport, 1998; Oki and Liskov, 1988; Ongaro and Ousterhout, 2014) capazes de garantir a ordenação total de comandos propostos ao grupo de réplicas. Esta abordagem utiliza-se da interação entre as diferentes réplicas através de troca de mensagens, e permite que comandos propostos sejam acordados e eventualmente refletidos em todo o grupo.

2. Utilização de processos sequenciadores (e.g. *Proxies*), que interceptam comandos propostos e encaminham estes ao conjunto de réplicas de maneira ordenada através de um canal confiável. Canais confiáveis implementam garantias de entrega ordenada, preservando a relação de "acontece antes"descrita por (Lamport, 1978).

Neste trabalho, optou-se pela primeira alternativa utilizando o protocolo Raft para a implementação de módulo de consenso. Raft (Ongaro and Ousterhout, 2014) é um algoritmo de consenso distribuído, utilizado para assegurar a ordenação total necessária para aplicação da técnica de SMR e, consequentemente, garantir uma evolução confiável do grupo de réplicas mesmo na existência de nodos falhos. Raft foi desenvolvido visando compreensibilidade, com o objetivo de ser de mais fácil compreensão que Paxos (Lamport, 1998) e de equivalente desempenho.

Apesar de recente, Raft apresenta semelhanças com o algoritmo de consenso *Viewstamped Replication*, proposto no final da década de 80 (Oki and Liskov, 1988). O protocolo também apresenta características próprias como *strong leadership*, que simplifica o gerenciamento do *log* de comandos ao encarregar a tarefa de *commit* de novos comandos única e exclusivamente ao líder. Nesta seção é apresentado o funcionamento Raft, e como ele lida com três pontos importantes de seu algoritmo: eleição de líder, replicação do *log* e garantia do funcionamento correto (*safety*).

5.1.1 Raft: Funcionamento Básico

Durante sua execução, o protocolo trata o tempo como períodos de duração arbitrária denominados *terms*. Estes intervalos são computados na forma de números inteiros, monotônicos e consecutivos, e representam períodos de execução de um líder na aplicação. A Figura 3 ilustra esta divisão de tempo pelo protocolo. É possível observar que um novo *term* sempre é iniciado na ocorrência de um novo processo de eleição (intervalos

representado em azul na figura), e permanece constante até a ocorrência de uma falha do processo líder. Um *term* pode ser rapidamente finalizado caso um processo de eleição não resulte em um novo líder, conforme observado em *t3*.

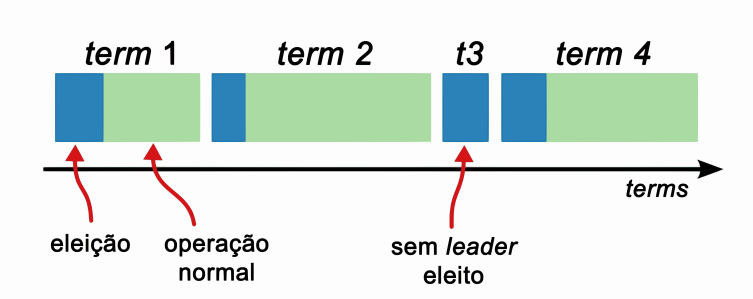

Figura 3: Divisão de tempo do Raft.
Adaptada de: (Ongaro and Ousterhout, 2014)

Em um conjunto de servidores utilizando Raft como mecanismo de consenso, um nodo pode assumir três diferentes estados de participação. A Figura 4 ilustra estes estados e apresenta suas respectivas transições.

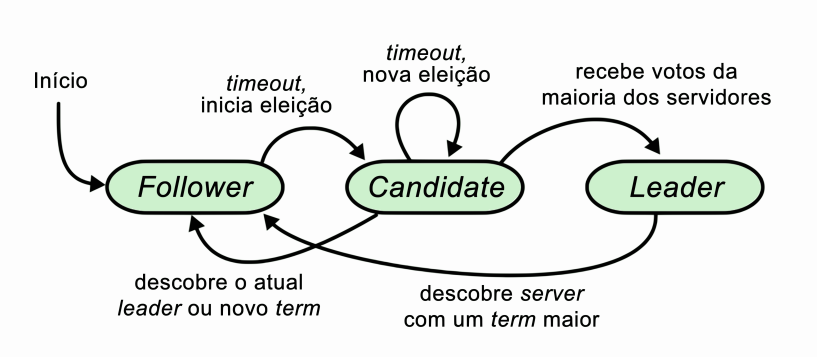

Figura 4: Possíveis estados de um servidor em um grupo Raft.
Adaptada de: (Ongaro and Ousterhout, 2014)

Follower: Participa de forma passiva no cluster, somente respondendo às chamadas de procedimento remotas (RPCs, *Remote Procedure Call*) de Requisição de Votos invocada por candidatos e de Propagação de Comando invocada pelo líder. Ao receber a primeira, o *Follower* responde **afirmativamente** somente no caso do *term* do candidato ser maior que o seu, assegurando que o candidato esteja no mínimo mais adiantado que seu próprio estado. Ao receber a segunda, que inicia o quórum de consenso do algoritmo, verifica o *term* do comando proposto pelo líder para:

1. Responder de modo **negativo** caso o comando proposto seja de um *term* menor que de seu último comando processado;

2. Responder de modo **negativo** caso não possua nenhum comando em seu próprio *log* no índice *prevLogIndex* que combine com o *prevLogTerm* informado;

3. Responder de modo **positivo** e remover parte de seu próprio *log* caso já possua um comando de mesmo índice, porém *term* diferente daquele proposto pelo líder;

4. Responder de modo **positivo** e adicionar o comando ao seu próprio *log* caso o comando proposto seja de maior índice e *term*.

Candidate: Estado intermediário, alcançado com o não recebimento da Propagação de Comandos imposta pelo líder em um definido intervalo de tempo, a mesma RPC de consenso funciona como um mecanismo de *heartbeat*. Ao assumir este estado, o servidor inicia o processo de eleição de um novo líder, cujo procedimento é demonstrado pelo Algoritmo 1. É possível perceber que uma réplica só assume o estado *Leader* quando recebe a maioria dos votos dos demais participantes do grupo, e retoma ao estado *Follower* ao detectar a presença de um novo líder.

Algorithm 1 Eleição de Líder do Raft

```
1:  procedure
2:      estadoAtual ← Candidate
3:      termAtual ← termAtual + 1
4:      Votos[ ] ← self
5:
6:      start TimeoutEleicao
7:      send RPCdeRequisicaoVotos to TodosParticipantes
8:
9:      if RecebeuMaioriaDeVotos then
10:         estadoAtual ← Leader
11:     if RecebeuRPCdeComandosDeNovoLider then
12:         estadoAtual ← Follower
13:     if TimeoutEleicao then
14:         goto procedure
```

Leader: Papel principal a ser exercido por um servidor no cluster, único responsável por propor novos comandos aos *Followers* a fim de se obter consenso. Ativamente envia RPC de Propagação de Comandos mesmo em períodos ociosos, de modo a prevenir a eleição de um novo líder. Ao receber um novo comando por um cliente, registra-o em seu próprio *log*, propõe a aplicação do comando ao grupo e responde ao cliente em caso afirmativo.

A evolução para um novo estado no sistema é realizada caso o líder receba uma maioria de respostas afirmativas do grupo após a propagação da RPC citada. Caso o índice do comando atualmente proposto seja maior que o índice de *log* atual de um *Follower*, envia no procedimento de propagação a este servidor todos os comandos contidos no *log* iniciando no índice desejado até o índice do comando atual. No caso da propagação de comandos falhar por inconsistência de *log* no destinatário, reenvia a requisição decrementando o índice do comando até retorno afirmativo.

5.1.2 Raft: Características Importantes

Em qualquer instante de tempo, é garantido pelo protocolo a veracidade das seguintes propriedades:

- No máximo um único líder será eleito em um mesmo *term*.
- Um líder nunca sobrescreve ou deleta comandos em seu *log*, somente adiciona novas entradas.
- Se dois *logs* possuem um registro de mesmo índice e *term*, então os dois *logs* são idênticos até este índice.
- Se um novo registro foi efetivado em um dado *term*, então ele estará presente nos *logs* de todos eventuais líderes em todos *terms* futuros.
- Se um servidor aplicou um comando em um dado índice a sua máquina de estados, nenhum outro servidor participante irá aplicar um comando diferente no mesmo índice.

Como já mencionado, para que um comando seja efetivado na máquina de estados basta que o líder que o propôs obtenha uma resposta afirmativa da maioria dos *Followers* presentes no *cluster*. Eventuais inconsistências de estado do *log* dos participantes são gerenciadas pelo próprio líder, que força os participantes a copiarem de seu estado. Em outras palavras, comandos conflitantes nos *logs* de participantes são sobrescritos por comandos do próprio líder. Este processo é iniciado com a resposta da RPC de Propagação de Comandos, e consiste no líder buscar em seu próprio *log* o registro mais atual que esteja também contido no *log* do *Follower*, apagar todas as entradas inconsistentes no *log* do *Follower*, e transmitir todos os comandos contidos em seu *log* iniciando neste ponto.

Uma grande preocupação no desenvolvimento de sistemas de alta vazão é quanto a disponibilidade do serviço. E neste sentido, um ponto crítico para o Raft é a preocupação com a latência de rede na troca de mensagens entre nós do grupo. Se a troca de mensagens entre os participantes leva mais tempo que o tempo médio entre falhas no sistema, o algoritmo não será capaz de eleger um líder. Sem um líder estável por tempo suficiente para realizar acordos, o progresso do sistema é impossível.

A inequação abaixo (Ongaro and Ousterhout, 2014) demonstra o requerimento mínimo de tempo que deve ser satisfeito para garantir o funcionamento do protocolo:

$$tempoDeBroadcast << TimeoutDeEleição << MTBF$$

Nesta expressão, *tempoDeBroadcast* representa o tempo médio que servidores levam para propagar RPCs em paralelo aos outros participantes no grupo e processar suas respostas; *TimeoutDeEleição* representa o tempo limite que cada *Candidate* espera até inciar

um novo processo de eleição, caso não obtenha uma votação majoritária; e *MTBF* representa o tempo médio entre falhas para um único servidor.

Na Figura 5 é ilustrada a topologia do *log* desacoplado com o uso do Raft, ao considerar um grupo de três réplicas da aplicação e dois processos *loggers*. É possível observar que requisições de clientes (representados pelas setas contínuas) são encaminhadas a todas réplicas do grupo, sendo de responsabilidade do líder definir uma sequência única de comandos que são então encaminhados aos demais servidores (representado pelas setas pontilhadas), para que estes possam processar e responder às requisições dos clientes. Com esta abordagem, os *loggers* não executam os comandos entregues pelo protocolo de acordo, não sendo necessário a eles implementar a lógica de execução de comandos da aplicação.

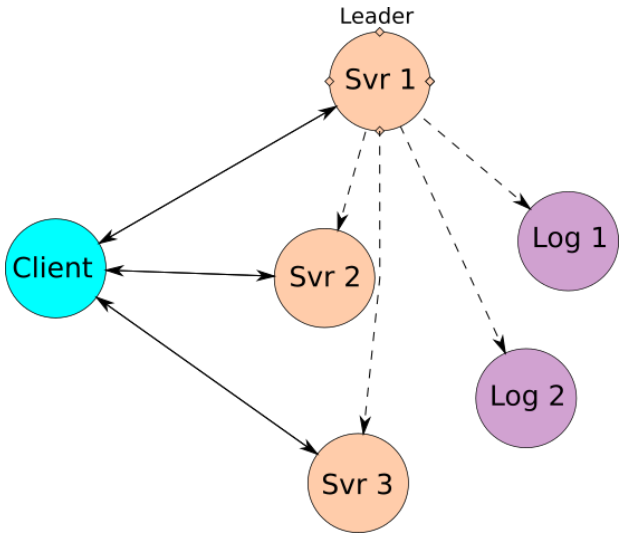

Figura 5: Topologia do *log* desacoplado com o uso do Raft.

5.2 Implementação do *Logger*

O serviço de *logger* implementado propõe-se a simplificar o desenvolvimento de aplicações SMR, abstraindo detalhes a respeito do gerenciamento do *log* de comandos a estes sistemas. Com relação ao projeto de engenharia e arquitetura, ao utilizar o serviço de *logger* desacoplado, desenvolvedores não precisam reescrever aplicativos do zero ou implementar rotinas de persistência otimizadas para manter o *log* de comandos de aplicações SMR. A interação do *logger* com as diferentes réplicas é realizada através de uma API (API – *Application Programming Interface*) muito simples, descrita a seguir.

- `join(ip)` Esta rotina é necessária para realizar a integração de processos *logger* à aplicação. Ao invocar `join`, a aplicação deve enviar como argumento o endereço IP da réplica *Leader*, já que a localidade desta é de conhecimento somente dos participantes do protocolo, sendo transparente à entidades externas. Ao receber

este endereço, o serviço de *logger* realiza uma requisição à máquina de estados da aplicação, utilizando a rotina de inserção de novos participantes por ela implementada. Este requerimento é necessário para que o *logger* seja capaz de receber mensagens do protocolo de acordo, já que no protocolo Raft novos participante são adicionados ao grupo de réplicas por um nodo *Leader*. Uma vez notificado de sua inserção no grupo de participantes, finalizando assim o *handshake*, o processo *logger* passa a ativamente receber comandos entregues pelo protocolo.

- `recover(i, n)` A rotina `recover` permite que as réplicas solicitem um intervalo de comandos ao *logger*. Geralmente, é necessário quando uma réplica em recuperação integra-se novamente ao sistema ou uma nova é iniciada, sendo assim necessário adquirir um intervalo de comandos perdidos para acompanhar o estado das outras réplicas. Os índices `i` e `n` representam um intervalo fechado de comandos no arquivo de *log*. O índice `i` corresponde ao primeiro comando a ser processado após a instalação de um *checkpoint* ou ao primeiro comando quando nenhum ponto de verificação está disponível, e `n` corresponde ao último comando que deve ser processado, uma vez que o próximo comando pode ser entregue pelo protocolo.

- `truncate(t)` A rotina de truncamento permite que uma réplica informe uma instância segura que pode ser usada pelos *logger* para remover um prefixo de comandos do *log* até o comando dado por `t`. Essa rotina é usada quando a maioria das réplicas corretas salva com êxito um estado de *checkpoint* que contém todas as atualizações executadas até o comando da instância `t`. Este procedimento truncará o *log* somente quando mais de $(n-1)/2$ réplicas solicitarem ao *logger* que o trunque. Nesse caso, o menor valor informado de `t`, representando o menor índice de comandos, será usado para truncar o *log*. O conhecimento desta rotina pela maioria de réplicas é necessário para garantir a correção do sistema. Por exemplo, é possível que um estado irrecuperável seja alcançado caso uma única réplica solicite o truncamento do *log* após a realização do *checkpoint* e falhe, resultando em um intervalo de comandos irrecuperáveis para recuperação dos demais participantes.

Na Figura 6 é possível observar um diagrama temporal ilustrando a troca de mensagens entre duas aplicações com o serviço de *logger* no decorrer do tempo. As chamadas à API são demonstradas em caracteres minúsculo, e eventos genéricos originados destas interações são exemplificados em caracteres maiúsculos. Detalhes sobre cada um destes eventos podem ser encontrados na Tabela 1. Pelo diagrama, é possível observar que o processo de *logger* é capaz de atender diferentes aplicações de maneira concorrente. Discussões sobre a correção desta execução e implicações de desempenho são apresentadas a seguir.

Na Seção 5.1 foi comentado sobre algumas limitações impostas à implementações práticas de SMR, de modo a satisfazer as propriedades do modelo. A respeito de res-

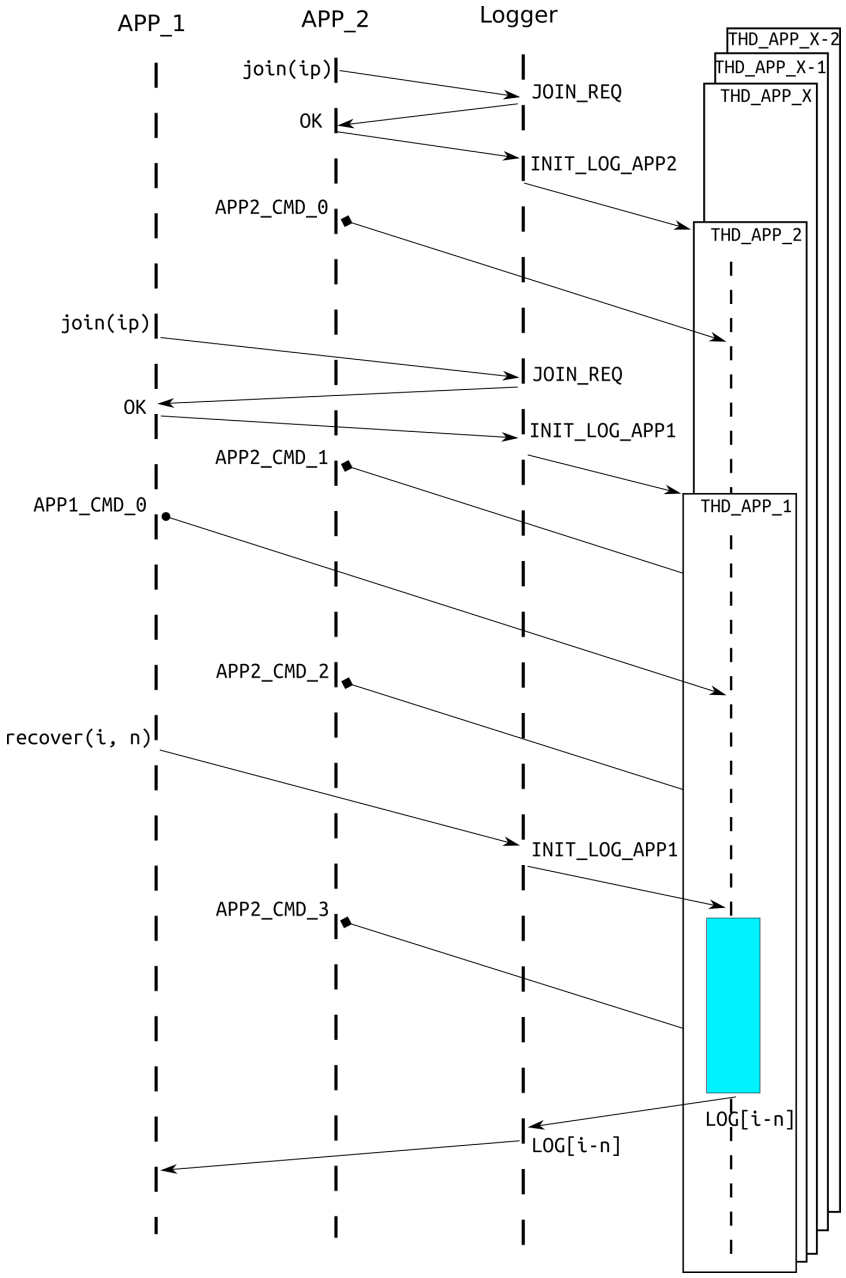

Figura 6: Análise temporal da comunicação do processo *Logger*.

tringir a execução *multithreaded* de sistemas SMR para preservar a propriedade de determinismo, na implementação do serviço de *logger* isso pode ser flexibilizado. Conforme ilustrado na Figura 6, uma *thread* [2] principal é responsável por atender chamadas à API. Ao receber uma requisição `join`, e estabelecer efetivamente o *handshake* de conexão, uma nova *thread* é criada para atender comandos entregues pelo protocolo de acordo comum às réplicas da aplicações requisitante. Este comportamento não viola a correção do modelo SMR, sendo garantido o isolamento do *log* de diferentes aplicações. O efeito de

[2]Em Go rotinas de execução concorrentes são denominadas *goroutines*, e seu perfil de execução difere-se consideravelmente de *threads* gerenciadas pelo *kernel* do sistema operacional. *Goroutines* consome menos recursos, iniciam com 4KB de memória *heap*, e são escalonadas para processadores lógicos pelo *go runtime*. A nomenclatura *thread* é utilizada nesta seção para simplificação.

Evento	Descrição
JOIN_REQ	Requisição encaminhada pelo processo *logger* em resposta à uma chamada de `join(ip)`, requisitando participação ao grupo do protocolo de consenso comum às réplicas da aplicação cliente.
OK	Resposta afirmativa encaminhada após uma chamada JOIN_REQ.
INIT_LOG_APP	Evento de inicialização de um novo procedimento de *log* para a aplicação requisitante.
APP_CMD	Entrega de comandos destinados à réplicas de uma mesma aplicação para o processo *logger* pelo protocolo de acordo.

Tabela 1: Eventos gerados durante a execução do processo *Logger*.

execução concorrente é transparente à clientes do serviço.

Este efeito de isolamento possibilita a utilização do serviço *logger* por soluções conhecidas na literatura. A biblioteca *R3Lib* (Pereira et al., 2019) apresentada na Seção 2.1.2 tem por objetivo facilitar o desenvolvimento de aplicações SMR abstraindo-se a utilização de protocolos de acordo. Em sua arquitetura, a biblioteca faz uso de uma *Camada de Consenso Única*, implementada de modo à compartilhar uma mesma instância de consenso entre diferentes aplicações. Este compartilhamento é vantajoso em cenários de recursos computacionais limitados, como ambientes virtualizados. Porém, como comandos originados de diferentes aplicações transitam sobre o mesmo protocolo de consenso, possuindo semânticas únicas a cada aplicação, a recuperação se torna um desafio. Do ponto de vista do protocolo de acordo, essa distinção entre a semânticas de comandos é desconhecida. Logo, para que uma aplicação recupere um *log* de comandos propostos durante a recuperação, seria necessário que esta conhecesse o intervalo de comandos com índices relativos ao protocolo de acordo utilizado, ou então realizasse a aquisição de todo arquivo de *log*, ignorando os comandos propostos por outras aplicações durante a recuperação. Com o uso do serviço de *logger* essa distinção entre comandos não se torna um problema, uma vez que ao invocar a rotina `recover` durante a recuperação, é garantido a entrega de comandos originados da mesma aplicação cliente.

A abordagem de desacoplamento apresentada é similar a estratégia de recuperação com o uso de um *helper process* (Clement et al., 2009). Procurando realizar a comunicação entre os processos da aplicação e *logger*, foram discutidas diversas alternativas como uso de *buffers* de memória compartilhada entre ambos processos, o que permitiria somente desacoplamento local. Como apresentado, na estratégia proposta a comunicação entre os diferentes processos se da através do próprio protocolo de acordo.

Isso é facilmente implementado com o Raft, pois este protocolo permite a participação de processos exercendo o papel de *non-Voter* no grupo de consenso, estes processos não participam do quórum de decisão e do processo de eleição de líder, somente recebem valores acordados pelo protocolo. O mesmo requerimento pode ser alcançado em implementações SMR que utilizem Paxos como mecanismo de consenso, necessitando somente que *loggers* exerçam o papel de *learners*.

5.3 Aplicações

Com o objetivo de avaliar o desempenho das propostas de redução do *log* de recuperação e execução do *log* desacoplado, foram desenvolvidas três aplicações de teste[3] que procuram simular aplicações distribuídas de alta vazão. Todas as aplicações implementam a técnica de SMR, utilizando Hashicorp Raft (Hashicorp, 2014) como protocolo de consenso. *Protocol Buffers*[4] são utilizados para possibilitar uma serialização de mensagens de uma maneira mais eficiente, reduzindo o tempo de interpretação e encaminhamento de requisições.

kvstore é a implementação de um serviço distribuído de banco de dados chave-valor com armazenamento em memória. Realiza a interpretação de requisições `set(k, v)`, `get(k)` e `delete(k)`, onde a chave `k` é representada por um número inteiro e o valor `v` por um vetor de *bytes* de tamanho configurável. Esta aplicação também é configurável para realizar a escrita de valores comprimidos, utilizando o algoritmo de compressão *gzip* especificado na RFC 1952.

diskstorage implementa um serviço de diretório, cujas entradas são mantidas em um armazenamento persistente. Este aplicativo representa um comportamento com uso intenso de E/S. O estado do aplicativo é totalmente representado por um arquivo de estado de 1GB, modificado por requisições `read(offset)` e `write(offset, v)`, em que `offset` descreve um registro no diretório e `v` é uma sequência de *bytes* com um tamanho fixo. Em nossos experimentos, o tamanho dos valores é definido como 1024. As atualizações de disco são comandos síncronos, que impõe uma imediata chamada ao sistema após cada atualização.

[3] Aplicações desenvolvidas estão disponíveis em: https://github.com/Lz-Gustavo/raft-demo
[4] Documentação disponível em: https://developers.google.com/protocol-buffers/

6 AVALIAÇÃO EXPERIMENTAL

A fim de validar a técnica de desacoplamento proposta, nesta seção são apresentados experimentos que exploram três métricas fundamentais, sendo estas:

1. *Impacto no desempenho nas réplicas durante a execução normal.* A atribuição do registro de comandos para o processo *logger* alivia a aplicação do custo computacional imposto por rotinas para persistência de dados. Porém, ao adicionar *loggers* no grupo de réplicas, alguma degradação no desempenho pode ser observada pelo protocolo de consenso. A análise compara a vazão da abordagem típica de *log* com a abordagem de desacoplamento proposta.

2. *Análise da velocidade relativa no processamento de comandos por réplicas e loggers.* Isso é analisado ao comparar a vazão das réplicas e dos *loggers* em tempo de execução, uma vazão menor do *logger* indicaria que este está atrasado frente à aplicação. Essa situação deve ser evitada, pois atrasaria a recuperação ou a adição de novas réplicas no sistema.

3. *Viabilidade em compartilhar o serviço de log.* O objetivo deste estudo é avaliar como o serviço de *log* distribuído é escalado com o número de aplicações.

6.1 Ambiente de Experimentação

Os experimentos foram realizados adotando uma topologia de máquinas Dell Power Edge 1435, equipadas com 2x Dual-Core AMD Opteron de 2GHz; 4GB de memória principal SDRAM DDR2, com taxa de transferência de 667MHz; e um disco de armazenamento persistente SATA II 7.2k RPM, com 500 GB e um *buffer* de 16 MB. Todos nodos são interconectados por um *switch* HP ProCurve 2920–48G gigabit.

6.2 Resultados

Em todos os experimentos apresentados nesta seção é adotada a configuração de três réplicas por aplicação, respeitando o fator de replicação definido no Capítulo 4. Nas

avaliações que exploram a utilização do serviço de *logger* desacoplado, são adicionados dois processos *logger* durante a configuração, isto é, antes que sejam executados os geradores de carga.

A Figura 7 mostra o gráfico de vazão média pelo 90 n-ésimo percentil da latência de *kvstore*, ao considerar 1.000.000 possíveis chaves e com valores de tamanho 1024 *bytes*. O número de clientes emulados varia de 1 a 19, com 3 clientes adicionados em cada ponto. Como esperado, a versão da aplicação sem que seja realizado nenhum registro de *log* apresenta o melhor desempenho, atingindo uma vazão máxima em torno de 2500 comandos/s e valores mais baixos de latência. No caso do *log* em nível da aplicação, onde esta é responsável pelo registro de comandos no dispositivo de armazenamento persistente, apresenta um desempenho um pouco melhor quando comparado à estratégia de *log* desacoplado. A adição de processos *logger* no protocolo de consenso pode incorrer em custos extras para a abordagem de *log* dissociada, ao considerar esta carga de trabalho.

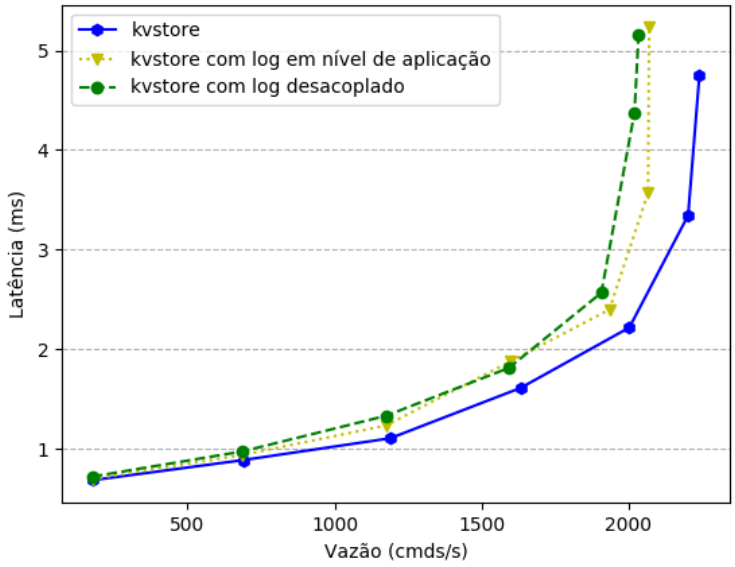

Figura 7: Saturação de *kvstore* com o desacoplamento.

Ao executar aplicações com cargas de trabalho intensivas em E/S, o registro de comandos aumenta a competição no acesso ao disco entre a execução e o registro dos próprios comandos. A Figura 8 mostra o gráfico da vazão versus latência para a aplicação *diskstorage*, também com valores de 1k *byte*. O número de clientes emulados varia de 1 a 7, com um cliente adicionado em cada ponto. Nesse caso, o desempenho da aplicação é severamente afetado, atingindo um ponto de saturação quando a vazão se aproxima de 280 comandos por segundo. No entanto, as diferenças entre as estratégias de exploração são menos perceptíveis. Até a configuração sem *logging* atinge uma vazão muito semelhante à obtida quando este está ativado.

Ao analisar estes resultados, é possível perceber que a adição de novas máquinas para

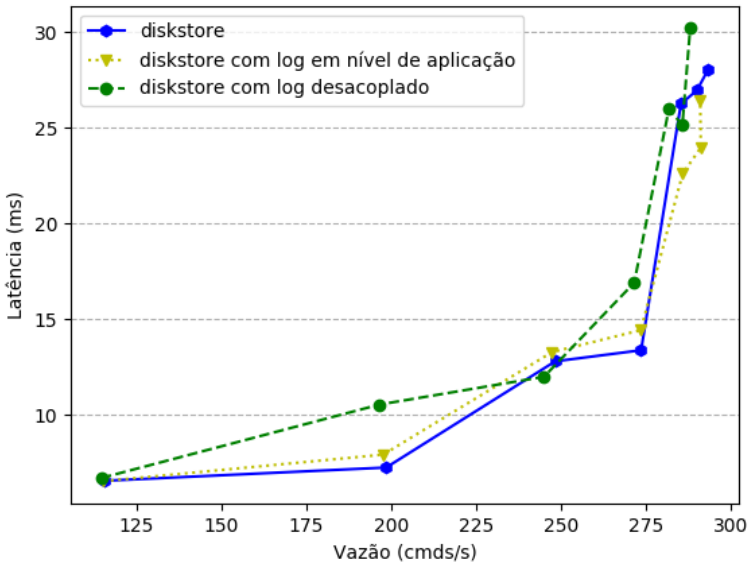

Figura 8: Saturação de *diskstorage* com o desacoplamento.

executar os processos *logger* pode introduzir uma pequena redução na vazão máxima que é observada, especialmente quando a carga de trabalho exige um maior uso de *CPU*. Esse custo provou-se insignificante para aplicações com E/S intensiva. Embora não exista nenhum benefício direto no uso da abordagem de *log* desacoplado para uma única aplicação, o compartilhamento de um único serviço de *log* entre vários aplicativos pode contribuir para um melhor uso dos recursos.

Para avaliar o impacto do *log* na infraestrutura compartilhada, são executadas várias instâncias das aplicações *kvstore* e *diskstorage* nos mesmos nós. Cada aplicativo hospedado é enviado para uma carga de trabalho de aproximadamente 70% da carga máxima. A Figura 9 mostra a variação da taxa de vazão de acordo com o número de aplicações *kvstore* independentes em execução no mesmo grupo de servidores. O eixo x indica o número de aplicações hospedadas, enquanto o eixo y mostra a vazão cumulativa, ou seja, o somatório da vazão média observada por cada aplicação individualmente. Como esperado, a maior vazão cumulativa é observada quando o *log* está desativado. À medida em que o número de aplicações hospedados aumenta, as abordagens de *log* em nível da aplicação e desacoplado experimentam uma diminuição na taxa de crescimento da vazão.

De maneira semelhante ao experimento anterior, é avaliado o impacto do *logger* em uma infraestrutura compartilhada executando a aplicação *diskstorage*. A Figura 10 mostra a vazão cumulativa para cenários com 1, 2, 3 e 4 aplicações hospedadas. Como pode ser observado, a vazão desempenhada pela aplicação sem *log* e a vazão da aplicação com a utilização do serviço de *log* desacoplado são semelhantes. Uma pequena redução na taxa de crescimento da vazão é observada quando o aplicativo é responsável pelos comandos de *log*. Essa redução de desempenho é uma consequência de uma competição maior para

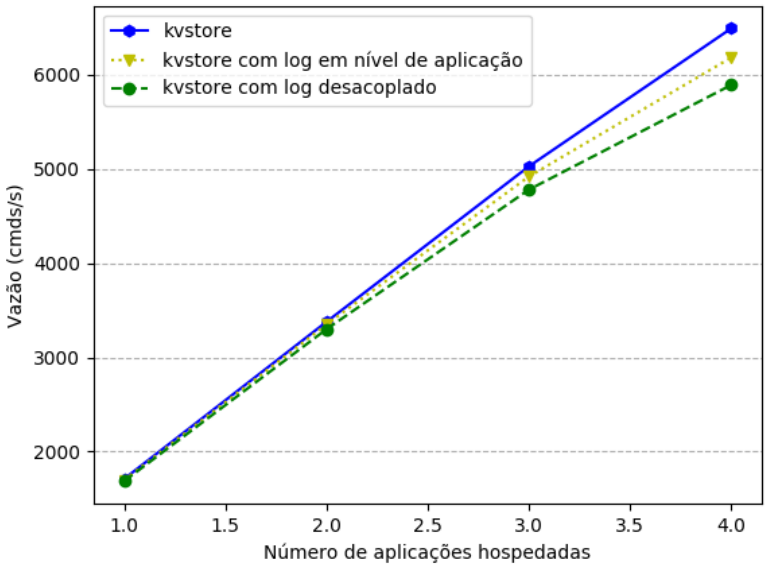

Figura 9: Escalabilidade do *Logger* ao atender aplicações *kvstore*.

que sejam realizadas escritas em disco, causada pela execução e pelo *log* de comandos executados por um número incremental de aplicações hospedadas.

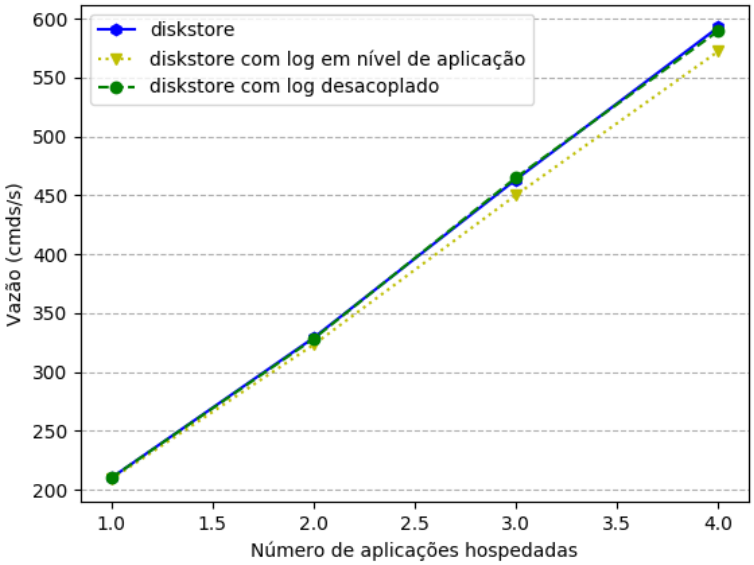

Figura 10: Escalabilidade do *Logger* ao atender aplicações *diskstorage*.

Até o momento, a comparação entre o *log* a nível da aplicação e o serviço desacoplado focou-se em avaliar como o rendimento da aplicação pode ser impactado pelas duas técnicas. Apesar da pequena redução na vazão para aplicações com uso intensivo de *CPU*, e do pequeno aumento na vazão para cargas de trabalho intensivas de E/S (especialmente quando vários aplicativos compartilham a mesma infraestrutura) observados com o

serviço de registro desacoplado, essa abordagem modular fornece uma solução escalável. As Figuras 9 e 10 resultam da plotagem para um máximo de 4 aplicações em uma mesma infraestrutura compartilhada, sendo este máximo número de aplicações hospedadas antes de que limitações dos próprios servidores sejam observadas. Quando esse limite é excedido, a vazão cumulativa não aumenta linearmente, no entanto, o serviço de *log* é capaz de continuar atendendo a mais aplicativos.

Para avaliar a escalabilidade do *log* desacoplado, é aumentado gradualmente o número de aplicações suportadas pelo serviço de *log*. Neste experimento, é delimitado um máximo de 3 aplicações por grupo de réplicas. A Figura 11 mostra a vazão cumulativa ao executar *kvstore* com registro desacoplado e a vazão acumulada do próprio *logger*. Incrementando até 9 o número de aplicações concorrentes, foi constatado que a aplicação e o *logger* mantiveram uma vazão muito semelhante, com uma diferença de 0,91%. Isso significa que o processo *logger* é capaz de lidar com essa carga sem degradação no seu desempenho. Com essa carga não foi possível identificar o ponto de saturação do *logger*. Espera-se que seja observada uma lenta deterioração na curva da vazão após um certo número de aplicações em execução, indicando que o *logger* estaria diminuindo a taxa de registro de novos comandos e, consequente, atrasando o processo de recuperação.

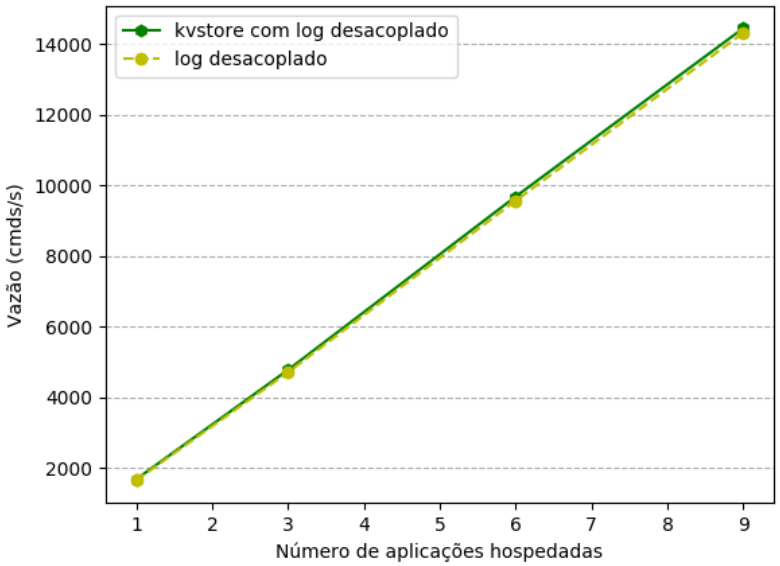

Figura 11: Escalabilidade e correção do serviço de *Log*.

Apesar do experimento de escalabilidade ser interrompido pela falta de servidores no ambiente de teste, experimentos adicionais foram conduzidos com o propósito de identificar quais seriam os limites de vazão da infraestrutura para projetar um possível ponto de saturação do serviço de *logging* desacoplado. Para isso, foram avaliadas as vazão máxima alcançada pela rotina de registro em *log* e pelo protocolo de consenso separadamente. Primeiro, foi reproduzida a carga de trabalho de *kvstore* registrando os comandos gerados

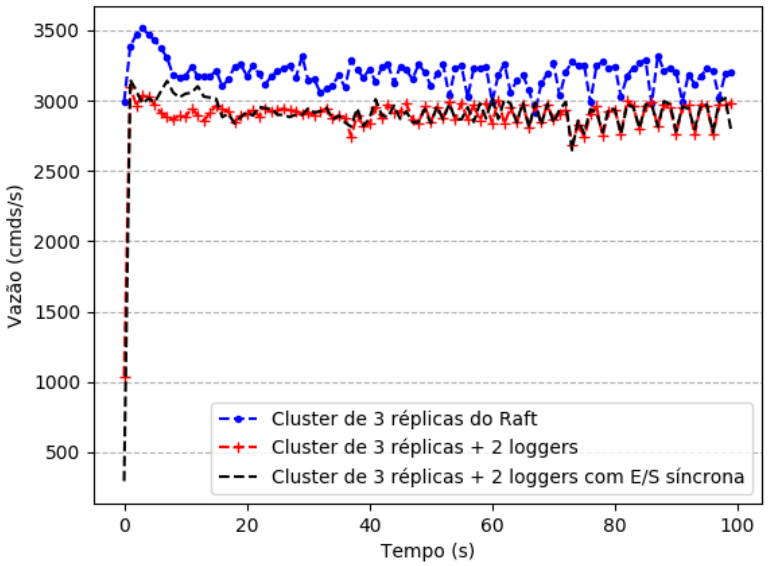

Figura 12: Vazão na entrega de comandos pelo protocolo Raft.

localmente, sem adição de custos de rede e pedidos. A vazão média observada para o *log* de comandos é de cerca de 46.000 comandos por segundo. Com 9 aplicações (conforme a Figura 11), o processo *logger* estava sujeito a 14.000 comandos/s, o que representa 30% da vazão máxima de registro.

Para estimar o máximo de comandos entregues pelo protocolo de consenso, foi implementada uma versão simplificada das aplicações SMR utilizadas. Esta aplicação utiliza Raft como algoritmo de consenso, e propõe continuamente comandos e réplicas simplesmente descartando os comandos entregues. Conforme ilustrado pela Figura 12, ao executar uma intensa carga de trabalho é observada uma média de 3.200 comandos sendo entregues por segundo. Testes anteriores geravam uma vazão média de 2.200 comandos por segundo, o que significa que sua vazão não era limitada pelo protocolo de consenso. Esses *benchmarks* não podem substituir uma análise detalhada da escalabilidade, mas fornecem algumas evidências de que o *logger* seria capaz de atender aplicações extras antes de prejudicar o desempenho. Com esses resultados, é possível perceber que os limites na taxa de escrita em disco e da entrega de comandos pelo algoritmo de consenso não foram excedidos durante nossos experimentos. Essa vazão expressa o número máximo de comandos do cliente que podem ser endereçados por um conjunto de réplicas.

Por fim, para avaliar o impacto do *log* desacoplado na recuperação das réplicas, é iniciada uma nova réplica durante a execução do teste. A nova réplica obtém o *log* completo de comandos emitindo uma solicitação de `recover`, e processa sequencialmente todos os comandos contidos no arquivo informado. A partir deste momento, a réplica é recuperada e se torna capaz de executar novos comandos propostos. Para comparar as estratégias de *log* tradicional e desacoplada, são medidos o tempo de recuperação e de instalação de

Execução(min)	Transferência(s)	Instalação(s)	Total(s)	Tamanho(MB)	Comandos
1	0.83	0.55	1.38	59.06	115360
3	2.88	1.48	4.36	179.51	349593
5	5.02	2.47	7.49	298.76	583881
7	6.08	3.42	9.50	416.17	812725
9	7.91	4.31	12.22	532.97	1039848

Tabela 2: Transferência de estado pela aplicação.

Execução(min)	Transferência(s)	Instalação(s)	Total(s)	Tamanho(MB)	Comandos
1	0.82	0.55	1.37	58.64	113932
3	2.54	1.45	4.00	175.63	343182
5	4.24	2.38	6.62	292.80	572239
7	6.30	3.38	9.68	408.06	797653
9	9.18	4.20	13.38	526.34	1028135

Tabela 3: Transferência de estado pelo *Logger*.

um novo estado para ambas as configurações, além de anotados o número de comandos e o tamanho do *log* obtido durante a recuperação.

Conforme demonstra a Tabela 2, o *log* contém cerca de 350.000 comandos após 3 minutos de execução do teste, o que corresponde a aproximadamente 180 MB de armazenamento. No uso do *log* pela aplicação o tempo gasto para transferir o *log* permanece próximo a 2,8 segundos, enquanto o tempo de processamento foi fixado em 1,5 segundos, o que resulta em aproximadamente 4,3 segundos para recuperar a réplica. Ao observar o mesmo tempo de execução na Tabela 3, que representa a utilização do *log* desacoplado, é constatada uma diminuição de $13,38\%$ no tempo de transferência as custas de um *log* $1,87\%$ menor. Este resultado não é observado em tempos de execução mais longos, onde no cenário de 9 minutos é notado um aumento de $15,18\%$ no tempo de transferência de estado pelo processo *logger*, mesmo com um estado menor. Com esses dados, não é possível concluir definitivamente se a técnica de *log* desacoplado é capaz de aliviar a sobrecarga do aplicativo durante a recuperação.

7 CONSIDERAÇÕES FINAIS

Este trabalho apresentou um serviço de *log* desacoplado da aplicação, com o objetivo de reduzir o custo do efeito de escrita de comandos em implementações de máquinas de estados. A ideia geral é adicionar processos leves, chamados *loggers*, ao conjunto de réplicas no sistema. Os *loggers* recebem exatamente a mesma sequência ordenada de comandos entregues às réplicas pelo protocolo de acordo, e são responsáveis por gerenciar e informar o *log* de comandos processados pelas réplicas durante sua recuperação. Esta monografia descreve a abordagem do *log* desacoplado, discute a correção da recuperação, sua implementação, e algumas implicações de desempenho dessa abordagem.

Do ponto de vista da programação, usando esse serviço, os desenvolvedores não precisam reescrever aplicativos do zero ou implementar rotinas de persistência otimizadas para manter o *log* de aplicações. O serviço de *logger* fornece uma API simples, permitindo que as réplicas de serviços recuperem o *log* durante recuperação de estado, ou o trunquem para eliminar um prefixo de comandos que não é mais necessários para restaurar um estado consistente.

De uma perspectiva de desempenho, além de aliviar a sobrecarga de réplicas de serviço com operações de E/S, os processos *logger* podem oferecer suporte a várias aplicações ao mesmo tempo. Experimentos apontam uma vazão média superior a 14.000 comandos atendidos pelo processo *logger* por segundo ao analisar 9 aplicações concorrentes, com uma diferença da vazão da aplicação inferior a 1%. O compartilhamento do *log* por diferentes aplicação resulta em um uso sustentável de recursos, o que é especialmente atraente para reduzir custos de provedores de serviços. Nesse sentido, o serviço de *log* desacoplado se torna uma boa alternativa aos serviços confiáveis, executados em ambientes compartilhados, como infraestruturas de nuvem.

7.1 Trabalhos Futuros

Durante o desenvolvimento do trabalho, algumas contribuições adicionais foram identificadas. Estes trabalhos representam possíveis extensões à ideia de *Log* desacoplado, fazendo uso desta abordagem para realizar procedimentos de maior custo computacional

sem interferir consideravelmente na execução de réplicas do sistema.

7.1.1 Execução do Serviço de *Logger* em Ambiente Compartilhado

Levando em consideração algumas soluções para oferecer confiabilidade em ambientes compartilhados mapeadas na literatura (Netto et al., 2018; Pereira et al., 2019), é possível incrementar tais configurações com a adição do serviço de *logging* desacoplado, estendendo os experimentos apresentados[1]. Conforme discutido na Seção 5.2, o serviço de *log* garante o isolamento da máquina de estados de diferentes aplicações atendidas concorrentemente, o que o torna viável para atender a recuperação em tais abordagens.

7.1.2 Compressão do *Log*

Esta abordagem tem por objetivo reduzir o número de operações contidas no arquivo de *log* para agilizar o processo de recuperação. Esta eliminação de operações não deve representar uma tarefa irrisória, uma vez que se faz necessário a implementação de mecanismos capazes de detectar possíveis dependências ou efeitos de sobrescrita para auxiliar este processo de truncamento, já que se faz necessário assegurar que o processamento deste *log* compactado alcance o mesmo estado consistente obtido com a execução do arquivo completo.

É possível que este processo de truncamento também evidencie uma pequena sobrecarga na execução normal da aplicação, as custas de possibilitar uma mais rápida recuperação de estado. Por isso, seria interessante analisar a execução e análise comparativa deste processo de compressão em três níveis de periodicidade:

- *Imediata:* Realizada logo após o registro de um novo comando. Paga-se uma maior sobrecarga na aplicação devido a realização de interrupções frequentes para o tratamento do arquivo em disco, porém possibilita uma recuperação mais rápida devido a disponibilidade imediata de um *log* já comprimido;

- *Intervalar:* Respeita uma periodicidade similar a um procedimento de *checkpoint*, por isso espera-se que desempenhe uma sobrecarga equivalente. Permite uma rápida recuperação na ocorrência de falhas logo após o procedimento de tratamento do *log*;

- *Adiantada:* Realizada somente com o início do processo de recuperação, esta abordagem prevê nenhuma interferência adicional além do procedimento de *logging* já realizado.

[1] Um estudo interessante seria realizar uma análise comparativa dos valores de latência observados por clientes da aplicação com e sem o uso do *logger*. Esse experimento tem por objetivo responder hipóteses apresentadas em (Einav, 2019).

7.1.3 Execução Paralela

A recuperação paralela de comandos do *log* é também uma alternativa para minimizar o tempo de recuperação de estado (Mendizabal et al., 2017b). Com a utilização de estruturas de dados capazes de rastrear eventuais dependências entre operações, como *Grafos Acíclicos Dirigidos* – DAGs, é possível criar subgrafos de operações dependentes através de uma redução transitiva. Vale ressaltar que os subgrafos gerados não são dependentes entre si, o que significa que independentemente da ordem adotada durante a execução paralela o efeito gerado será sempre determinístico do ponto de vista da aplicação.

Em (Abadi et al., 2016) DAGs são adotados para identificar possíveis sub rotinas de processamento paralelo em um dispositivo de processamento gráfico – GPU. Considerando-se cada nodo do grafo como uma operação tensorial que recebe n tensores como entrada e produz k tensores como saída, duas operações são consideradas dependentes caso operem sobre valores compartilhados de n e k, logo estas devem estar contidas na mesa sub rotina para não serem paralelizadas.

Uma contribuição importante seria estender a identificação de dependência para geração de arquivos de *log* idempotentes, que podem ser executados de forma paralela para possibilitar uma recuperação de estado mais eficiente. Acredita-se que esta abordagem, assim como a compressão do *log* como um todo, podem ser combinadas e implementadas em uma única biblioteca de *logging* para recuperação, de maneira a disseminar sua adoção em diversas aplicações SMR.

REFERÊNCIAS

Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., et al. (2016). Tensorflow: A system for large-scale machine learning. In *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, pages 265–283.

Alchieri, E., Dotti, F., Marandi, P., Mendizabal, O., and Pedone, F. (2018). Boosting state machine replication with concurrent execution. In *2018 Eighth Latin-American Symposium on Dependable Computing (LADC)*, pages 77–86. IEEE.

Bessani, A., Santos, M., Felix, J., Neves, N., and Correia, M. (2013). On the efficiency of durable state machine replication. In *Presented as part of the 2013 {USENIX} Annual Technical Conference ({USENIX}{ATC} 13)*, pages 169–180.

Bessani, A. N. and Alchieri, E. (2014). A guided tour on the theory and practice of state machine replication. In *Tutorial at the 32nd Brazilian symposium on computer networks and distributed systems*.

Budhiraja, N., Marzullo, K., Schneider, F. B., and Toueg, S. (1993). The primary-backup approach. *Distributed systems*, 2:199–216.

Burns, B., Grant, B., Oppenheimer, D., Brewer, E., and Wilkes, J. (2016). Borg, omega, and kubernetes.

Chaczko, Z., Mahadevan, V., Aslanzadeh, S., and Mcdermid, C. (2011). Availability and load balancing in cloud computing. In *International Conference on Computer and Software Modeling, Singapore*, volume 14.

Clement, A., Kapritsos, M., Lee, S., Wang, Y., Alvisi, L., Dahlin, M., and Riche, T. (2009). Upright cluster services. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 277–290. ACM.

Coulouris, G., Dollimore, J., Kindberg, T., and Blair, G. (2011). Distributed systems: Concepts and design.

Dinh, T. T. A., Liu, R., Zhang, M., Chen, G., Ooi, B. C., and Wang, J. (2018). Untangling blockchain: A data processing view of blockchain systems. *IEEE Transactions on Knowledge and Data Engineering*, 30(7):1366–1385.

Dwork, C., Lynch, N., and Stockmeyer, L. (1988). Consensus in the presence of partial synchrony. *Journal of the ACM (JACM)*, 35(2):288–323.

Einav, Y. (2019). Amazon found every 100ms of latency cost them 1% in sales. Disponível em: https://www.gigaspaces.com/blog/amazon-found-every-100ms-of-latency-cost-them-1-in-sales/. Acesso em: Dezembro de 2019.

Elnozahy, E. N., Alvisi, L., Wang, Y.-M., and Johnson, D. B. (2002). A survey of rollback-recovery protocols in message-passing systems. *ACM Computing Surveys (CSUR)*, 34(3):375–408.

Fischer, M. J., Lynch, N. A., and Paterson, M. S. (1982). Impossibility of distributed consensus with one faulty process. Technical report, Massachusetts Inst of Tech Cambridge lab for Computer Science.

Hadzilacos, V. and Toueg, S. (1994). A modular approach to fault-tolerant broadcasts and related problems. Technical report, Cornell University.

Hashicorp (2014). Repositório de código aberto do hashicorp raft. Disponível em: https://github.com/hashicorp/raft.

Herlihy, M. P. and Wing, J. M. (1990). Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3):463–492.

Hoare, C. A. R. (1978). Communicating sequential processes. In *The origin of concurrent programming*, pages 413–443. Springer.

Jhawar, R. and Piuri, V. (2017). Fault tolerance and resilience in cloud computing environments. In *Computer and information security handbook*, pages 165–181. Elsevier.

JoSEP, A. D., KAtz, R., KonWinSKi, A., Gunho, L., PAttERSon, D., and RABKin, A. (2010). A view of cloud computing. *Communications of the ACM*, 53(4).

Kapritsos, M., Wang, Y., Quema, V., Clement, A., Alvisi, L., and Dahlin, M. (2012). All about eve: execute-verify replication for multi-core servers. In *Presented as part of the 10th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 12)*, pages 237–250.

Kończak, J. Z., Wojciechowski, P. T., Santos, N., Żurkowski, T., and Schiper, A. (2019). Recovery algorithms for paxos-based state machine replication. *IEEE Transactions on Dependable and Secure Computing*.

Kotla, R. and Dahlin, M. (2004). High throughput byzantine fault tolerance. In *International Conference on Dependable Systems and Networks, 2004*, pages 575–584. IEEE.

Lamport, L. (1978). Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565.

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

Lamport, L., Shostak, R., and Pease, M. (1982). The byzantine generals problem. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 4(3):382–401.

Mendizabal, O. M., De Moura, R. S., Dotti, F. L., and Pedone, F. (2017a). Efficient and deterministic scheduling for parallel state machine replication. In *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 748–757. IEEE.

Mendizabal, O. M., Dotti, F. L., and Pedone, F. (2016). Analysis of checkpointing overhead in parallel state machine replication. In *Proceedings of the 31st Annual ACM Symposium on Applied Computing*, SAC '16, pages 534–537, New York, NY, USA. ACM.

Mendizabal, O. M., Dotti, F. L., and Pedone, F. (2017b). High performance recovery for parallel state machine replication. In *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*, pages 34–44. IEEE.

Netto, H. V., Luiz, A. F., Correia, M., de Oliveira Rech, L., and Oliveira, C. P. (2018). Koordinator: A service approach for replicating docker containers in kubernetes. In *2018 IEEE Symposium on Computers and Communications (ISCC)*, pages 00058–00063. IEEE.

Oki, B. M. and Liskov, B. H. (1988). Viewstamped replication: A new primary copy method to support highly-available distributed systems. In *Proceedings of the seventh annual ACM Symposium on Principles of distributed computing*, pages 8–17. ACM.

Ongaro, D. and Ousterhout, J. (2014). In search of an understandable consensus algorithm. In *2014 {USENIX} Annual Technical Conference ({USENIX}{ATC} 14)*, pages 305–319.

Pease, M., Shostak, R., and Lamport, L. (1980). Reaching agreement in the presence of faults. *Journal of the ACM (JACM)*, 27(2):228–234.

Pereira, P. M., Dotti, F. L., Meinhardt, C., and Mendizabal, O. M. (2019). A library for services transparent replication. In *Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing*, pages 268–275. ACM.

Popomaronis, T. (2016). Prime day gives amazon over 600 reasons per second to celebrate. Disponível em: https://www.inc.com/tom-popomaronis/amazon-just-eclipsed-records-selling-over-600-items-per-second.htm. Acesso em: Novembro de 2019.

Schneider, F. B. (1990). Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys (CSUR)*, 22(4):299–319.

Soltesz, S., Pötzl, H., Fiuczynski, M. E., Bavier, A., and Peterson, L. (2007). Container-based operating system virtualization: a scalable, high-performance alternative to hypervisors. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 275–287. ACM.

Turek, J. and Shasha, D. (1992). The many faces of consensus in distributed systems. *Computer*, 25(6):8–17.