Everaldo de Avila Gomes Junior

Redução do custo da durabilidade em Replicação Máquina de Estados através de *checkpoints* particionados

Brasil

2020

Everaldo de Avila Gomes Junior

Redução do custo da durabilidade em Replicação Máquina de Estados através de *checkpoints* particionados

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Computação da Universidade Federal do Rio Grande como requerimento para obtenção do grau de Mestre em Engenharia de Computação.

Universidade Federal do Rio Grande – FURG

Centro de Ciências Computacionais

Programa de Pós-Graduação em Computação

Orientador: Odorico Machado Mendizabal

Brasil

2020

Ficha Catalográfica

G633r Gomes Junior, Everaldo de Avila.
Redução do custo da durabilidade em Replicação Máquina de Estados através de *checkpoints* particionados / Everaldo de Avila Gomes Junior. – 2020.
86 f.

Dissertação (mestrado) – Universidade Federal do Rio Grande – FURG, Programa de Pós-Graduação em Computação, Rio Grande/RS, 2020.
Orientador: Dr. Odorico Machado Mendizabal.

1. Tolerância a Falhas 2. Replicação 3. Recuperação 4. *Checkpointing* I. Mendizabal, Odorico Machado II. Título.

CDU 004

Catalogação na Fonte: Bibliotecário José Paulo dos Santos CRB 10/2344

Everaldo de Avila Gomes Junior

Redução do custo da durabilidade em Replicação Máquina de Estados através de *checkpoints* particionados

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Computação da Universidade Federal do Rio Grande como requerimento para obtenção do grau de Mestre em Engenharia de Computação.

Prof.ª Dra. Patrícia Pithan de Araújo Barcelos

Prof.ª Dra. Cristina Meinhardt

Prof. Dr. Pedro de Botelho Marcos

Orientador Prof. Dr. Odorico Machado Mendizabal

Brasil
2020

Agradecimentos

Agradeço a todos os meus familiares, especialmente a meus pais Flávia e Everaldo por sempre terem me apoiado. Agradeço a meus amigos e colegas de laboratório por terem me ajudado e aconselhado sempre que precisei. Agradeço a minha namorada, Mariana, por estado sempre a meu lado durante este último ano.

Agradeço a todos os professores que contribuíram na minha formação, desde fundamental até durante o curso de pós graduação. Agradeço a meu orientador, Odorico, por ter me guiado e prestado toda atenção necessária para que este trabalho fosse concluído. Agradeço a banca avaliadora por ter contribuído para aperfeiçoamento deste trabalho. Por fim agradeço ao professor Eduardo Alchieri por ter me auxiliado durante momentos importantes deste trabalho.

"Ninguém pode ver nem compreender nos outros o que ele próprio não tiver vivido."
(Hermann Hesse)

Resumo

Replicação é uma técnica comumente utilizada para implementação de serviços tolerantes a falhas. Ao ser replicado, o serviço se mantém operando corretamente apesar da falha de uma quantidade limitada de réplicas. Para aumentar a disponibilidade, estratégias de recuperação se fazem necessárias, desta forma réplicas extras podem ser adicionadas durante a execução do serviço.

Replicação Máquina de Estados (RME) é um tipo de replicação bastante difundido para implementação de sistemas tolerantes a falhas. Em RME, todas as réplicas partem de um mesmo estado inicial e executam a mesma sequência determinística de comandos. Desta forma, as réplicas transitam pela mesma sequência de estados e produzem exatamente as mesmas respostas para cada requisição. Para garantir que réplicas faltosas recuperem-se ou novas réplicas sejam adicionadas ao sistema, estratégias de durabilidade como *logging*, *checkpointing* e transferência de estados devem ser implementadas nas réplicas do serviço. Enquanto processam as requisições, as réplicas registram os comandos executados em um *log* em armazenamento persistente, permitindo que réplicas faltosas obtenham uma sequência de comandos já executada. Para evitar que o *log* cresça indefinidamente, réplicas salvam periodicamente seus estados em armazenamento persistente (*checkpoint*) e removem do *log* os comandos cujas alterações estão refletidas no *checkpoint*. Apesar de melhorarem a disponibilidade, estratégias de durabilidade degradam a performance do serviço. Técnicas tradicionais de *checkpointing* exigem que a execução de novas requisições seja interrompida enquanto o estado é salvo. Esta sincronização é necessária para garantir consistência, porém a vazão do serviço é reduzida e a latência de resposta aumenta.

Neste trabalho é proposta uma estratégia que reduz a degradação do desempenho causada por *checkpointing* com base no particionamento do estado da aplicação. É apresentado um algoritmo onde diferentes *threads* de execução realizam *checkpointing* de determinadas partições em diferentes instantes de tempo. Desta maneira, durante o salvamento de estado, são impedidos de executar somente os comandos que operam sobre a partição que encontra-se em *checkpointing*. Requisições sobre as demais partições podem ser executadas normalmente. Através desta abordagem foi possível obter melhor desempenho durante a execução do serviço. A nova estratégia apresentou melhores resultados na vazão do serviço. Ainda, foi possível observar que a latência de resposta para as requisições realizadas durante a realização de *checkpoints* foi consideravelmente menor. A nova abordagem apresentou menor tempo necessário para recuperação do sistema.

Palavras-chaves: Tolerância a falhas. Replicação. Recuperação. Checkpointing.

Abstract

Replication is a technique commonly adopted on the development of fault-tolerant systems. By replicating a service, it remains accessible by clients despite the occurrence of faults in a bounded number of replicas. To increase availability, recovery strategies are required, so that extra replicas can be added to the system at run time.

State Machine Replication (SMR) is a well-known replication technique for implementing strong consistency across replicas. In SMR, every replica starts in the same state and executes a totally ordered sequence of deterministic commands. Thus, each replica evolves through the same sequence of state changes and produce exactly the same outputs for every request. In order to support recovery of faulty replicas or addition of new ones, replicas must implement durability strategies, such as logging, checkpointing and state transfer. While processing commands, replicas log these commands in a persistent storage, allowing new replicas retrieve a sequence of commands already executed. To avoid the continuos growth of logs, replicas periodically save their states and truncates their logs, removing those commands represented by the checkpoints. Althought durability approaches benefits overall availability, they hurt system's performance. Traditional checkpointing approaches interrupt the execution of incoming commands while service state is being saved. This synchronization is necessary to ensure consistence, but it causes a reduction in the throughput and latency increase.

This work proposes a strategy based on state partition to aleviate the performance degradation caused by checkpointing. The work presents a new algorithm that save states at different times. This way, during a checkpointing, new commands are blocked only if they access a partition being saved. Requests addressed to other partitions can be normally executed. By employing this new approach one can observe better performance during normal service execution. The new approach presented higher throughput. Moreover, it was noticed that the latency for those requests issued during the checkpointing execution where considerable low. The new approach presented lower time for system's recovery.

Key-words: Fault tolerance. Replication. Recovery. Checkpointing.

Lista de ilustrações

Figura 1 – Presença de falhas em um sistema replicado. 18
Figura 2 – Recuperação da réplica. 19

Figura 3 – Modelo clássico de Replicação Máquina de Estados 23
Figura 4 – Escalonamento e sincronização de comandos 24

Figura 5 – Modelo de falhas adotado no UpRight. 33

Figura 6 – Realização de *checkpoints* particionados 41
Figura 7 – Comando 41
Figura 8 – Comando 41
Figura 9 – Procedimento de recuperação de uma réplica. 46

Figura 10 – Relação entre *checkpoint* completo e particionado. 51
Figura 11 – Ponto de saturação do sistema para configuração com 4 *threads* de execução e carga de trabalho exclusivamente escrita. 54
Figura 12 – Vazão média para diferentes intervalos de *checkpoint*, usando a carga de trabalho 100r0c e 4 partições. 57
Figura 13 – Vazão média para diferentes intervalos de *checkpoint* utilizando CnE com a carga de trabalho 100r0c e 4 partições. 58
Figura 14 – Vazão média para diferentes intervalos de *checkpoint*, utilizando a carga de trabalho $0r0c$ e 4 partições. 59
Figura 15 – Vazão média para diferentes intervalos de *checkpoint*, usando a carga de trabalho $0r100c$ e 4 partições. 59
Figura 16 – Vazão média para diferentes intervalos de *checkpoint*, usando a carga de trabalho $0r0c$ e 8 partições. 60
Figura 17 – Vazão média para diferentes intervalos de *checkpoint*, usando a carga de trabalho $90r1c$ e 4 partições. 61
Figura 18 – Vazão média para diferentes intervalos de *checkpoint*, usando a carga de trabalho $90r1c$ e 8 partições. 62
Figura 19 – Latência de resposta para intervalo de *checkpoint* a cada 50000 comandos, utilizando a carga de trabalho $90r1c$ e 4 partições. 63
Figura 20 – Vazão do serviço ao longo do tempo com *checkpoints* a cada 50000 comandos, utilizando a carga de trabalho $90r1c$ e 4 partições. 63
Figura 21 – Recuperação utilizando *checkpoints* particionados. 65

Figura 22 – Vazão média para diferentes intervalos de *checkpoint* sem utilizar CnE - 100r0c 74

Figura 23 – Vazão média para diferentes intervalos de *checkpoint* utilizando CnE com a carga de trabalho 100r0c e 4 partições 75
Figura 24 – Vazão média para diferentes intervalos de *checkpoint* sem utilizar CnE - 90r1c 75
Figura 25 – Vazão média para diferentes intervalos de *checkpoint* utilizando CnE - 90r1c 76
Figura 26 – Vazão média para diferentes intervalos de *checkpoint* sem utilizar CnE - 50r50c 76
Figura 27 – Vazão média para diferentes intervalos de *checkpoint* utilizando CnE - 50r50c 77
Figura 28 – Vazão média para diferentes intervalos de *checkpoint* sem utilizar CnE - 0r50c 77
Figura 29 – Vazão média para diferentes intervalos de *checkpoint* utilizando CnE - 0r50c 78
Figura 30 – Vazão média para diferentes intervalos de *checkpoint* sem utilizar CnE - 0r100c 78
Figura 31 – Vazão média para diferentes intervalos de *checkpoint* utilizando CnE - 0r100c 79
Figura 32 – Vazão média para diferentes intervalos de *checkpoint* sem utilizar CnE - 0r0c 79
Figura 33 – Vazão média para diferentes intervalos de *checkpoint* utilizando CnE - 0r0c 80
Figura 34 – Vazão média para diferentes intervalos de *checkpoint* sem utilizar CnE - 100r0c 82
Figura 35 – Vazão média para diferentes intervalos de *checkpoint* utilizando CnE - 100r0c 82
Figura 36 – Vazão média para diferentes intervalos de *checkpoint* sem utilizar CnE - 90r1c 83
Figura 37 – Vazão média para diferentes intervalos de *checkpoint* utilizando CnE - 90r1c 83
Figura 38 – Vazão média para diferentes intervalos de *checkpoint* sem utilizar CnE - 50r50c 84
Figura 39 – Vazão média para diferentes intervalos de *checkpoint* utilizando CnE - 50r50c 84
Figura 40 – Vazão média para diferentes intervalos de *checkpoint* sem utilizar CnE - 0r50c 85
Figura 41 – Vazão média para diferentes intervalos de *checkpoint* utilizando CnE - 0r50c 85

Figura 42 – Vazão média para diferentes intervalos de *checkpoint* sem utilizar CnE - 0r100c 86
Figura 43 – Vazão média para diferentes intervalos de *checkpoint* utilizando CnE - $0r100c$ 86
Figura 44 – Vazão média para diferentes intervalos de *checkpoint* sem utilizar CnE - 0r0c 87
Figura 45 – Vazão média para diferentes intervalos de *checkpoint* utilizando CnE - 0r0c 87

Lista de tabelas

Tabela 1 – Resultados para 10 repetições do caso $90r1c$. 56
Tabela 2 – Simulação do número de partições envolvidas antes da execução de um *checkpoint* para configurações com 4 partições e carga de trabalho $90r1c$. 64
Tabela 3 – Tempo de recuperação para a estratégia tradicional. 65
Tabela 4 – Tempo de recuperação para a estratégia de *checkpointing* particionado. 66
Tabela 5 – Resultados dos experimentos utilizando 4 partições. 74
Tabela 6 – Resultados dos experimentos utilizando 8 partições. 81

Lista de abreviaturas e siglas

SMR	*State Machine Replication*
PSMR	*Parallel State Machine Replication*
RME	Replicação Máquina de Estados
RMEP	Replicação Máquina de Estados Paralela
CPU	*Central Process Unit*
CnE	Cópia na escrita
JVM	*Java Virtual Machine*
MB	*Megabyte*
SATA	*Serial Advanced Technology Attachment*
SSD	*Solid State Drive*

Sumário

1 Introdução 17
1.1 Motivação 18
1.2 Objetivos 20
1.3 Organização do trabalho 21

2 Replicação Máquina de Estados Paralela 22
2.1 Replicação Máquina de Estados Paralela com escalonamento antecipado . 23
2.2 Outros modelos de RMEP 25
2.2.1 CBASE 25
2.2.2 Escalonamento de lotes de comandos 26
2.2.3 EVE 26
2.2.4 P-SMR 27

3 Trabalhos Relacionados 28
3.1 Técnicas de *checkpointing* em RME 28
3.1.1 *Checkpointing* Sequencial 28
3.1.2 *Checkpointing* em P-SMR 29
3.1.3 *Checkpointing* em CBASE 29
3.1.4 *Checkpointing* em Eve 30
3.1.5 Recuperação rápida e transferência de estados sob demanda 30
3.1.6 *Checkpointing* do protocolo *Zeno* 31
3.1.7 Recuperação proativa em sistemas tolerantes a falhas bizantinas . . 31
3.1.8 Recuperação da biblioteca de replicação *UpRight* 33
3.2 Técnicas de recuperação em outros modelos de replicação 34
3.2.1 Recuperação no protocolo *Spinnaker* 34
3.2.2 Checkpointing em SiloR 35

4 Técnica para *checkpointing* particionado 36
4.1 Modelo do sistema 36
4.2 Modelo de execução das réplicas 37
4.2.1 Algoritmos de execução das réplicas em RMEP 37
4.3 *Checkpointing* particionado 39
4.3.1 Algoritmo para mapeamento de conflitos 42
4.3.2 Algoritmo para armazenamento de *checkpoints* particionados 43
4.4 Recuperação utilizando *checkpoints* particionados 44
4.4.1 Algoritmo de recuperação 46

4.5 Discussão 47

5 Avaliação Experimental 50
5.1 Implementação do protótipo 50
5.2 Ambiente de experimentação 53
5.3 Caracterização da carga de trabalho utilizada 53
5.3.1 Cargas de trabalho 54
5.4 Avaliação de desempenho 55

6 Conclusão 67
6.1 Trabalhos Futuros 68

Referências 69

Apêndices 73

APÊNDICE A Resultados para testes com 4 partições 74

APÊNDICE B Resultados para testes com 8 partições 81

1 Introdução

Atualmente os serviços providos através da Internet possuem rigorosos requisitos de disponibilidade (BURROWS, 2006; GUNARATHNE; QIU; FOX, 2011; WANG et al., 2018). Falhas nos sistemas atuais podem causar instabilidade no serviço ou ainda a perda de dados (DEAN, 2009). Períodos de indisponibilidade em serviços de escala global podem acarretar a perda de milhares de dólares. Por exemplo, uma falha no sistema de auto escalabilidade da Amazon fez com que o serviço de vendas fosse afetado durante o "Amazon Prime Day". Durante o evento, a empresa registrou um total de USD$ 4.2 bilhões em vendas durante 36 horas de evento, ou USD$ 116 milhões por hora (CNBC, 2018). Uma maneira de aumentar a disponibilidade de um serviço é através da replicação. Uma arquitetura altamente disponível possui componentes replicados operando em paralelo para garantir que um serviço mantenha-se ininterruptamente responsivo durante um determinado período de tempo, mesmo que alguns componentes possam falhar.

Replicação é uma técnica que visa prover alta disponibilidade de dados ou serviços, a despeito da ocorrência de um certo número e tipo de falhas. Replicação Máquina de Estados[1] (RME) (SCHNEIDER, 1990) é uma técnica de replicação bastante difundida para desenvolvimento de sistemas tolerantes a falhas (BURROWS, 2006; HUNT et al., 2010; CORBETT et al., 2012). Isto se deve à simplicidade da técnica e à garantia de consistência forte (HERLIHY; WING, 1990). Neste modelo de replicação, as requisições enviadas pelos clientes são ordenadas (por exemplo, utilizando um protocolo de consenso (LAMPORT et al., 2001; ONGARO; OUSTERHOUT, 2014)) e entregues para as réplicas do serviço na mesma ordem. Além disso, as réplicas executam as requisições respeitando a ordem de chegada e a execução das requisições é determinística, ou seja, o resultado da requisição depende apenas do estado atual do sistema e dos argumentos de entrada da requisição. Desta maneira, é garantido que todas as réplicas do serviço atravessem a mesma sequência de estados, assegurando a consistência entre si. Porém, esta estratégia não aproveita o paralelismo das arquiteturas com múltiplos núcleos de processamento atuais. Para aliviar esta desvantagem, foram propostas abordagens que buscam melhorar o desempenho da técnica (KOTLA; DAHLIN, 2004; KAPRITSOS et al., 2012; MARANDI; PEDONE, 2014; ALCHIERI et al., 2017), permitindo que requisições cujas execuções não modifiquem dados em comum possam ser executadas em paralelo.

Através do paralelismo na execução de comandos independentes é possível atender a uma maior quantidade de requisições por unidade de tempo (BESSANI; SOUSA; ALCHIERI, 2014; ALCHIERI et al., 2018). Por exemplo, em (ALCHIERI et al., 2018) através de um mecanismo de classes de conflito entre comandos é possível identificar quais

[1] Neste trabalho é utilizada como tradução da Língua Inglesa para *State Machine Replication*.

requisições podem ser executadas em paralelo. Já a estratégia apresentada em (KOTLA; DAHLIN, 2004) observa o conjunto de dados manipulados por diferentes operações para decidir se estas requisições podem executar em paralelo.

1.1 Motivação

Embora estratégias de replicação permitam que falhas sejam mascaradas, é necessário que um quórum de réplicas corretas esteja em execução para que o serviço se mantenha operacional. Portanto, para aumentar a disponibilidade, mecanismos de recuperação devem permitir que réplicas faltosas retornem à execução corretamente ou permitir a inserção de novas réplicas no sistema. Caso contrário, eventualmente todas as réplicas do serviço podem tornar-se indisponíveis.

A Figura 1 ilustra um serviço replicado, onde réplicas R_0 a R_3 implementam o modelo RME e, portanto, executam a mesma sequência de requisições. Apesar da falha da réplica R_0 imediatamente após a execução da requisição e_1 (retângulo $exec(e_1)$), as demais réplicas executam as requisições e_1 e e_2. Com isso, os clientes do serviço não serão afetados pela falha em R_0. Porém, eventualmente outras réplicas do serviço podem tornar-se indisponíveis. Para garantir a existência de um quórum de réplicas corretas e o progresso na execução de requisições dos clientes, técnicas de recuperação devem ser implementadas. Neste exemplo, note que a réplica R_0 retoma a execução a partir de um procedimento de recuperação e aumenta o número de réplicas corretas no sistema.

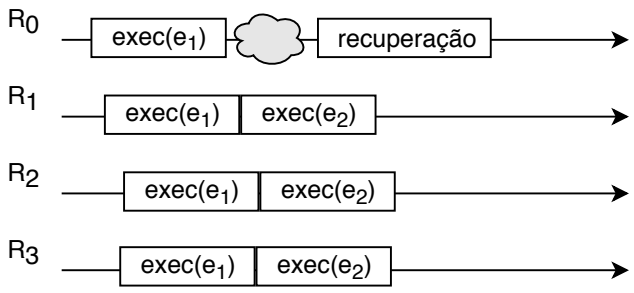
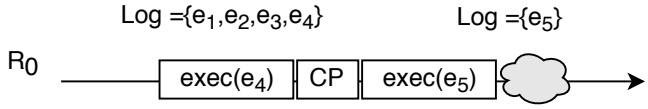


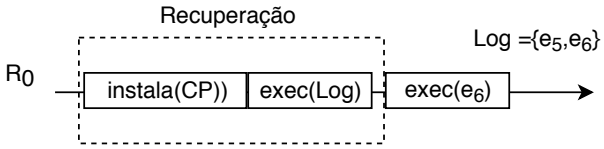
Figura 1 – Presença de falhas em um sistema replicado.

A implementação de procedimentos de recuperação combina técnicas usadas para preservar a durabilidade do estado do serviço, como *logging*, *checkpointing* e transferência de estados (ELNOZAHY et al., 2002; BESSANI et al., 2013). Em cenários suscetíveis a falhas, as réplicas do serviço registram suas operações em um *log* para que, em uma eventual falha de uma réplica, esta réplica possa restaurar um estado consistente reprocessando o *log* de operações obtido de uma réplica correta. Contudo, este registro pode tornar-se muito grande, fazendo com que tanto o armazenamento de requisições quanto a recuperação torne-se inviável. Para evitar o crescimento indefinido de *logs*, as réplicas do serviço devem salvar seus estados periodicamente em uma unidade persistente e truncar o *log* contendo

operações anteriores ao salvamento do estado. Este mecanismo é chamado de *checkpointing* e é ilustrado na Figura 2. Durante a execução normal, uma réplica executa uma requisição e a registra no *log* de comandos (e.g., requisição e_4). Em momentos específicos, a réplica executa o procedimento de *checkpointing*, suspendendo a execução de novos comandos durante esta ação (retângulo CP na Figura 2a). Em seguida, o *log* é truncado e novas requisições podem ser processadas e adicionadas ao *log*, como ilustrado pela requisição e_5. Após a ocorrência de uma falha (nuvem na Figura 2a), a réplica executa um procedimento de recuperação, conforme ilustrado pela Figura 2b. Durante este procedimento a réplica busca o *checkpoint* mais atual (CP), que pode estar armazenado em sua unidade permanente ou obtido de uma outra réplica do serviço e processa o *log de operações*, restaurando o estado a partir de alguma réplica correta do sistema.



(a) Execução anterior a falha



(b) Estado após a recuperação

Figura 2 – Recuperação da réplica.

Checkpointing pode ser feito de diferentes maneiras. Por exemplo, a aplicação se encarrega de realizar o armazenamento do estado ou deixa a cargo do programador decidir quando o *checkpoint* é feito (RANDELL, 1975). Pode ainda ser feito de maneira transparente, isto é, não depende da intervenção da aplicação ou do programador. O sistema realiza o *checkpoint* automaticamente através de uma política previamente definida e também se recupera após a ocorrência de uma falha (LAMPORT, 1998; BESSANI et al., 2013). Esta estratégia alivia a complexidade da implementação do sistema, pois o programador não precisa se preocupar com tarefas complexas e propensas a erros para implementação de tolerância a falhas (ELNOZAHY et al., 2002).

Entretanto, a realização *checkpoints* causa degradação no desempenho do serviço, uma vez que novas requisições não podem ser atendidas durante a execução do procedimento de *checkpointing* (CASTRO; LISKOV, 2000). Apesar da técnica de RME ser bastante adotada na literatura (ZHENG et al., 2014; WANG, 1993; WANG; FUCHS, 1993; TAMIR; SEQUIN, 1984), poucas estratégias foram sugeridas para aliviar a sobrecarga causada pelos mecanismos de durabilidade (BESSANI et al., 2013; MENDIZABAL; DOTTI; PEDONE, 2017; CLEMENT et al., 2009; HUNT et al., 2010).

1.2 Objetivos

Este trabalho tem como *objetivo principal* apresentar uma estratégia que reduz a sobrecarga causada pela execução de procedimentos de *checkpointing* durante a execução normal do serviço. A ideia geral é permitir que novas requisições sejam executadas em paralelo ao processo de *checkpointing* através do particionamento do estado da aplicação. Desta forma, podem ser armazenados em dispositivo de armazenamento persistente fragmentos do estado da aplicação em diferentes instantes de tempo, possibilitando que operações que modifiquem as demais partições executem em paralelo.

Para alcançar este objetivo e permitir uma avaliação sobre o ganho esperado, são descritos os seguintes *objetivos específicos*.

- Fazer um levantamento das implementações de Replicação Máquina de Estados existentes na literatura e discutir os mecanismos de *checkpoint* presentes em cada uma dessas técnicas. Em especial, pretende-se fazer revisão aprofundada sobre estratégias de *checkpointing* otimizadas para desempenho. O Capítulo 3 apresenta os principais trabalhos identificados;

- Propor um algoritmo para *checkpointing* particionado, de forma que partes do estado da aplicação sejam salvas em instantes diferentes. Dessa forma, enquanto uma partição do estado do serviço é salva, comandos que acessam as demais partições podem ser executados sem a interrupção completa do serviço. O Capítulo 4 apresenta a descrição em detalhes desta abordagem;

- Efetuar otimizações e avaliar o efeito de parametrizações do algoritmo, permitindo observar como o número de partições, a frequência de *checkpoints* e a carga de trabalho das aplicações podem afetar o desempenho dos serviços em execução;

- Avaliar o desempenho da técnica proposta através da implementação de protótipo de serviço de alta vazão implementado com RME. Para implementação de RME, será utilizada a biblioteca de replicação BFT-SMaRt (BESSANI; SOUSA; ALCHIERI, 2014) com sua variante para RMEP proposta em (ALCHIERI et al., 2017). Pretende-se comparar o impacto na vazão e latência do sistema causados pela técnica de *checkpointing* particionado e pela estratégia de *checkpointing* tradicional;

- Além de avaliar a redução de sobrecarga e, consequentemente, ganho de desempenho na execução normal do sistema, pretende-se avaliar a redução no tempo de recuperação de réplicas com o uso de *checkpoints* salvos por partição;

- Uma vez que o modelo de RMEP apresentado em (ALCHIERI et al., 2017) será usado para avaliação de desempenho, a técnica de *checkpointing* proposta será integrada à implementação de RMEP discutida em (ALCHIERI et al., 2017).

1.3 Organização do trabalho

O restante deste trabalho está organizado da seguinte forma. O Capítulo 2 introduz conceitos fundamentais sobre RME e RMEP. No Capítulo 3 são apresentados os trabalhos relacionados. O Capítulo 4 descreve a técnica de *checkpointing* particionado. No Capítulo 5 são discutidos os impactos no desempenho com o uso da técnica proposta, através de avaliação experimental. Os Apêndices A e B complementam os resultados avaliados. Finalmente, o Capítulo 6 conclui o trabalho.

2 Replicação Máquina de Estados Paralela

Replicação Máquina de Estados é uma técnica utilizada para implementação de serviços tolerantes a falhas, através da replicação do serviço (servidores) e coordenação das interações dos clientes com as réplicas do serviço (SCHNEIDER, 1990). O serviço é definido como uma máquina de estados, que consiste de um conjunto de variáveis que compõe o seu estado e um conjunto de operações que operam sobre este estado. Cada comando é executado de maneira atômica, modificando as variáveis de estado ou produzindo uma resposta ao cliente.

Este modelo de replicação provê aos clientes a abstração de um serviço altamente disponível, mantendo transparente a existência de replicação. Os clientes enviam suas requisições para todas as réplicas do serviço e estas, através de algum protocolo de ordenação, recebem e executam todas as requisições em uma mesma ordem. A execução dos comandos acontece de maneira determinística, isto é, o estado resultante obtido após a execução de um comando depende dos dados acessados ou modificados pelo comando e do estado anterior à execução do comando. Para garantir que as réplicas do serviço produzam a mesma sequência de estados, as réplicas devem partir de um mesmo estado inicial e devem executar a mesma sequência de comandos enviados pelos clientes. Assim sendo, se as réplicas do serviço partem de um mesmo estado inicial e executam os comandos exatamente na mesma ordem as réplicas produzirão as mesmas modificações em seus estados, garantindo consistência entre todas as réplicas.

A Figura 3 ilustra o modelo de Replicação Máquina de Estados, onde os comandos são gerenciados por um *proxy cliente*. Este *proxy* realiza o *broadcast* das requisições dos clientes a todas as réplicas do serviço e, em seguida, aguarda pela resposta de uma das réplicas, descartando as respostas repetidas. As requisições são enviadas aos *proxies servidores*, que entregam estas requisições para todas as réplicas. A camada do protocolo de consenso (LAMPORT, 1998; ONGARO; OUSTERHOUT, 2014) é responsável por ordenar as requisições dos clientes, garantindo uma ordem total na entrega de comandos entre as réplicas do serviço.

Apesar de garantir a consistência do estado das réplicas através da sequencialização de comandos, o modelo tradicional de RME não permite que comandos sejam executados em paralelo, pois tal execução poderia gerar inconsistências. Contudo, foi observado que é possível executar comandos independentes sem violar a consistência (SCHNEIDER, 1990). Para isto, foram propostas estratégias que permitem que requisições sejam executadas em paralelo, através da observação entre a dependência de comandos (KOTLA; DAHLIN, 2004; KAPRITSOS et al., 2012; MARANDI; BEZERRA; PEDONE, 2014; ALCHIERI et

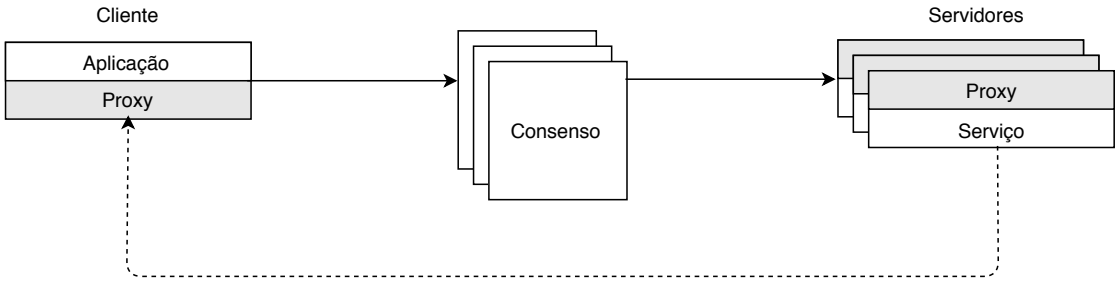


Figura 3 – Modelo clássico de Replicação Máquina de Estados

al., 2018), originando o modelo de Replicação Máquina de Estado Paralela (RMEP).

A seguir, são apresentados o modelo de RMEP proposto em (ALCHIERI et al., 2017), que será utilizado para a implementação das réplicas deste trabalho, e outras abordagens para implementação de RMEP. Embora a estratégia de *checkpointing* proposta neste trabalho seja construída sobre a implementação de (ALCHIERI et al., 2017), a técnica pode ser generalizado para outros modelos de RMEP.

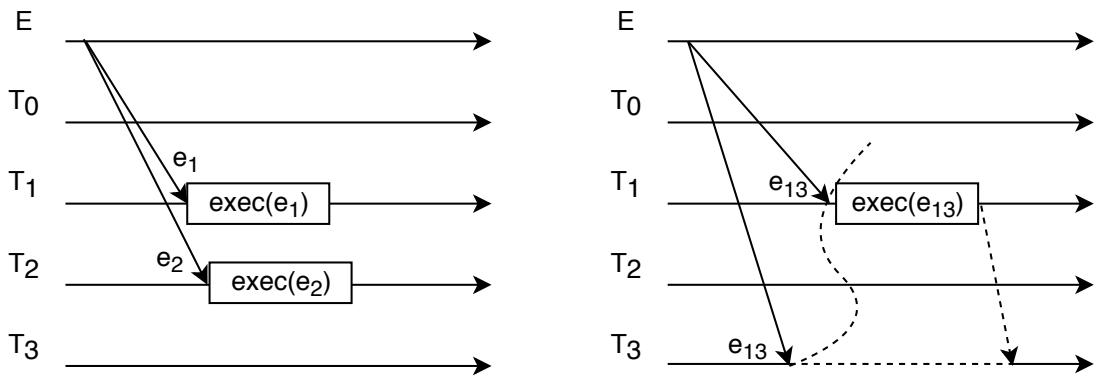
2.1 Replicação Máquina de Estados Paralela com escalonamento antecipado

Em (ALCHIERI et al., 2017) é apresentado um modelo de RMEP baseado em *classes de conflito* para determinar a concorrência entre comandos durante a execução do serviço. Neste modelo, cada requisição enviada pelos clientes pertence a uma única classe de conflito. As réplicas do serviço possuem uma *thread escalonadora* responsável por despachar as requisições para as demais *threads* de execução do serviço. As requisições são inseridas nas filas das *threads* conforme a classe de conflito associada a cada requisição.

Uma classe de conflito define se um comando deve ser executado de maneira sequencial ou concorrente e quais *threads* são responsáveis pela execução da requisição. As requisições enviadas pelos clientes pertencem ao conjunto de requisições $\mathcal{R}$, onde cada requisição $r_i = \langle c\_id, op, data, tipo \rangle$ contém o número da sua instância dentre as requisições entregues pelo protocolo de consenso (c_id), a operação a ser executada (op), um conjunto de argumentos ($data$), e está associada a uma classe de conflito ($tipo$). O conjunto das classes de conflito é definido por $\mathbb{C} \rightarrow \{Seq, Cnc\} \times \mathcal{P}(\mathcal{T})$, onde $\{Seq, Cnc\}$, indicam se uma requisição deve ser executada de maneira sequencial (Seq), ou concorrente (Cnc), e $\mathcal{P}(\mathcal{T})$ representa o conjunto das partes[1] de $\mathcal{T} = \{0, 1, 2, ..., n-2, n-1\}$, onde $\mathcal{T}$ é o conjunto de *threads* executoras do serviço.

[1] O conjunto das partes refere-se à tradução para *power set*, do inglês, e é um conjunto formado por todos os subconjuntos de um conjunto de referência.

Para ilustrar o modelo de execução, considere que as réplicas implementam um banco de dados chave-valor esta aplicação poderia ser particionada de forma que cada partição contenha uma das tabelas do serviço implementado. As requisições enviadas pelos clientes podem escrever ou ler valores das tabelas do serviço. Cada comando c é composto pela tupla $\langle op, table[], key[], values[], tipo \rangle$[2], onde op indica qual operação será executada envolvendo uma ou mais tabelas ($table[]$), chaves ($key[]$) e valores ($values[]$). O $tipo$ indica a qual classe de conflito a requisição está associada. Por exemplo, o comando $e_1 = \langle write, [1], [4], [32], W1 \rangle$ deve ser inserido na fila de execução da *thread* T_1, conforme a Figura 4a, enquanto o comando $e_2 = \langle write, [2], [10], [45], W2 \rangle$ é executado pela *thread* T_2 em paralelo. A escolha do escalonador pelas *threads* T_1 e T_2, neste exemplo, se dá pela informação das classes de conflito $W1 = \langle Seq, \{1\} \rangle$ e $W2 = \langle Seq, \{2\} \rangle$, que indicam que atualizações exclusivamente nas partições 1 ou 2, devem ser sequencializadas entre si, porém, atualizações independentes em cada uma das partições podem ocorrer em paralelo.



(a) Comando não sincronizante (b) Comando sincronizante

Figura 4 – Escalonamento e sincronização de comandos

Este modelo de execução de RMEP permite que um comando opere sobre mais de uma partição. Neste caso, é necessário que as *theads* executoras responsáveis pelas partições sincronizem para a execução do comando. Para isto, a *thread* escalonadora insere o comando nas filas de execução de todas as *threads* associadas à classe de conflito deste comando. Ao receber o comando, as *threads* envolvidas aguardam em uma barreira pelas demais. Em seguida, uma única *thread*, por exemplo a *thread* de menor *id*, executa o comando e libera as demais *threads* para seguirem com a execução. Suponha o comando $e_{13} = \langle write, [1,3], [4,1], [32,40], W13 \rangle$ onde são inseridas chaves nas tabelas 1 e 3, e $W13 = \langle Seq, \{1,3\} \rangle$. Conforme a Figura 4b, a *thread* escalonadora (E) insere o comando e_{13} nas filas das *threads* T_1 e T_3. A *thread* T_1 percebe que é a *thread* de menor *id* com base na classe de conflito associada ao comando e_{13}, então a *thread* T_1 aguarda a *thread* T_3 em uma barreira (linha pontilhada sobre as linhas de execução das *threads* T_1 e T_2). A *thread* T_3 aguarda enquanto a *thread* T_1 executa o comando e_{13}, conforme o traço pontilhado na

[2] Neste exemplo foi omitido o campo referente à instância de consenso.

linha de execução da *thread* T_3. Ao executar o comando e_{13} a *thread* T_1 libera a *thread* T_3 para que prossigam com a execução normal (indicado pela seta com linha pontilhada). Mais detalhes sobre o funcionamento deste modelo de RMEP são apresentados na Seção 4.2, com a descrição dos algoritmos que implementam o escalonador e *threads* executoras.

2.2 Outros modelos de RMEP

Nesta seção são discutidas outras abordagens para implementação de RMEP propostos na literatura. Todos os modelos de RMEP baseiam-se na ideia de extrair uma relação de dependência entre os comandos entregues às réplicas. As diferenças se dão principalmente na forma como estas dependências são representadas e avaliadas pelas réplicas. Algumas técnicas baseiam-se em escalonador, escalonamento estático definido antecipadamente (como o modelo discutido anteriormente) ou abordagens especulativas, sem escalonamento. Através da análise de dependências, é possível que comandos independentes, isto é, que não operam sobre o mesmo conjunto de dados, sejam executados em paralelo pelas réplicas do serviço.

2.2.1 CBASE

Em (KOTLA; DAHLIN, 2004), é apresentado um modelo de RMEP que relaxa o modelo de ordenação clássico de RME, isto é, a *ordem total* na execução de comandos, para um modelo de *ordenação parcial*. Para isto, a estratégia utiliza um paralelizador que recebe, através do protocolo de consenso, uma sequência de comandos totalmente ordenados, e determina a dependência entre os comandos com base na semântica da aplicação.

O paralelizador cria um grafo de dependências direcionado e acíclico para anotar as dependências entre as requisições recebidas e que estão aguardando para serem executadas. As requisições são representadas por vértices e as arestas indicam a dependência entre dois comandos. Para determinar se uma nova requisição é dependente de alguma requisição contida no grafo, o paralelizador utiliza uma matriz de conflitos, que indica, primeiramente, se duas operações são conflitantes e, em segundo momento, avalia se os argumentos das requisições são conflitantes. Como exemplo, imagine um sistema de arquivos, onde dois comandos *ls*, que são comandos de leitura, não conflitam entre si. Porém, um comando *ls* e *rm* podem conflitar, dependendo do *caminho* associado aos comandos. Se os comandos operam sobre caminhos diferentes, eles podem ser considerados independentes. Porém, se eles operam sobre o mesmo caminho, a ordem de execução entre estes dois comandos precisa ser preservada entre todas as réplicas. Neste caso, haverá uma aresta conectando estes dois comandos no grafo de dependência.

Para permitir a execução paralela de comandos, o sistema é equipado com um *pool* de *threads* executoras, que são encarregadas de executar e remover os comandos executados

do grafo de dependência. Com o uso do grafo de dependências, esta abordagem é capaz de prover um escalonamento ótimo, uma vez que qualquer comando independente pode ser executado, bastando haver uma *thread* disponível para executá-lo. Entretanto, o grafo pode se tornar um gargalo de desempenho, visto que tanto o paralelizador quanto as *threads* executoras precisam acessar esta estrutura em exclusão mútua.

2.2.2 Escalonamento de lotes de comandos

O modelo de RMEP apresentado em (MENDIZABAL et al., 2017) evita o uso de grafo de dependência e propõe um escalonador que despacha lotes de comandos para as *threads* executoras do serviço. Para representar conflitos entre diferentes lotes, a técnica utiliza um mapa de *bits*, que codifica as variáveis lidas ou escritas pelos comandos contidos em cada lote. Cada *thread* executora mantém um mapa de bits que indica quais variáveis são acessadas pelos lotes armazenados em suas filas. Dessa forma, ao receber um novo lote, o escalonador verifica se este lote tem bits coincidentes com os mapas de bits de alguma *thread* executora. Não havendo conflito, o lote pode ser despachado para qualquer *thread*, e o mapa de bits desta *thread* é atualizado para incluir também a codificação de conflitos deste novo lote. Caso contrário, se houver bits coincidentes com alguma *thread*, o lote precisa ser adicionado na fila das *threads* conflitantes, para garantir a ordem de execução entre comandos que acessam mesmas chaves. A eliminação do grafo de comandos remove um elemento central de contenção entre as *threads* e o escalonador. O uso de mapa de bits para verificação de conflitos também apresenta-se como uma estratégia eficiente, pois o custo de operações bit-a-bit é baixo. Contudo, agrupar comandos em lotes pode limitar o grau de paralelismo na execução de comandos, visto que basta que apenas um comando de um novo lote conflite com algum comando de outro lote, que todos os comandos deste novo lote serão sequencializados em relação ao lote mais antigo.

2.2.3 EVE

Para que requisições sejam executadas em paralelo, a abordagem proposta em (KAPRITSOS et al., 2012) agrupa requisições em *lotes* e as réplicas do serviço executam os lotes de requisições possivelmente em diferentes ordens. No modelo tradicional de RME, as réplicas participantes do protocolo de consenso primeiramente decidem uma ordem de execução de requisições e, em seguida, processam as requisições conforme a ordem definida. Na abordagem proposta em (KAPRITSOS et al., 2012), primeiramente as réplicas executam os lotes de requisições de forma especulativa e, ao fim da execução, as réplicas verificam se a execução causou inconsistência entre seus estados, através de um estágio de verificação. Caso réplicas divirjam sobre o estado após a execução de um conjunto de lotes, elas decidem assumir o estado obtido pela maioria das réplicas. Caso não seja possível

chegar a um consenso sobre qual estado deve ser assumido, as réplica retrocedem seus estado e executam os lotes de requisições em uma ordem determinística.

Para minimizar o efeito da divergência entre as réplicas, a estratégia realiza um procedimento de *mescla*. Este procedimento utiliza informações específicas da aplicação para agrupar as requisições de maneira que, ao executarem lotes de requisições, as réplicas dificilmente divirjam.

2.2.4 P-SMR

Para permitir a execução de requisições de maneira concorrente, o modelo de RMEP apresentado em (MARANDI; BEZERRA; PEDONE, 2014) paraleliza tanto a entrega quanto a execução de comandos. Em vez de enviar uma única sequência de requisições para as réplicas do serviço, é utilizado o protocolo de consenso *Multi-Ring Paxos* (MARANDI; PRIMI; PEDONE, 2012), que entrega tantas sequências de requisições quanto o número de *threads* executoras do serviço.

As *threads* executoras do serviço operam em dois modos, *sequencial* e *paralelo*. Durante a execução em modo sequencial, um conjunto de *threads* executoras coordenam-se para execução de um comando. Em modo paralelo, todas as *threads* executoras processam requisições recebidas pelo seu canal de requisições de forma independente.

Nessa abordagem, o estado do sistema pode ser particionado e requisições destinadas a uma determinada partição são sempre enviadas para uma *thread* específica. Já os comandos que envolvam acesso à múltiplas partições precisam ter a execução entre as *threads* sincronizada, logo são enviados para o canal cuja execução dos comandos é realizada de maneira sequencial.

3 Trabalhos Relacionados

Técnicas de recuperação tradicionais confiam na restauração do estado a partir do reprocessamento de comandos processados por outras réplicas do sistema, de modo a restaurar um estado consistente do serviço. Para isso, normalmente são mantidos um registro de comandos processados (o *log* de recuperação) e *checkpoints*, que salvam uma imagem do estado, possibilitando a remoção dos comandos do *log* até o instante no qual a imagem do estado do serviço foi salva. Este capítulo faz um levantamento sobre técnicas presentes na literatura que visam reduzir custos com durabilidade e acelerar a recuperação ou inserção de réplicas no sistema.

3.1 Técnicas de *checkpointing* em RME

3.1.1 *Checkpointing* Sequencial

Em (BESSANI et al., 2013), é proposta uma estratégia de *checkpointing* que explora a redundância e a ordem total de requisições recebidas, inerentes aos serviços replicados com RME. Um serviço implementando RME realiza progresso sempre que um quórum de réplicas concorda sobre a ordem do próximo comando a ser executado. Assim, o comando é executado e o próximo comando a ser executado pelas réplicas é escolhido. Normalmente, ao processar um determinado número de comandos, as réplicas criam um novo *checkpoint*. Embora as réplicas avancem independentemente, na prática, a velocidade relativa entre as réplicas na execução dos requisições é praticamente a mesma. Dessa forma, é muito provável que várias réplicas estejam efetuando o salvamento de estado, através da execução de seus *checkpoints* aproximadamente no mesmo período, causando um atraso na entrega e no processamento de novos comandos. Consequentemente, observa-se uma queda na vazão do sistema durante a execução de *checkpoints*.

Com base nesta observação, os autores propõem uma estratégia na qual cada réplica do serviço realiza *checkpoint* em um instante de tempo diferente, garantindo que haja um quórum mínimo de réplicas disponíveis para garantir o progresso na escolha de novos comandos a serem processados. Para isso, o protocolo proposto assume uma quantidade fixa, por exemplo n operações de escritas que são realizadas entre cada *checkpoint*, acrescido de um intervalo que caracterize uma defasagem entre as réplicas. Os autores obtiveram ganho na vazão durante a execução normal do sistema, pois a cada intervalo entre *checkpoints*, apenas uma réplica interrompe a sua execução para salvar o estado da aplicação, enquanto outras réplicas seguem executando normalmente.

Esta estratégia difere da técnica proposta pois não existe particionamento no estado

da aplicação. O desempenho foi melhorado através da defasagem do *checkpoint* entre as réplicas do serviço, sendo a técnica apresentada neste trabalho inspirada nesta abordagem. Na técnica de *checkpoitning* particionado, cada réplica do serviço realiza *checkpoint* de uma partição diferente a cada intervalo de *checkpoints*, isto é, a defasagem não se dá por réplicas independentes, mas por partições de uma mesma réplica.

3.1.2 *Checkpointing* em P-SMR

O modelo de execução proposto em (MARANDI; BEZERRA; PEDONE, 2014) paraleliza não só o processamento dos comandos como a entrega de requisições pelo protocolo de consenso. Para este modelo de RMEP, foram propostas duas estratégias para *checkpointing*, descritas em (MENDIZABAL et al., 2014). A primeira delas é realizada de maneira coordenada, isto é, para que um *checkpoint* seja realizado, as réplicas do serviço devem atingir um mesmo estado em comum. Na segunda abordagem, as réplicas realizam *checkpoints* de maneira independente.

Para garantir que as réplicas produzam a mesma sequência de *checkpoints*, a abordagem coordenada utiliza um comando de *checkpoint*. Este comando é gerado pelo protocolo de consenso e executado dentro da sequência de requisições processadas pelas réplicas do serviço. Já na abordagem não coordenada a execução da requisição de *checkpoint* é realizada em instantes de tempo diferentes, pois esta é gerada por cada réplica do serviço e não é ordenada pelo protocolo de consenso.

Na abordagem de *checkpoints* particionados, a requisição de *checkpoints*, assim como no modelo apresentado, é gerada periodicamente por cada réplica do serviço. Contudo, por existir somente uma sequência de requisições entregue pelo protocolo de consenso não existe divergência entre os estados criados pelas réplicas do serviço. Outra vantagem da estratégia de *checkpoints* particionados é a possibilidade de ser utilizada independente do modelo de RMEP diferentemente da estratégia apresentada.

3.1.3 *Checkpointing* em CBASE

Em (KOTLA; DAHLIN, 2004) as réplicas do serviço recebem requisições através de um processo *paralelizador*. O paralelizador é responsável por serializar a execução de comandos dependentes e garantir que sua ordem de execução respeite a ordem em que as requisições foram recebidas. Comandos independentes podem ser processados em paralelo, pelo conjunto de *threads executoras*. Devido ao sequenciamento das requisições, criado pelo protocolo de consenso, todas as réplicas produzem os mesmos estados, consequentemente produzem os mesmos *checkpoints*.

Embora os autores não apresentem uma abordagem de *checkpoint* para este modelo de RMEP, o sistema permite a definição de comandos que conflitam com qualquer

outro comando no grafo. Portanto, o *checkpoint* pode ser executado como um comando sincronizante com todos os demais, conforme discutido em (MENDIZABAL; DOTTI; PEDONE, 2017). Dessa forma, durante a execução de um *checkpoint*, é garantido que todos os comandos anteriores a esse evento tenham sido executados pelas *threads* executoras. Nesta estratégia, *checkpoints* podem ser criados a cada intervalo fixo e determinístico, por exemplo, a cada n requisições executadas desde o último *checkpoint*.

3.1.4 *Checkpointing* em Eve

A estratégia de *checkpointing* apresentada em (KAPRITSOS et al., 2012) é muito semelhante a estratégia convencional para *checkpoints* descrita no início do capítulo. Réplicas evoluem através do processamento de comandos e, a cada intervalo entre *checkpoints*, que pode ser dado por intervalos de tempo ou número de comandos, realiza o *checkpoint*. Entretanto, nesta estratégia as requisições são agrupadas em lotes.

Esta abordagem, assim como as demais descritas nesta seção, precisam interromper a execução de novos comandos enquanto salvam o estado da aplicação. O trabalho proposto por nesta dissertação bloqueia apenas a execução de requisições envolvendo as partições do estado sendo salvas no momento. Portanto, a abordagem de *checkoints* particionados possibilita que a vazão de cada réplica seja menos impactada pelo procedimento de *checkpointing*.

3.1.5 Recuperação rápida e transferência de estados sob demanda

Em (MENDIZABAL; DOTTI; PEDONE, 2017), são propostas duas estratégias para redução do tempo de recuperação. A primeira delas, *speedy recovery*, baseia-se na observação da dependência entre as requisições registradas no *log* de comandos a serem reprocessados e novas requisições. Caso não haja dependência entre uma nova requisição com os comandos armazenados no *log*, a nova requisição pode ser executada em paralelo com o processamento das requisições contidas no *log*. Com esta abordagem, réplicas em recuperação podem antecipar a execução de alguns comandos novos, desde que estes não afetem o estado a ser reconstituído a partir do *log* de recuperação. Anteriormente ao reprocessamento dos comandos do *log*, a réplica precisa restaurar um estado a partir de um *checkpoint* recente.

A abordagem permite que o *checkpoint* seja instalado por partes, conforme necessidade. Com essa estratégia, uma partição do estado da aplicação, representada por um segmento do *checkpoint*, é instalada na primeira vez que uma requisição opera sobre o espaço de dados deste segmento.

De forma semelhante ao nosso trabalho, em (MENDIZABAL; DOTTI; PEDONE, 2017) os autores salvam as partições do estado em arquivos separados, possibilitando a

transferência e instalação dos mesmos em paralelo. Entretanto, os autores não planejam a execução do salvamento destas partições de forma defasada, o que permitiria a execução de comandos que não envolvam partições em salvamento. Apesar de ambas as técnicas possibilitarem a transferência e instalação de *checkpoints* em paralelo, o salvamento de partições de forma defasada, como proposto nesta dissertação, é menos danoso para a vazão das réplicas durante a execução normal.

3.1.6 *Checkpointing* do protocolo *Zeno*

Zeno (SINGH et al., 2009) é um protocolo para Replicação Máquina de Estados tolerante a falhas Bizantinas. Neste protocolo, a garantia de consistência forte é substituída pelo modelo de consistência eventual. O protocolo utiliza dois quóruns distintos: *quórum forte*, consistindo de $2f + 1$ réplicas; e *quórum fraco*, consistindo de $f + 1$ réplicas. Zeno executa através de uma sequência de configurações, chamadas visões (*views*). Em cada *view*, uma réplica primária é responsável por atribuir, em ordem crescente e monotônica, os números de sequência das requisições enviadas pelos clientes.

Para que o registro de comandos de atualização do estado do protocolo não cresça indefinidamente, *checkpoints* devem ser realizados periodicamente. Quando uma réplica r recebe uma requisição do primário, a réplica verifica se foram executadas n operações desde o último *checkpoint*. Em caso positivo, a réplica faz *broadcast* de uma mensagem de *commit* às demais réplicas. Assim que uma réplica recebe $2f + 1$ respostas ao *commit* da mensagem de *checkpoint*, então a réplica envia uma mensagem de *checkpoint* para todas as réplicas. Ao receber $f + 1$ mensagens de *checkpoint*, o estado é considerado estável e todas as requisições com números de sequência menores que este *commit* são descartadas (truncamento do *log*).

Na abordagem apresentada, é possível que uma operação de *checkpoint* não seja executada dentro de um limite de tempo, pois é necessário que existam pelo menos $f + 1$ réplicas do serviço em execução. Na estratégia de *checkpoints* particionados, as requisições de *checkpoint* são criadas por cada réplica do serviço e executadas de forma independente.

3.1.7 Recuperação proativa em sistemas tolerantes a falhas bizantinas

Em (CASTRO; LISKOV, 2000), é apresentado um sistema para Replicação Máquina de Estados (LAMPORT, 1978; SCHNEIDER, 1990) que oferece integridade e alta disponibilidade na presença de falhas Bizantinas. O sistema continua operando corretamente mesmo que algumas réplicas tenham sido comprometidas por um atacante (um processo bizantino), ou simplesmente apresentem comportamentos arbitrários. O sistema tolera falhas, prevendo que, no máximo, 1/3 das réplicas apresentem comportamento falho.

O trabalho descreve algumas técnicas necessárias para que seja possível recuperar o sistema na presença de falhas Bizantinas:

1. ***Recuperação Proativa.*** Uma réplica sob efeito de falhas Bizantinas aparenta um comportamento correto mesmo tendo sido corrompida por um atacante. O algoritmo recupera as réplicas periodicamente, independente de qualquer mecanismo de detecção de falhas. Portanto, basta que alguma réplica apresente um estado divergente da maioria das réplicas, que esta terá o seu estado restaurado com base no quórum de réplicas corretas. Além disso, a recuperação proativa é feita de modo que não mais de 1/3 das réplicas do serviço sejam interrompidas.

2. ***Transferência de Estado eficiente.*** Na estratégia de replicação proativa, réplicas frequentemente restauram seus estados com base em um quórum de réplicas corretas, como forma de remover estados potencialmente impactados por processos maliciosos. Por esta razão, a eficiência deste processo é crucial para que seja possível realizar recuperações com frequência. Para recuperar uma réplica, o mecanismo de transferência de estado verifica o estado da réplica armazenado e determina quais porções do estado estão atualizadas e não corrompidas. Em seguida, deve garantir que o estado recebido de outras réplicas está correto. Para isso, os autores desenvolveram um mecanismo de transferência hierárquico, baseado em *hash* encadeado e criptografia incremental (BELLARE; MICCIANCIO, 1997), que permite modificação dos estados durante a transferência.

O protocolo de recuperação é responsável por fazer réplicas defeituosas retornarem ao seu comportamento correto. Para isto, o protocolo garante, que após a recuperação, a réplica possua um estado atualizado e consistente.

A recuperação é feita de maneira proativa, iniciando periodicamente sempre que um monitor (*watchdog*) dispare. O monitor de recuperação salva em dispositivo de armazenamento persistente o estado da réplica, isto é, o *log* de comandos e uma imagem do serviço. Em seguida, reinicia o processo e a réplica retoma a sua execução a partir do estado armazenado.

É importante que o mecanismo de transferência seja eficiente, pois ele é necessário para que uma réplica se atualize durante a recuperação. Os pontos chave para garantir esta eficiência são a redução das informações a serem transferidas e a carga imposta sobre as réplicas. Este mecanismo deve se responsabilizar pela confiabilidade do estado entregue. O mecanismo utiliza particionamento hierárquico (em árvore) para reduzir a quantidade de informações transmitidas. A partição raiz corresponde ao estado completo do serviço e cada nodo corresponde a uma partição e são divididos em sub-partições s, de mesmo tamanho.

Cada réplica, mantém para cada *checkpoint*, uma cópia da árvore de partições. Uma nova cópia é criada quando um *checkpoint* é feito e descartada quando o *checkpoint* se torna estável. A árvore de um *checkpoint* guarda uma tupla $\langle lm, d \rangle$ para cada partição de meta-dados e uma tupla $\langle lm, d, p \rangle$ para cada página, onde lm é o número de sequência do *checkpoint*, d é o resumo da partição e p é o valor da página. As cópias da árvore de partições contém somente as tuplas modificadas, reduzindo o espaço e tempo necessário para manter estes *checkpoints*.

3.1.8 Recuperação da biblioteca de replicação *UpRight*

UpRight (CLEMENT et al., 2009) é uma biblioteca para implementação de RME. Sistemas construídos sob esta biblioteca podem tolerar falhas Bizantinas. UpRight é configurado de forma a tolerar um número de falhas por omissão maior do que um número de falhas bizantinas. Além disso, os autores distinguem as falhas bizantinas em dois grupos, conforme a Figura 5, descrevendo o comportamento falho em falhas por omissão que incluem falhas por colapso (*crash*), e falhas de comissão (*comission*), que representam os comportamentos como o envio de mensagens fora da especificação do protocolo.

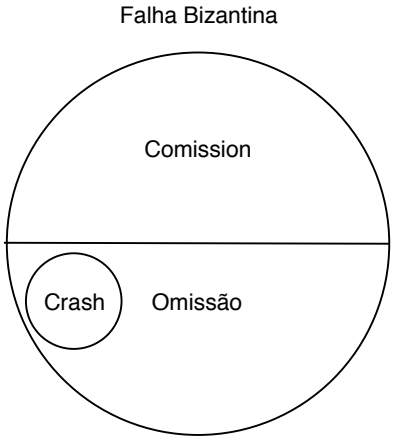


Figura 5 – Modelo de falhas adotado no UpRight.

A biblioteca é construída com o objetivo de garantir que o sistema seja correto apesar de k falhas por *comission* e qualquer número de falhas por omissão. UpRight deve garantir que o sistema esteja disponível durante intervalos de tempo suficientemente longos onde estão presentes u falhas nas quais, no máximo, k são falhas por *comission* e o restante são falhas por omissão.

A arquitetura de um sistema UpRight segue o modelo cliente-servidor. A aplicação do cliente envia requisições para a biblioteca de replicação que retorna respostas a estas requisições. A aplicação do servidor recebe requisições, executa o processamento e envia respostas ao cliente. Cada processo servidor ou cliente é composto pela própria aplicação,

uma interface (*glue*) entre a aplicação e o mecanismo de comunicação (*shim*) entre os componentes do sistema UpRight.

Para a realização de *checkpoints*, a interface de comunicação entre a aplicação e os outros componentes do sistema solicita que a aplicação do servidor armazene periodicamente o seu estado em unidade permanente. São permitidas três estratégias diferentes para realização de *checkpoint*. A primeira é a mais tradicional, ou seja, o servidor para completamente o processamento de requisições enquanto o estado está sendo armazenado. Na segunda estratégia, duas instâncias da aplicação do servidor recebem a mesma sequência de requisições. Na instância *primária*, a realização de *checkpoint* é ignorada, enquanto na segunda instância (*helper*), o retorno de respostas a estas requisições é desativado, permitindo que o nodo realize *checkpoint* enquanto novas requisições estão sendo processadas. A última estratégia realiza cópia-na-escrita, que permite aos processos manterem cópias de suas estruturas de dados, enquanto o *checkpoint* é realizado os efeitos das requisições são aplicados a esta cópia.

A estratégia de *checkpointing* com processo auxiliar permite a execução de novas requisições durante o salvamento do estado. Esta abordagem, assim como a proposta nesta dissertação, melhora a vazão do sistema durante a execução normal das réplicas. Entretanto, no UpRight o número de processos no sistema é duplicado. Com relação à entrega de mensagens para o processo primário e auxiliar, ou é necessário duplicar os canais de comunicação, garantindo que cada processo leia mensagens de um canal independente, ou estes processos precisam acessar mensagens de uma mesma fila, o que ocasiona sincronização adicional para coordenação destes processos. A abordagem de *checkpoints* cópia-na-escrita também permite o processamento de novas requisições em paralelo com o salvamento do estado. Neste trabalho, a técnica de cópia-na-escrita foi avaliada e seu desempenho foi comparado com a técnica de *checkpoints* particionados e é possível perceber que, apesar do paralelismo adicionado, há impacto na vazão do sistema, especialmente em sistemas de alta vazão, onde uma grande quantidade de informação precisa ser copiada durante o processo de salvamento do estado.

3.2 Técnicas de recuperação em outros modelos de replicação

Nesta seção são apresentadas técnicas de recuperação implementadas em diversos modelos de sistemas replicados, não estando restritas à RME ou RMEP.

3.2.1 Recuperação no protocolo *Spinnaker*

Em (RAO; SHEKITA; TATA, 2011), os autores apresentam uma solução para o problema de garantia de consistência do banco de dados *Spinnaker*. A recuperação é realizada em duas etapas, *local* e *catch up*. Na fase de recuperação *local*, o processo em

recuperação pode seguramente buscar o *checkpoint* armazenado e re-executar seu *log* de registros. Esta execução é feita de maneira idempotente, garantindo que um mesmo estado seja alcançado ao reprocessar o mesmo conjunto de comandos. Requisições não presentes no seu *log* local são obtidas de *logs* de outras réplicas do sistema, as quais permaneceram operantes durante o período de indisponibilidade da réplica em recuperação. Na fase de *catch up*, todas as novas operações de escrita são bloqueadas até que seja garantido que o processo tenha recuperado-se completamente.

A técnica de *checkpoints* particionados também é capaz de buscar fragmentos do *log* de execução de diferentes réplicas. Contudo, a abordagem não especifica se existe a transferência do estado (*checkpoint*) entre as réplicas do serviço. Ainda, embora fragmentos do *log* sejam obtidos das demais réplicas do serviço as réplicas não armazenam *logs* por fragmento do estado nem sugerem fragmentação de estado.

3.2.2 Checkpointing em SiloR

O mecanismo de *checkpointing* presente no banco de dados em memória SiloR (ZHENG et al., 2014) utiliza *threads* responsáveis por armazenar o estado da aplicação. Cada *thread* armazena uma *fatia* do estado em unidades permanentes diferentes. *Checkpoints* são criados a cada intervalo de tempo previamente definido. Neste modelo, existe uma *thread gerenciadora*, que é responsável por atribuir as *fatias* do estado a cada uma das *threads* responsáveis pela criação do *checkpoint* (*checkpointers*). Para garantir que o custo de *checkpoint* seja uniforme entre as partições, cada *checkpointer* percorre uma quantidade fixa de chaves no banco de dados.

Contudo, nesta abordagem, comandos podem alterar o estado da aplicação enquanto *checkpoints* são armazenados, fazendo com que os estados salvos sejam inconsistentes. Os autores afirmam que o fato de *checkpoints* inconsistentes acontecerem é menos custoso em termos de memória. Em comparação à técnica apresentada em (ZHENG et al., 2014), na abordagem de *checkpoints* particionados, os fragmentos do estado são armazenados em uma mesma unidade permanente. Uma vantagem da técnica de *checkpoints* particionados é que os *checkpoints* criados são consistentes pois as *threads* responsáveis por executar requisições sobre as partições são as mesmas que armazenam o estado na unidade permanente. Durante a avaliação da técnica de *checkpoints* particionados não foi utilizada mais de uma unidade permanente. Desta forma não é possível afirmar se, assim como em (ZHENG et al., 2014), a técnica apresentaria melhor desempenho.

4 Técnica para *checkpointing* particionado

Neste capítulo é apresentada uma estratégia para *checkpointing* particionado, que busca reduzir os custos com salvamento de estado durante a execução normal de serviços, assim como o tempo de restauração de estado, por exemplo, para a inclusão de nova réplica ou recuperação de uma réplica. A ideia é que o estado do serviço seja particionado e que imagens das diferentes partições possam ser salvas em instantes distintos. Desta maneira, enquanto é criada a imagem de uma partição, são impedidos de executar somente os comandos que operam sobre esta partição. Operações sobre dados nas demais partições podem ser executadas em paralelo com o procedimento de *checkpointing*. Dessa forma, a degradação na vazão média de serviços implementados com RMEP através do particionamento do estado do serviço é suavizada. Além disso, a latência observada pelos clientes enquanto as réplicas salvam seus estados também é reduzida. O fato do estado do serviço estar particionado, um procedimento de recuperação também pode tirar proveito de arquiteturas com múltiplos núcleos de processamento para efetuar a recuperação em paralelo, reduzindo o tempo de recuperação de uma réplica.

A estratégia de *checkpointing* proposta foi desenvolvida para serviços que implementam RMEP, conforme discutido no Capítulo 2. Entretanto, esta estratégia pode ser generalizada para outros estilos de implementação para RMEP, como por exemplo os propostos em (KOTLA; DAHLIN, 2004; MENDIZABAL et al., 2017; MARANDI; BEZERRA; PEDONE, 2014; LI; XU; KAPITZA, 2018). Note que a execução do procedimento de *checkpoiting*, assim como a notação de dependência entre partições ocorre a partir de uma requisição semelhante a outras operações normais do serviço. Portanto, a execução do *checkpointing* é vista como um comando do máquina de estados replicada. Desta forma, a implementação da técnica é feita com base na especificação das operações do modelo de RMEP utilizado.

4.1 Modelo do sistema

Assume-se um sistema distribuído onde processos são interconectados por uma rede e a comunicação entre processos é feita por troca de mensagens. Processos distinguem-se entre clientes e réplicas do serviço. Considera-se uma quantidade infinita de clientes e um conjunto finito de réplicas. Os clientes são responsáveis pelo envio de requisições, enquanto as réplicas do serviço encarregam-se de executar e responder às requisições dos clientes. As réplicas implementam RMEP. No modelo apresentado, existe um conjunto $\mathcal{T}$ de *threads* executoras do serviço, sendo a quantidade de *threads* executoras e o número de partições

do estado da aplicação totalmente configurável[1]. O modelo de comunicação considerado é assíncrono, ou seja, não exitem limites na velocidade relativa de cada processo nem no tempo necessário para entrega das mensagens.

Foi adotado o modelo de falha por colapso (*crash*), onde um processo segue a sua especificação até que este falhe. Processos que nunca falham são considerados *corretos*. Um processo que falha e posteriormente se recupera usando estado de armazenamento persistente é considerado correto, ainda que temporariamente lento. Processos falham de forma independente uns dos outros e o fato de um processo ter falhado não é detectado por outros processos.

Réplicas do serviço possuem uma memória volátil e uma memória permanente. Sempre que uma réplica falha, o conteúdo de sua memória volátil é perdido. Porém, os dados presentes em seu dispositivo de armazenamento persistente não sofrem nenhum tipo de alteração (perda ou corrupção). Para retomar a sua execução normal, uma réplica faltosa deve realizar um procedimento de recuperação.

4.2 Modelo de execução das réplicas

A técnica de *checkpoints* particionados assume que o estado de cada uma das réplicas do serviço é dividido em fragmentos, onde cada fragmento contém uma partição do estado total. Mais especificamente, considere P o conjunto contendo as partições do estado de uma réplica, representado por $P = \{p_1, \ldots p_j, \ldots p_n\}$, onde p_j indica a partição j. Cada partição contém dados relacionados ao estado do serviço implementado e um mesmo dado não está representado em mais de uma partição. Logo, $\bigcup_{j=1}^{n} p_j = P$ e $\forall p_i, p_j \in P \mid p_i \neq p_j \rightarrow p_i \cap p_j = \varnothing$.

A aplicação implementada pelas réplicas do serviço recebe requisições enviadas pelos clientes, que operam sobre os dados da aplicação. As requisições são entregues a uma *thread* escalonadora que despacha as requisições para as *threads* executoras do serviço com base na classe de conflito associada às requisições, conforme discutido na Seção 2.1.

4.2.1 Algoritmos de execução das réplicas em RMEP

Esta seção apresenta os algoritmos que descrevem o modelo de execução das réplicas. Para facilitar o entendimento é apresentado primeiramente o algoritmo (Algoritmo 2) de que descreve o comportamento das *threads* executoras e em seguida o algoritmo utilizado pela *thread* escalonadora.

O Algoritmo 1 descreve o procedimento de inicialização das réplicas. O estado das réplicas é dividido em n partições, de forma que de forma que porções do estado

[1] No protótipo avaliado neste trabalho foi utilizado o número de *threads* executoras igual ao número de partições. Contudo, pode-se configurar o sistema com mais partições do que executores, ou mais executores do que partições.

sejam fragmentadas e imagens do estado da aplicação sejam mantidas pelo *checkpoint* C, composto por *checkpoints* das partições $\mathcal{C}_0$ a $\mathcal{C}_{n-1}$ (linha 1). Na linha 2 é definido o intervalo entre *checkpoints*, dado em número de comandos recebidos. Nas linhas 3 e 4 são inicializadas a matriz de conflitos, utilizada para garantir a consistência de *checkpoints*, e que será detalhada na Seção 4.3, e as filas de execução das *threads*. O número do último comando escalonado é inicializado com 0 e a variável *proximo_cp*, recebe o *id* da próxima partição a ser salva durante o procedimento de *checkpointing*. Para permitir que as réplicas salvem diferentes porções do estado da aplicação, a cada *checkpoint*, *proximo_cp* é iniciado com o valor da partição correspondente ao *id* da réplica do serviço onde o algoritmo é executado (linhas 5-6). São inicializadas as *threads* executoras, cada uma associada a pelo menos uma partição (linhas 7-8). No esquema de inicialização proposto neste algoritmo, são inicializadas tantas *threads executoras* quanto o número de partições definido para a réplica. Porém, outras abordagens podem ser definidas, a critério do programador. Note que uma mesma *thread* poderia ser responsável por mais de uma partição. Finalmente, é inicializada a *thread escalonadora* (linha 9) e é invocado o procedimento de recuperação responsável por restaurar um estado válido de outras réplicas, caso este exista. O procedimento de recuperação é discutido em detalhes na Seção 4.4. O acesso às variáveis apresentadas no algoritmo é compartilhado entre as *threads*.

Algoritmo 1 $inicia(num\_particoes, intervalo\_checkpoints)$

```
1:  n ← num_particoes;                                  {Número de partições}
2:  Δ_cp ← intervalo_checkpoints;                       {Define o intervalo entre checkpoints}
3:  conf[n][n] ← ℐ;                                     {Matriz que identifica a interação entre threads}
4:  c_list[0..n − 1] ← ∅;                               {Vetor de filas de execução de cada thread}
5:  scheduled ← 0                                       {total de comandos escalonados}
6:  proximo_cp ← replica.id mod n;                      {id da próxima partição que terá seu estado salvo}
7:  for i = 0..num_particoes do
8:      executora(t_i, i);                              {Inicializa n threads executoras}
9:  escalonador();                                      {Inicia a thread escalonadora}
10: recuperacao();                                      {Restaura estado inicial}
```

O comportamento das *threads* executoras é descrito pelo Algoritmo 2. Uma *thread* executora busca o primeiro comando na sua fila de execução (linha 2). Em seguida, na linha 3, extrai os *ids* das *threads* associadas a classe de conflito do comando c (dep_list) e obtém o menor *id* da linha 4. Em seguida, a *thread* verifica se o menor *id* em *dep_list* corresponde ao seu próprio *id*. Caso positivo, a *thread* aguarda em uma barreira até que todas as *threads* envolvidas sinalizem para dar início à execução do comando (linha 13). Ao receber um sinal de todas as *threads* envolvidas na sincronização (laço da linha 6), a *thread* de menor *id* executa o comando e registra a operação em seu *log* de execução (linhas 8 e 9). Em seguida, ela sinaliza as demais *threads* (laço da linha 10) para prosseguirem com sua execução normal, liberando a barreira na linha 14.

Algoritmo 2 *executora*$(thread\_id, particao\_id)$

```
1:  while true do
2:      c ← head(c_list[thread_id]);   {Busca o primeiro elemento da fila de execução.}
3:      dep_list ← get_executores(c.tipo)        {busca o executor da classe de conflito.}
4:      menor ← min(dep_list);              {id da thread que deve executar o comando}
5:      if thread_id = menor then
6:          for all d ∈ dep_list ∧ d ≠ thread_id do
7:              wait(t_d);          {Aguarda sinal das demais threads sincronizantes e ...}
8:          exec(c);                                              {Executa a requisição}
9:          L_particao_id.add(c)
10:         for all d ∈ dep_list ∧ d ≠ id do
11:             sign(t_d)                                    {sinal para prosseguir...}
12:     else
13:         sign(t_menor);                              {avise a thread de menor id e...}
14:         wait(t_thread_id);                        {...aguarde sinal para prosseguir}
```

O Algoritmo 3 descreve o comportamento da *thread escalonadora*. A *thread escalonadora* aguarda a chegada de novas requisições, representadas pelo comando c e verifica a qual classe de conflito a requisição recebida pertence (linhas 1-2). Caso a requisição seja do tipo concorrente (Cnc), ou seja, não requer coordenação entre *threads* e pode ser executada fora da ordem proposta pelo protocolo de consenso, então o escalonador sorteia o *id* da *thread* executora (linha 4) e despacha a operação para a *thread* (linha 5). Caso contrário, a requisição é inserida nas filas de todas as *threads* envolvidas na execução da operação c (linhas 7-9). Para garantir a consistência dos *checkpoints* realizados a *thread* escalonadora mantém o registro de todas as interações realizadas entre as *threads* executoras. O escalonador monta uma *matriz de conflitos* com as informações de quais partições estiveram envolvidas em comandos multi-partições (linhas 10-12). Por fim, a cada intervalo Δ_{cp} (linha 13) de comandos enviados às *threads* executoras, o escalonador cria uma requisição de *checkpoint*, *cp*, responsável por armazenar o estado de, pelo menos, uma partição do estado do serviço. O escalonador busca, através da matriz de conflitos, quais *threads* interagiram com a *thread* com *id* igual a *proximo_cp* (linha 14) e associa à requisição *cp* a classe de conflito envolvendo estas *threads* (linhas 15 e 16). Detalhes sobre a identificação destes conflitos e execução do procedimento de *checkpointing* são discutidos na Seção 4.3. Finalmente, a requisição *cp* é inserida nas filas de execução de todas as *threads* envolvidas (laço da linha 17) e o escalonador incrementa *proximo_cp*, ou seja, atualiza o valor da próxima partição que deve ter seu estado armazenado.

4.3 *Checkpointing* particionado

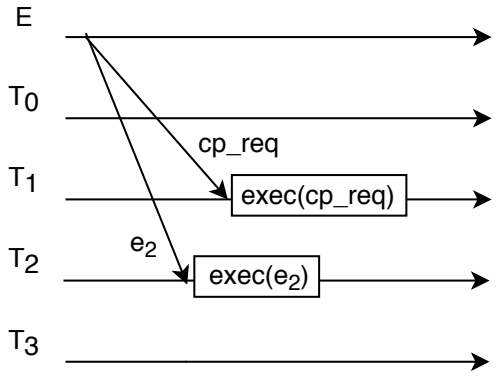
A técnica de *checkpointing* particionado utiliza o mecanismo de execução apresentado na seção anterior. Durante a execução normal do serviço, imagens do sistema

Algoritmo 3 escalonador( )

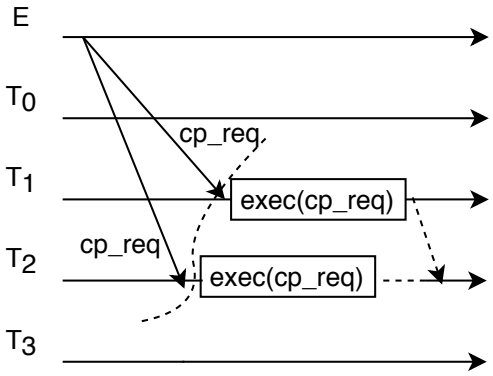
```
1:  while recebe(c) do                                   {Comando recebido pela réplica}
2:      type ← get_tipo(c.tipo);
3:      if type = Cnc then
4:          executor ← random(n);                        {escolhe aleatoriamente uma das threads}
5:          c_list[executor].add(c);                     {Insere o comando na fila da thread executora}
6:      else
7:          dep_list ← get_executores(c.tipo)            {busca o executor da classe de conflito}
8:          for i ∈ dep_list do
9:              c_list[i].add(c);                        {Insere o comando nas filas das threads executoras}
10:     for i ∈ dep_list do
11:         for j ∈ dep_list do
12:             conf[i][j] ← 1;
13:     if scheduled mod Δ_cp then
14:         conflitam ← mapeia_conflito(proximo_cp);     {Busca a lista de threads
        envolvidas no checkpoint}
15:         cp_conf ← get_classe(conflitam);             {Busca a classe de conflito associada a
        lista de threads envolvidas}
16:         cp_req ← ⟨cp, conflitam, cp_conf⟩;           {cria requisição de checkpoint}
17:         for id ∈ conflitam do
18:             c_list[id].add(cp_req);                  {Insere o comando de checkpoint nas filas das
        executoras}
19:         proximo_cp ← (proximo_cp + 1) mod n
```

devem ser armazenadas em disco. Para isto, o escalonador decide, através da matriz de conflitos, quais partições devem ter seus estados armazenados. Ao criar uma requisição de *checkpoint*, o escalonador a despacha para as *threads* que devem sincronizar durante a realização do *checkpoint*. Uma requisição de *checkpoint* é representada pela tupla $cp = \langle cp, particoes[], tipo \rangle$, onde cp indica que é uma requisição de *checkpoint*, $particoes[]$ indica quais partições são armazenadas e *tipo* indica a classe de conflito associada. Considere a Figura 6, onde são ilustrados os casos onde não é necessária a sincronização entre *threads* (6a) e um caso onde existe sincronização entre *threads* (6b). Ao receber uma requisição de *checkpointing*, não havendo necessidade de sincronizar o *checkpoint* com outras partições, a *thread* T_1 (Figura 6a) armazena o estado da partição 1 ($exec(cp\_req)$) e comandos envolvendo outras partições podem ser executados em paralelo. Na Figura 6b a classe de conflito associada ao comando *cp* sincroniza *threads* T_1 e T_2, responsáveis por executar a requisição. Portanto, as *threads* T_1 e T_2 coordenam as suas execuções através de uma barreira e, quando ambas recebem *cp*, elas armazenam, em paralelo, os estados das partições 1 e 2.

Conforme observado no Algoritmo 3 (linhas 10-12), a *thread* escalonadora também é responsável por registrar interações entre *threads*. As interações são registradas em uma *matriz de conflitos*, $conf$. Uma interação entre *threads* ocorre sempre que houver a



(a) *Checkpoint* de uma única partição



(b) Sincronização de *threads* para realização de *checkpoints* em paralelo

Figura 6 – Realização de *checkpoints* particionados

sincronização entre duas ou mais *threads* para a execução de um comando que atualiza dados em mais de uma partição (como o exemplo apresentado na Figura 4b). A matriz de conflitos é uma matriz quadrada de ordem n, onde n é o número de *threads* executoras. Cada elemento da matriz assume os valores 0 ou 1, onde $conf[lin][col] = 1$ significa que a *thread lin* interagiu com a *thread col*. Inicialmente a matriz de conflitos é uma matriz identidade, significando que cada *thread* interage com no mínimo a partição respectiva ao seu índice. Cada linha *lin* indica as interações de cada *thread lin* com as demais *threads*.

Para melhor ilustrar esta ideia, considere a execução dos comandos e_1, e_2, e e_{13} da Figura 4 e a matriz abaixo, representando a matriz de conflito da réplica em execução. Nela é possível observar que para executar e_1 e e_2, as *threads* t_1 e t_2 não interagiram com as demais *threads*, mantendo os valores de $conf[1][1]$ e $conf[2][2]$ em 1. Já para a execução de e_{13}, a *thread* t_1 interagiu com a *thread* t_3, atualizando os valores de $conf[1][3]$ e $conf[3][1]$ também para 1.

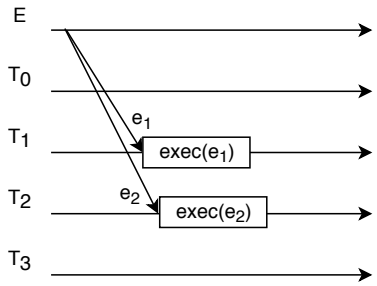


Figura 7 – Comando

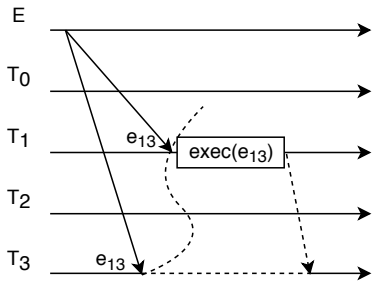


Figura 8 – Comando

$$Conf = \begin{pmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 1 \end{pmatrix}$$

Um *checkpoint* é realizado sempre que uma determinada quantidade de comandos q for executada. Por exemplo, o *checkpoint* de uma partição é realizado sempre que Δ_{cp} operações forem executadas. A *thread* escalonadora é responsável por determinar quando o *checkpoint* de uma partição deve ser realizado. Portanto, o escalonador insere uma requisição de *checkpoint* na fila da *thread* executora responsável pela partição. Suponha

que a partição 1 deve ter seu estado armazenado. Isso ocorre quando Δ_{cp} operações foram entregues e $proximo\_cp = 1$. Então, o escalonador insere na fila de execução da *thread* 1 a requisição $\langle cp, [1], C1 \rangle$, onde *cp* identifica a operação de *checkpoint* sobre a partição 1, e $C1$ indica a classe de conflito associada a esta requisição. As classes de conflito associadas aos comandos de *checkpoint* são determinadas através da matriz de conflitos (vide Algoritmo 3, linhas 13-18).

4.3.1 Algoritmo para mapeamento de conflitos

O Algoritmo 4 descreve como a classe de conflito associada a uma requisição de *checkpoint* é construída. Ele retorna uma lista contendo os identificadores das partições que devem ter seus conteúdos salvos em armazenamento persistente. Este procedimento recebe como parâmetro *proximo_cp*, isto é, a *thread* responsável por executar operações sobre a partição que deve ter seu estado armazenado. Primeiramente, o algoritmo inicializa a lista de partições envolvidas no *checkpoint* (linha 1). Em seguida, adiciona todas os *ids* das partições cujas *threads* interagiram diretamente com a partição *proximo_cp* (linhas 2–4), isto é, aquelas indicadas por $conf[proximo\_cp][i] = 1$. Em seguida, nas linhas 5 à 9, são observadas possíveis interações de maneira transitiva. São verificadas as interações entre as *threads* que interagiram com *proximo_cp*, mas também com outras *threads*. Caso sejam encontradas interações (linha 8), os *ids* das *threads* são inseridos na lista de partições envolvidas (linha 9)[2]. Após completar a lista de partições envolvidas no *checkpoint*, é necessário reinicializar as linhas da matriz $conf$, para evitar falsos positivos entre interações nos *checkpoints* futuros. Para isto, o algoritmo executa as linhas 10 à 15. A lista de partições que precisam ter o seu estado salvo no *checkpoint* corrente é retornada pelo algoritmo (linha 16).

Para ilustrar estes passos, considere que a matriz $conf$ é a matriz representada a seguir e $proximo\_cp = 0$, ou seja, na próxima execução do procedimento de *checkpointing*, pelo menos, a partição 0 terá seu estado salvo.

$$Conf = \begin{pmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 1 \end{pmatrix}$$

Após a execução das linhas 1 à 4 do Algoritmo 4, a lista *particoes* contém os *ids* 0 e 3. Isto significa que algum comando causou atualização nos valores de ambas partições. Para garantir que atualizações envolvendo a partição 3 e outras partições também sejam

[2] Neste algoritmo, assume-se a semântica onde os valores retornados para o iterador i, no laço da linha 5, são avaliados a cada iteração. Isso é importante pois a lista $particoes[]$ pode ter novos elementos inseridos neste laço (conforme a linha 9 do algoritmo).

Algoritmo 4 *mapeia_conflito*(*proximo_cp*)

```
1:  particoes[] ← ∅                                   {lista de partições envolvidas no checkpoint}
2:  for i = 0..n − 1 do                                                  {para todas as partições..}
3:      if conf[proximo_cp][i] = 1 then{..se comandos envolveram partições iniciador e
    i..}
4:          particoes.put(i);                                  {.. insira esta partição na lista.}
5:  for i ∈ particoes[] do          {para todas as partições que interagiram com iniciador}
6:      if  i ≠ proximo_cp then                                          {se não for o iniciador}
7:          for k = 0..n − 1 do                                           {para cada partição..}
8:              if conf[i][k] = 1 ∧ k ∉ particoes[] then                  {...verifique interações}
9:                  particoes.put(k);                                        {insira a partição}
10: for i ∈ particoes[] do          {para cada partição com interações com proximo_cp}
11:     for j = 0..n − 1 do                {para todas possíveis interações entre partições}
12:         if i=j then
13:             conf[i][j] ← 1;                          {cada partição interage com ela mesma}
14:         else
15:             conf[i][j] ← 0;                                          {não existe interação}
16: return particoes[];   {retorne a lista com os ids das partições a serem armazenadas}
```

avaliadas, após a execução das linhas 5 à 9 do algoritmo, a partição 2 também é inserida na lista *particoes*. Logo, no instante de salvamento da imagem da partição 0, também serão salvas as partições 2 e 3. Finalmente, as linhas 0, 2 e 3 da matriz *conf* devem ser reinicializadas para uma representação na qual elas não indicam conflito com as demais partições (linhas 10 a 15), resultando: $conf[0] = [1, 0, 0, 0]$, $conf[2] = [0, 0, 1, 0]$ e $conf[3] = [0, 0, 0, 1]$.

4.3.2 Algoritmo para armazenamento de *checkpoints* particionados

Ao criar uma requisição de *checkpoint*, o escalonador despacha esta requisição para as *threads* executoras cujos estados devem ser salvos. Esta requisição é tratada conforme descrito no modelo de execução do serviço, através da execução do comando $exec(c)$, conforme descrito no Algoritmo 2, linha 8. A execução de um comando da aplicação é ilustrada pelo Algoritmo 5 e o Algoritmo 6 descreve o procedimento de *checkpointing*. Para cada comando recebido, a operação do comando é verificada e o procedimento adequado para a execução do comando é executado (linhas 1 a 7). Após executar o comando, a *thread* atualiza o índice do último comando executado naquela partição.

Ao receber uma requisição de *checkpoint* (linhas 6-7 do Algoritmo 5), uma *thread* de execução se comporta conforme descrito no Algoritmo 6. São passados como parâmetros o identificador da partição e o último comando que executou sobre esta partição. O algoritmo inicia com a serialização do estado da partição, que deverá ser salvo (linha 1) e com o armazenamento em disco (linha 2), informando o nome do arquivo (texto indicado pelo primeiro parâmetro da função *write*) e os dados a serem armazenados (o estado

Algoritmo 5 $exec(c)$

```
1: if c.op = C_1 then
2:     ...;                                  {código de execução do comando C_1 da aplicação}
3:     ⋮
4: else if c.op = C_n then
5:     ...;                                  {código de execução do comando C_n}
6: else if c.op = cp then
7:     checkpoint(particao_id, ultimo_exec);  {executa o procedimento de checkpoint}
8: ultimo_exec ← c.instancia;                 {Atualiza a instância do último comando executado}
9:
```

serializado, conforme o segundo parâmetro da função *write*). Na linha 3 é adicionada uma marcação (#) indicando que o arquivo foi armazenado corretamente. Este passo é necessário para garantir que não ocorreram falhas durante o armazenamento da imagem, ou seja, o arquivo não está corrompido. Na linha 4 é criado um arquivo de metadados, identificando a última instância que operou sobre a partição (linha 4). Em seguida, na linha 5, também é adicionada uma marcação indicando que o arquivo de metadados foi atualizado corretamente. Por fim, na linha 6 o *log* de execução contendo apenas os comandos desta partição é apagado. Dessa forma, todas as requisições associadas à partição *particao* não farão mais parte do log $\mathcal{L}_{particao}$.

Algoritmo 6 $checkpoint(particao, ultimo\_exec)$

```
1: C ← get_estado(particao);                          {copia o estado da partição desejada}
2: write("e.particao.ultimo_exec", C);                {armazena o estado de forma persistente}
3: append("e.particao.ultimo_exec", #);               {adiciona marca de verificação}
4: write("m.particao.ultimo_exec", m);                {salva o arquivo de metadados de forma
   persistente}
5: append("m.particao.ultimo_exec", #);               {adiciona marca de verificação}
6: truncate(L_particao);                              {limpa o log de operações}
```

4.4 Recuperação utilizando *checkpoints* particionados

Durante sua execução normal, uma réplica do serviço realiza *checkpoints* periodicamente. Os *checkpoints* criados pelas réplicas do serviço são utilizados para que, após uma eventual falha, seja possível que a réplica recupere o estado a partir de outras réplicas. Durante sua execução normal, a réplica do serviço mantém registradas em um *log* todas as operações processadas antes da criação de um *checkpoint*. Ao armazenar um *checkpoint* em disco, a réplica do serviço apaga o *log* de operações, pois o resultado da execução do *log* já está refletido na imagem salva em disco.

A estratégia de *checkpoints* particionados mantém, junto com os dados da aplicação, um conjunto de metadados, identificando qual a última operação que executou sobre o

checkpoint de cada partição. O conjunto de metadados pode ser descrito por tuplas da forma $\langle n, c_x \rangle$, onde n indica o índice da partição e c_x identifica o último comando executado pelo *checkpoint* mais recente da partição n. Além disso, as réplicas mantêm um conjunto de *logs* de execução. Cada *log* contém o registro das operações que executaram sobre cada uma das partições do serviço. Caso exista pelo menos um *checkpoint* associado à partição, então o *log* contém apenas os comandos posteriores à execução do *checkpoint*.

Em implementações tradicionais de recuperação, uma réplica em recuperação r' requisita o estado atual de outras réplicas do serviço. Ao receber uma requisição de estado, uma réplica correta, r, envia o seu *checkpoint* e o *log* de operações. Ao receber o estado de r, a réplica r' instala a imagem recebida e executa o *log* de operações.

A recuperação utilizando *checkpoints* particionados comporta-se de maneira semelhante. Contudo, nesta estratégia a réplica em recuperação requisita às demais réplicas porções do estado baseada nos metadados dos *checkpoints* de cada partição do estado armazenada. Esta estratégia permite que uma nova réplica receba partições do estado de diferentes réplicas do serviço em paralelo.

Ao iniciar o procedimento de recuperação, a réplica em recuperação requisita a todas as réplicas do serviço para que enviem os metadados referentes aos *checkpoints* de todas as partições. Ao receber os metadados, a réplica r compara os identificadores dos últimos comandos que operaram sobre cada partição nas demais réplicas, identificando os mais atuais. Então, a réplica r requisita a imagem e o *log* das partições mais atualizadas. Conforme o recebimento destas informações, a réplica instala os *checkpoints* e, após instalar os *checkpoints* para todas as partições, processa o *log*. Após restaurar todas as partições, incluindo o processamento dos seus *logs*, a réplica é considerada correta e pode processar novas requisições.

A Figura 9 ilustra como este mecanismo funciona. Suponha que o estado do serviço é particionado em 4 e a réplica em recuperação é a réplica R_0. Considere que o arquivo de metadados da réplica R_0 possui as seguintes informações $\langle (0, 100), (1, 200) \rangle$, ou seja, o último comando que operou sobre a partição 0 foi o comando de *id* 100 e o comando de *id* 200 sobre a partição 1. O estado das demais partições não foi localizado, pois antes mesmo de recuperar, a réplica executava, mas falhou antes do armazenamento destes estados. Primeiramente, a réplica R_0 requisita os metadados (mensagem $get(m)$) das réplicas R_1 e R_2, recebendo os dados $\langle (0, 500), (1, 200), (2, 300), (3, 400) \rangle$ da réplica R_1 e $\langle (0, 500), (1, 200), (2, 300), (3, 400) \rangle$ da réplica R_2 pela mensagem $send(m)$. Em seguida, R_0 compara os arquivos de metadados recebidos (procedimento $monta\_lista()$), montando a seguinte lista $\langle (0, 2), (1, 1), (2, 1), (3, 1) \rangle$, onde o primeiro índice de cada par indica a partição a ser solicitada e o segundo, o *id* da réplica. Após montar esta lista, R_0 requisita as imagens e os *logs* da partição 0 para a réplica R_1 ($get(0)$ na figura) e das partições $1, 2$ e 3 (método $get(1, 2, 3)$) para a réplica R_2. Estas informações solicitadas correspondem

aos *checkpoints* mais recentes e, consequentemente, *logs* mais curtos disponíveis para cada partição. Em seguida, a réplica R_0 instala os *checkpoints* recebidos, sendo possível a execução do procedimento de instalação em paralelo para diferentes fragmentos do estado (retângulos contendo CP_0 e CP_1 e CP_1 e CP_2). Finalmente, a réplica executa os *logs* das partições. Este último procedimento também pode ocorrer em paralelo, respeitando as classes de conflito dos comandos.

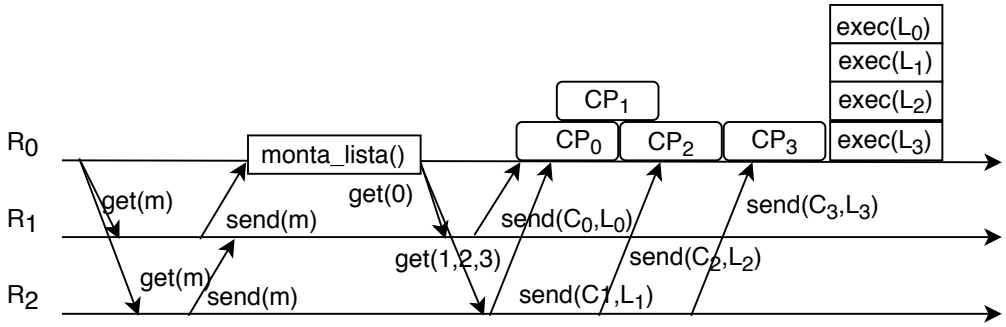


Figura 9 – Procedimento de recuperação de uma réplica.

4.4.1 Algoritmo de recuperação

O Algoritmo 7 descreve o processo de recuperação de uma réplica. Ao reiniciar, a réplica busca localmente a última instância executada em cada partição representada por seus *checkpoints* armazenados localmente até o momento da falha (linha 1). Em seguida, a réplica cria uma *lista de metadados* r para os *checkpoints* de cada partição (linhas 2-3), inserindo seu *id*, e os metadados dos seus *checkpoints* locais, ou seja, os *ids* das partições (n) e o identificador do último comando associado ao *checkpointing* de cada partição (cx). A réplica procede enviando a requisição de recuperação ($get(m)$) para todas as réplicas do sistema (linha 5). Como resposta a esta requisição, a réplica em recuperação recebe, de cada réplica correta, uma lista contendo o *id* da réplica e o *id* do último comando que operou sobre cada partição (linhas 6-7). Para cada uma das listas recebida, a réplica compara as tuplas (linhas 8-10), observando qual o comando mais recente associado a *checkpoints* das partições obtidos pelas demais réplicas. Caso sejam maiores do que os contidos atualmente na lista r, a réplica atualiza os valores da lista para que um *checkpoint* mais atual possa ser recuperado (linhas 9-10). Ao finalizar a construção da lista, a réplica em recuperação fará a requisição das partições com base em sua lista (linha 12-14). Para possibilitar paralelismo entre requisição de *checkpoints* de diferentes partições, para cada requisição feita para *checkpoint* de partição em outra réplica, uma nova *thread* é criada e responsável por receber o *checkpoint* e *log* da réplica requisitada (linhas 15-20). Ao receber a resposta (linha 17), a *thread* instala o *checkpoint* recebido (linha 18) e atualiza o *log* de comandos para a partição em questão (linha 19). Uma variável *rec_cp* é usada para indicar o número de restaurações de partições pendentes. Isso é importante para garantir

que a execução dos comandos nos arquivos de *log* seja feita apenas após a restauração do *checkpoint* de todas as partições. Esta garantia é dada pela espera implementada nas linhas 21 e 22[3]. Por fim, a réplica reexecuta os *logs* (linhas 23-25) e então segue com sua execução normal.

Algoritmo 7 Recuperação()

```
1: C ← busca_estado();                {Obtém índice de checkpoints de cada partição do armazenamento local }
2: for c ∈ C do                       {Metadados contendo informações de checkpoint local}
3:     r.append(⟨r_id, c.n, c.cx⟩)    {Monta a lista de metadados }
4: for r_i ∈ R onde i ≠ id do         {Para todas as réplicas ativas...}
5:     send(r_i, get(m));             {... solicita ids do último comando associado a checkpoints em cada partição}
6: for r_i ∈ R onde i ≠ id do         {Para todas as réplicas ativas...}
7:     recv(m);                       {... recebe metadados }
8:     for j = 0..n − 1 do            {para todos itens de m}
9:         if m[j].cx > r[j].cx then  {...se possui checkpoint mais atual...}
10:            r[j] ← m[j]            {... atualiza tupla sobre checkpoint da partição}
11: rec_cp ← 0;                       {variável para indicar número de restaurações em andamento}
12: for i = 0..n − 1 do                {para todos elementos presentes na lista r...}
13:     if i ≠ id then                 {...caso precise buscar em outra réplica...}
14:         send(r_id, get(r[i]));     {...requisite o estado e o log}
15:         nova thread:
16:             rec_cp ← rec_cp + 1    {restauração desta partição está em andamento}
17:             recv(C, L_i);          {recebimento do checkpoint e log}
18:             C_k ← C;               {instale o estado recebido}
19:             L[k] ← L;              {atualize o log}
20:             rec_cp ← rec_cp − 1    {indica que a restauração desta partição foi completada}
21: while rec_cp ≠ 0 do
22:     no − op;                       {Aguarda restaurações de partições encerrarem}
23: for i = 0..n − 1 do                {para todos arquivos de log...}
24:     for each c ∈ L[i] do           {para cada comando no arquivo de log...}
25:         exec(c);                   {execute o comando}
```

4.5 Discussão

Foi apresentada neste capítulo a estratégia de *checkpoints* particionados. A motivação em dividir a execução de *checkpoints*, de forma a salvar porções menores do estado a cada instante permite, inicialmente, que a vazão das réplicas em execução normal seja menos afetada pelo bloqueio das *threads* durante o salvamento de estado. Além de processar comandos em paralelo com o salvamento parcial do estado, a técnica deve

[3] No Algoritmo 7 foi apresentado como uma *espera ocupada* para melhor compreensão. Contudo, mecanismos mais eficientes podem ser implementados.

proporcionar menor latência para os clientes, visto que o salvamento do estado é mais rápido, dado que apenas uma porção do estado é salva por vez. Além disso, ao configurar réplicas para salvarem partições distintas a cada intervalo de *checkpoint*, enquanto uma réplica r_i bloqueia a execução de comandos destinados a uma determinada partição p_x, possivelmente outra réplica r_j esteja salvando outra a partição p_y e, como consequência, os comandos destinados à partição p_x serão executados pela réplica r_j, enquanto os comandos destinados à partição p_y serão executados pela réplica r_i. Com isso, além de observar menor impacto na vazão, também se espera menor interferência na latência de comandos durante a execução de *checkpoints*.

Outro resultado que pode ser observado com a estratégia de *checkpoints* particionados é a recuperação mais rápida. Note que ao particionar o estado da aplicação em partições, a transferência de estados durante a recuperação pode envolver diversas réplicas, distribuindo a carga de trabalho envolvida na transferência entre mais de uma réplica. Além disso, tanto a transferência quanto a instalação do estado podem ser executadas em paralelo, fazendo melhor uso de arquiteturas com múltiplos núcleos de processamento.

Um aspecto que exige destaque é o adiantamento do *checkpoint* de partições. Conforme apresentado, a cada intervalo entre *checkpoints*, é necessário que uma *thread* realize o salvamento de pelo menos uma partição. Contudo, se comandos envolvendo esta e outras partições foram executados no intervalo, é necessário que o estado destas demais partições também seja armazenado em disco. Este efeito pode encadear-se de tal forma que todas as partições tenham interagido entre si, transitivamente ou diretamente. Tal efeito provoca a sincronização entre todas as *threads*, bloqueando totalmente a execução de requisições durante o salvamento do estado completo, neste caso. Note, entretanto, que diferentemente da técnica tradicional, onde uma única *thread* salva o estado da aplicação, na técnica de *checkpoints* particionados o salvamento do estado ocorre em paralelo. Com isso, espera-se um salvamento mais rápido, determinado pelo número de partições e número de *threads* em execução.

O comportamento da aplicação e carga de trabalho devem influenciar o desempenho da técnica. Assumindo que a distribuição de requisições entre partições é uniformemente distribuída, espera-se que os tamanhos dos *checkpoints* das partições tenham tamanhos semelhantes e, consequentemente, o tempo para salvar o estado de qualquer partição é semelhante. Entretanto, em cenários onde poucas partições são muito acessadas, enquanto outras são raramente acessadas, o tempo para salvamento das partições mais pesadas pode representar praticamente o tempo para salvar o estado completo da aplicação, enquanto salvar as demais partições ocorre de forma praticamente instantânea. Embora neste cenário os benefícios da técnica sejam pouco perceptíveis, o comportamento do sistema durante o *checkpoint* seria semelhante ao da abordagem de *checkpoint* tradicional. Dessa forma, no pior caso, pode-se esperar um desempenho semelhante ao que já existe nas técnicas

tradicionais de *checkpointing*.

Para obter maior beneficio com a técnica de *checkpointing* particionado, políticas de particionamento de estado devem proporcionar um balanceamento de carga entre partições equilibrado, ao mesmo tempo em que reduzem a probabilidade de comandos envolvendo mais de uma partição. Embora não faça parte do escopo deste trabalho, estratégias para particionamento eficiente podem ser combinadas com a técnica de *checkpointing* particionado (CURINO et al., 2010; KUMAR; DESHPANDE; KHULLER, 2013; QUAMAR; KUMAR; DESHPANDE, 2013; LE et al., ; MARANDI; BEZERRA; PEDONE, 2014; LI; XU; KAPITZA, 2018; Coelho; Pedone, 2018). Com relação à redução de conflitos envolvendo múltiplas partições, diversas aplicações enquadram-se no caso onde os conflitos concentram-se entre grupos disjuntos de partições, como por exemplo sistemas de fila de mensagem e de processamento de *streaming*, onde grande parte das mensagens são direcionadas apenas à subconjuntos de filas, sendo estes subconjuntos disjuntos (EUGSTER et al., 2003; SACHS et al., 2010; CHEN et al., 2011; VIEL; UEDA, 2014; CHENG et al., 2017).

5 Avaliação Experimental

Este capítulo descreve a metodologia utilizada para avaliar a técnica de *checkpointing* particionado, comparando seu desempenho com o modelo de *checkpointing* tradicional. Também é apresentado o comportamento de ambas as estratégias combinando a técnica de Cópia na Escrita[1] (CnE), visto que esta estratégia tem como objetivo reduzir o bloqueio na execução de comandos durante o salvamento de estado. Para efeitos de comparação, foram avaliados os seguintes aspectos:

- Como a sobrecarga causada pelo procedimento de *checkpointing* afeta o sistema para diferentes quantidades de *threads executoras*;
- O quanto a frequência na operação de *checkpointing* afeta a vazão média do sistema ao longo da execução;
- A influência da técnica de cópia na escrita no desempenho usando ambas as técnicas de *checkpointing*, particionado e tradicional.

5.1 Implementação do protótipo

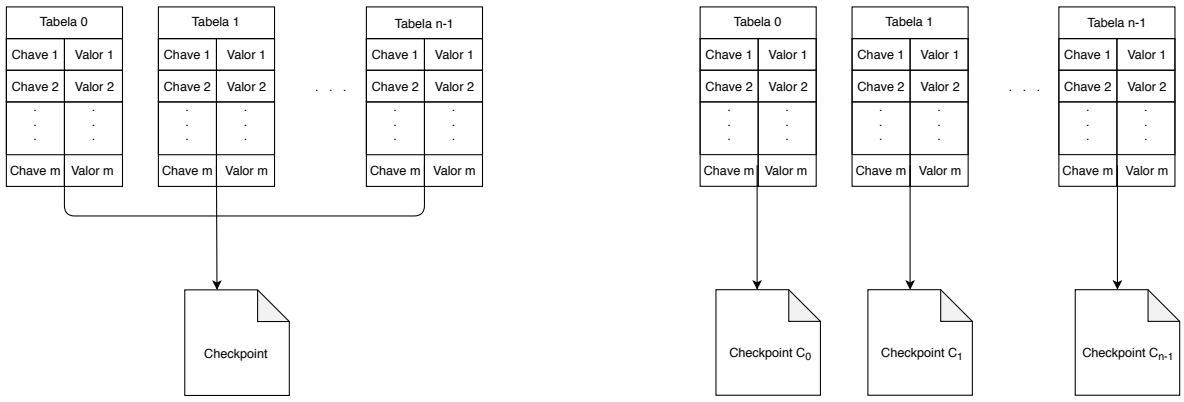
Para avaliar o desempenho das técnicas de *checkpointing*, foi implementado um protótipo de banco de dados chave-valor usando a linguagem de programação Java. O serviço é executado seguindo o modelo de RMEP proposto em (ALCHIERI et al., 2017). A aplicação é composta por um conjunto de tabelas chave-valor, onde cada chave é representada por um número inteiro e o valores associados a cada chave são *arrays* de 1024 *bytes*. A aplicação implementa as seguintes operações:

- *put(table, key, value)* - Insere um *array* de *bytes* (*value*), associado a chave (*key*) em uma determinada tabela (*table*);
- *get(table, key)* - Retorna o *array* de *bytes* associado a chave *key* da tabela *table*. Caso a chave ou a tabela não forem encontradas, retorna *null*;
- *put_table(table)* - Cria uma nova tabela vazia com identificador *table*;
- *get_table(table)* - Retorna todos os registros contidos na tabela *table*;
- *remove(table,key)* - Remove de uma tabela *table* o registro associado a chave *key*. Caso a chave ou a tabela não forem encontradas, retorna *null*;

[1] Tradução do Inglês *copy-on-write*.

- *table_remove(table)* - Remove a tabela *table* do conjunto de tabelas. Caso a tabela não exista, retorna *null*;
- *table_size(table)* - Retorna a quantidade de pares chave–valor da tabela *table*;
- *table_check(table)* - Retorna *true* caso a tabela com identificador *table* exista, e *false* caso contrário;
- *swap(table1,key1,table2,key2)* - Troca mutuamente os valores associados às chaves *key1* e *key2*, pertencentes as tabelas *table1* e *tables2*, respectivamente;
- *multi_table_put(table[t1, t2, ...], key[k1, k2, ...], value[v1, v2, ...])* - Associa os valores contidos no *array value* às chaves contidas no *array key* e insere os pares chave–valor resultantes nas tabelas indicadas pelo *array table*. É necessário que os *arrays table, key* e *value* tenham o mesmo tamanho. Caso contrário, é retornado o valor *null*;
- *checkpoint(particao,last_cid)* - Comando invocado especificamente pela aplicação. Através deste comando é dado o início do *checkpoint* da partição de identificador (*particao*) e *last_cid* é o identificador do último comando que operou sobre a partição.

Para o particionamento do estado do serviço, cada tabela chave-valor é considerada uma partição. Por exemplo, a operação *checkpoint([0])* armazena em disco os dados contidos na tabela 0, enquanto a operação *checkpoint([0,3,4])* cria uma imagem em disco das partições 0, 3 e 4. Na abordagem tradicional a operação *checkpoint* salva todas as tabelas do serviço. A Figura 10, ilustra o salvamento de estado utilizando o *checkpoint* completo (Figura 10a) e *checkpoints* particionados (Figura 10b), onde partições do estado são armazenadas em imagens independentes.



(a) *checkpoint* tradicional (b) *checkpoint* particionado

Figura 10 – Relação entre *checkpoint* completo e particionado.

O modelo de execução proposto em (ALCHIERI et al., 2017) descreve a noção de *classes de conflito* para estabelecer a relação de dependência entre requisições. Dessa

forma, para qualquer par de requisições r_i e r_j, a relação entre as suas classes de conflito estipula que estas executem sequencialmente caso ambas requisições acessem um mesmo dado e pelo menos uma das requisições é uma atualização. Alternativamente, quando r_i e r_j não acessam o mesmo dado ou não atualizam o dado acessado, estas podem executar concorrentemente. Cada classe de conflito está associada ao menos a uma *thread executora.* No protótipo de avaliação as operações *put*, *put_table*, *remove*, *table_remove*, *multi_table_put* e *swap*, quando acessam pelo menos uma mesma tabela, devem ser executadas sequencialmente. Portanto, foram definidas classes de conflito relacionando as *threads executoras* de cada partição e estes comandos, chamados de sequencias (Seq). Por exemplo, a operação $swap(0, 1, 3, 5)$ realiza a troca entre os valores das chaves 1 da tabela 0 e 5 da tabela 3. Uma classe de conflito do tipo Seq, definida como $(Seq, (0, 3))$, está associado ao comando *swap* e às *threads* t_0 e t_3, garantindo a sincronização entre as *threads* t_0 e t_3 para a execução do comando. Requisições de *checkpoint* também forçam sincronização entre partições. No caso de *checkpointing* tradicional, todas as *threads* executoras devem sincronizar. Na técnica com particionamento de *checkpointing*, sincronizam apenas as *threads* que tiveram comandos envolvendo suas partições. Para requisições do tipo Cnc foi atribuída uma *thread* executora por partição. Por exemplo, a operação *get(0,5)* é executada pela *thread* 0, e esta associada à classe de conflito $get(0, n) = (Cnc, 0)$, onde n é qualquer chave pertencente a tabela 0.

Para geração de carga foi implementado um protótipo cliente que escolhe aleatoriamente requisições a serem enviadas ao serviço. O cliente é composto por diversas *threads* e cada uma é responsável por enviar requisições ao serviço. As operações são escolhidas baseado em dois parâmetros informados na inicialização do gerador de carga. O primeiro é a *taxa de leituras* e o segundo é a *taxa de conflito*. Para criar uma requisição, cada *thread* sorteia um valor entre $[1, 100]$. Caso este valor seja menor ou igual a *taxa de leituras* então é criada uma operação de leitura, caso contrário é gerada uma operação de escrita. Operações de escrita podem acessar dados de múltiplas partições (p. ex. comando *swap* ou *multi_table_put*). Então, é sorteado um valor entre $[1, 100]$ e caso este valor seja menor ou igual a *taxa de conflitos*, o gerador de carga cria uma operação de escrita conflitante, por exemplo *multi_table_put*.

Cada tabela representa um fragmento no estado da aplicação. O tamanho do estado completo é de $1GB$. Para este estudo, a imagem do serviço foi dividida em 4 e 8 partições, onde no primeiro caso cada partição tem tamanho de $256MB$ e $128MB$ no segundo caso. Para criação das tabelas, foram executados, durante a inicialização do sistema, n comandos $put\_table(table)$, onde n representa o número de partições do estado do serviço e $table$ o identificador da tabela monotônico crescente pertencendo ao intervalo $[1, n]$. Para inserção das chaves, foi utilizado o comando $put(table, key, value)$. Como cada chave (key) mapeia um *array* $(value)$ de 1024 *bytes* cada tabela possui 250000 chaves distintas, quando o estado da aplicação é particionado em 4, e quando particionado em 8 cada tabela

possui 125000 chaves distintas. Neste trabalho, foi considerado que o tamanho do estado da aplicação não altera de tamanho ao longo da execução. Essa premissa foi necessária para evitar que a sobrecarga ocasionada por *checkpoints* sucessivos varie em função do tamanho do estado da aplicação. Portanto, as operações *remove_table* e *put_table* não são executadas durante os testes. Foram utilizadas operações dos tipos *put* e *multi_table_put* para atualizar valores das tabelas do serviço.

5.2 Ambiente de experimentação

Para a realização dos experimentos deste trabalho utilizamos a plataforma Emulab (WHITE et al., 2002). O Emulab é um ambiente controlável, onde é possível que cada usuário configure e acesse uma rede de computadores, criada utilizando recursos físicos ou virtualizados. Foram utilizadas 4 máquinas físicas, sendo elas 3 réplicas servidores e 1 réplica cliente. A escolha por 3 réplicas servidoras se deve ao fato de as réplicas do serviço fazerem parte do protocolo de consenso, sendo necessárias $2f + 1$ réplicas do serviço para tolerar f falhas. Com esta configuração o serviço tolera 1 nodo faltoso sem comprometer a sua disponibilidade. Na máquina cliente é instanciado um gerador de carga com número de *threads* responsáveis por gerar requisições variável. Durante a execução do experimento configurou-se a *heap* de memória da *Java Virtual Machine* (JVM) em 32 GB.

As especificações das máquinas para as réplicas do serviço e do gerador de carga são as mesmas, tendo sido hospedadas em máquinas do tipo d430, com a seguinte configuração: 2 processadores Intel®Xeon®2.4 GHz 64-bit 8-núcleos, 64GB RAM e disco de 200 GB 6Gbps SATA SSD.

5.3 Caracterização da carga de trabalho utilizada

Primeiramente, para avaliar a vazão média do serviço, buscou-se identificar o ponto de saturação do sistema. Para isso, foram realizados testes de carga sobre o sistema em execução normal, ou seja, sem a ocorrência de falhas e sem execução de *checkpoints*. Para cada execução foi utilizada uma quantidade diferente de clientes, representando um acréscimo global na vazão do sistema. A carga gerada pelos clientes foi composta em 100% de operações de escrita. As réplicas do serviço foram configuradas com 4 *threads* de execução. Durante o intervalo de 1 minuto foi observada a vazão do sistema a cada segundo. Ao final deste período de tempo foi calculada a vazão média do sistema. A Figura 11 apresenta o ponto de saturação do sistema. Nela, podemos observar a vazão média (eixo horizontal) expressa em quilocomandos *versus* o 90º percentil da latência (eixo vertical) expressa em milisegundos. Com base neste é gráfico, é possível observar que após a carga de 8 kcmds/s existe um aumento considerável na latência de resposta. Devido a esta observação conclui-se que o sistema encontra-se em estado de saturação, fazendo com que

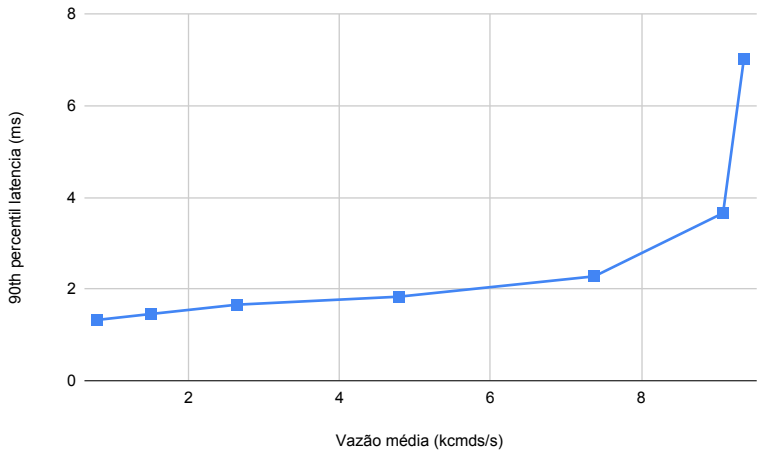


Figura 11 – Ponto de saturação do sistema para configuração com 4 *threads* de execução e carga de trabalho exclusivamente escrita.

requisições fiquem enfileiradas fazendo com que o tempo de resposta comece a aumentar. Durante a realização dos experimentos foi utilizada a quantidade de clientes necessária para gerar 70% da carga máxima (aproximadamente 8 kcmds/s), o que corresponde a 22 clientes.

5.3.1 Cargas de trabalho

As cargas de trabalho geradas para os experimentos diferem em dois pontos. O primeiro em relação a operação a ser realizada, sendo estas operações escritas ou leituras. O segundo, está relacionada a classe de conflito de cada operação. Conforme mencionado anteriormente, operações podem ser executadas concorrentemente ou de maneira sequencial dependendo da classe de conflito associada. Algumas cargas de trabalho possuem requisições que forçam a sincronização de determinadas *threads*, por exemplo operações do tipo *swap*, que envolvem a sincronização entre duas *threads* executoras. Ao todo, foram utilizadas 6 cargas de trabalho distintas, conforme a lista a seguir.

- Operações 100% de leitura e 0% de conflitos ($100r0c$) - Nesta carga de trabalho todas as operações realizadas são de leitura e pertencentes a classe de conflito Cnc, ou seja, comandos que conflitam com somente uma partição. Esta carga de trabalho representa o melhor caso em termos de desempenho, pois operações de leitura são leves e a ausência de conflitos não causa sincronização entre *threads* para a execução de comandos;
- Operações 90% de leitura e 1% de conflitos ($90r1c$) - Neste caso 90% das requisições são de leitura e apenas 10% das requisições são de escrita, sendo que 1% do total de escritas pertencem as classes de conflito Seq, ou seja, comandos que conflitam entre duas ou mais partições ou comandos que devem ser executados de forma sequencial.

Esta carga de trabalho representa cenários majoritariamente de leitura, com poucas atualizações e poucos conflitos;

- Operações 50% de leitura e 50% de conflitos ($50r50c$) - Semelhante à carga de trabalho anterior, contudo com uma taxa maior de escritas e de conflitos sendo metade das requisições de leitura e metade, escrita. Das operações de escrita, 50% provocam conflitos entre partições;
- Operações 0% de leitura e 50% de conflitos ($0r50c$) - Carga composta somente por operações de escritas, sendo que metade das operações devem ser executadas sequencialmente, ou seja, classe de conflito Seq;
- Operações 0% de leitura e 100% de conflitos ($0r100c$) - Carga com geração apenas de operações de escrita, onde todas as operações geram conflito entre partições. Este cenário representa o pior caso em termos de sincronização, pois todas as requisições implicam em coordenação entre *threads*;
- Operações 0% de leitura e 0% de conflitos ($0r0c$) - Carga composta somente por escritas, com ausência de conflitos.

Para as cargas de trabalho definidas neste trabalho, as operações de leitura não causam conflitos entre partições, portanto uma relação de conflito envolvendo múltiplas partições pode ocorrer apenas na execução de operações de escrita sobre múltiplos dados. A fim de avaliar como o período entre *checkpoints* afeta o desempenho do sistema, todas as cargas de trabalho descritas anteriormente foram avaliadas com intervalos de *checkpoint*, Δ_{cp}, a cada 50, 100, 150 e 200 mil comandos. Dessa forma, considerando i o i-ésimo comando recebido pela réplica do serviço, sempre que i % Δ_{cp} = 0, o procedimento de *checkpoint* é executado.

5.4 Avaliação de desempenho

Para comparar o impacto no desempenho causado pelas técnicas de *checkpointing* tradicional e particionado, foram realizados experimentos com ambas as estratégias. Na estratégia tradicional, um *checkpoint* contendo uma imagem com o estado completo do serviço é realizado a cada Δ_{cp} requisições. Na abordagem com *checkpointing* particionado, uma partição é salva a cada Δ_{cp} requisições. Todos os experimentos foram realizados utilizando as configurações de 4 e 8 *threads* executoras, com 4 e 8 partições, respectivamente. Para efeitos de concisão, serão discutidos a seguir os resultados considerando as cargas de trabalho $100r0c$, $0r0c$ e $90r1c$. Enquanto a primeira e segunda exibem o melhor e pior caso em termos de custos com sincronização, a terceira exibe um cenário com a execução de comandos que geram conflitos entre partições a uma taxa de 1%. O restante dos

resultados obtidos, considerando as demais cargas de trabalho apresentadas, encontram-se nos Apêndices A e B.

Os experimentos foram repetidos um total de 10 vezes. A Tabela 1 mostra as médias, desvio padrão e intervalo de confiança obtidos para as execuções onde todas as operações são de 90% escrita e 1% de conflito entre requisições de escrita, com intervalos de *checkpoint* a cada 150 mil comandos para as estratégias de *checkpointing* tradicional e com *checkpoints* particionados. A partir destas amostras foram calculadas as médias, o desvio padrão e os intervalos de confiança para confiabilidade de 95%. Podemos observar que o modelo tradicional de *checkpointing* apresenta intervalo de confiança, ao nível 95%, entre $10341,71718$ e $10532,70782$, enquanto a técnica de *checkpoints* particionados apresenta intervalo de confiança entre $16190,98357$ e $16887,06976$. Pode-se afirmar que, se repetidos estes experimentos existe 95% de confiabilidade nos resultados apresentados. Os resultados apresentados nas Figuras 12 -18 apresentam os resultados de uma das 10 execuções realizadas para cada experimento.

Execução	Tradicional	Particionado
1	10547,34	15072,35
2	10607,76	16260,55
3	10344,30	16634,95
4	10237,60	16768,79
5	10366,71	16451,00
6	10670,30	16828,73
7	10262,50	16864,32
8	10297,01	16684,72
9	10541,75	17070,18
10	10496,80	16754,64
Média	10437,21	16539,02
Desvio padrão	154,07	561,54
Intervalo de Confiança	±95,49	±348,04

Tabela 1 – Resultados para 10 repetições do caso $90r1c$.

A Figura 12 mostra como a periodicidade de *checkpoints*, indicada no eixo horizontal, afeta a vazão média, indicada no eixo vertical, para o caso $100r0c$, ou seja, carga de trabalho com 100% de operações de leitura e nenhum conflito entre partições O gráfico exibe a vazão em função do intervalo de *checkpoints* para as seguinte configurações das réplicas: Tradicional (técnica de *checkpoint* tradicional), Particionado (técnica de *checkpoint* particionado), GP-Tradicional (técnica de *checkpoint* tradicional e conflitos envolvendo grupos disjuntos de partições), GP-Particionado (técnica de *checkpoint* particionado e conflitos envolvendo grupos disjuntos de partições).

Pode-se observar que a técnica de *checkpointing* particionado apresentou melhor desempenho do que a técnica tradicional em todos os cenários avaliados. Nota-se, ainda, que com *checkpoints* mais espaçados, a estratégia de *checkpointing* tradicional apresenta

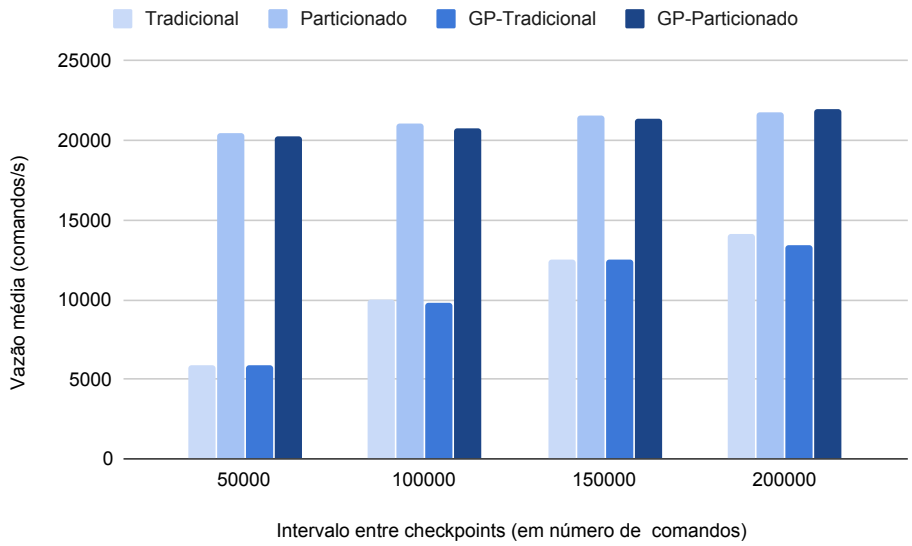


Figura 12 – Vazão média para diferentes intervalos de *checkpoint*, usando a carga de trabalho 100r0c e 4 partições.

maior vazão, como esperado. Contudo, quanto menor o intervalo entre *checkpoints*, menos custoso se torna o reprocessamento do *log* em uma eventual recuperação. Note que com *checkpoints* particionados mesmo com intervalos de *checkpoint* a cada 50.000 comandos, a vazão é aproximadamente 4 vezes superior à vazão obtida com a técnica tradicional. Mesmo ao considerar intervalos entre *checkpoints* mais espaçados, como 200.000 comandos, a técnica ainda apresenta ganho com relação à técnica tradicional. Esse resultado indica que pode ser vantajoso adotar uma periodicidade mais frequente de *checkpoints* com a abordagem proposta. Ao comparar estratégias com conflitos envolvendo quaisquer partições ou grupos disjuntos de partições, não se percebe diferenças significativas, uma vez que a carga de trabalho gerada ($100r0c$) não causa nenhum conflito.

Na Figura 13 são apresentados os resultados de vazão em função do intervalo de *checkpoints* obtidos com o uso de Cópia na Escrita. Esta técnica reduz a sincronização necessária ao fazer cópias do estado durante a criação de um *checkpoint*, permitindo que novos comandos sejam executados paralelamente ao salvamento de estado. Porém, uma grande quantidade de memória pode ser necessária para a duplicação de dados durante o salvamento de estado. O gráfico exibe a vazão em função do intervalo de *checkpoints* para as seguinte configurações das réplicas: Tradicional-CnE (técnica de *checkpoint* tradicional com CnE), Particionado-CnE (técnica de *checkpoint* particionado com CnE), GP-Tradicional-CnE (técnica de *checkpoint* tradicional com CnE e conflitos envolvendo grupos disjuntos de partições), GP-Particionado-CnE (técnica de *checkpoint* particionado com CnE e conflitos envolvendo grupos disjuntos de partições).

O uso de Cópia na Escrita beneficiou especialmente a estratégia de *checkpoint* tradicional. Em comparação com a Figura 12, a técnica tradicional apresentou maior vazão

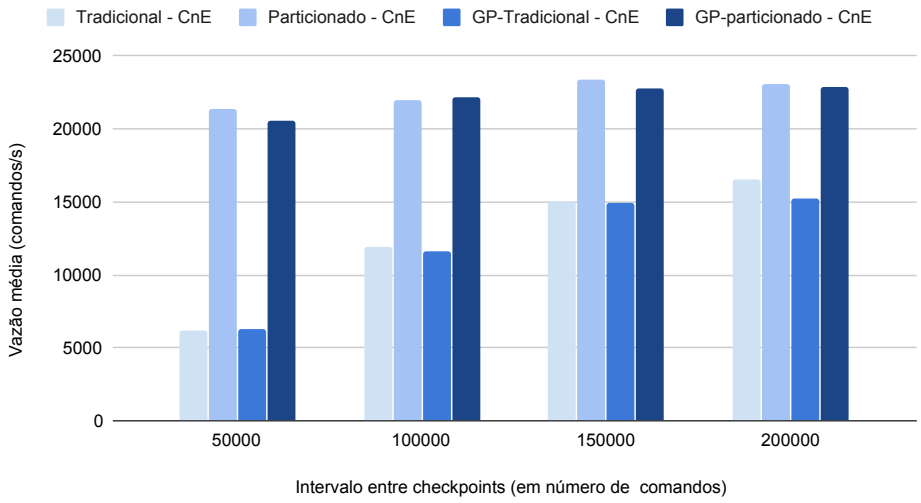


Figura 13 – Vazão média para diferentes intervalos de *checkpoint* utilizando CnE com a carga de trabalho 100r0c e 4 partições.

para todos os cenários com intervalos de *checkpoint* iguais ou superiores a 100.000 comandos. Esta otimização proporcionou um leve ganho para os casos onde a técnica de *checkpointing* particionado foi utilizado. Nos casos envolvendo a técnica tradicional o ganho é mais evidente. Durante a realização deste trabalho foi realizado um experimento comparando diferentes configurações de *heap*. Foi possível constatar que a técnica tradicional necessita de uma maior quantidade de memória para realização das cópias do estado, fazendo com que o sistema falhasse sempre que a configuração de *heap* era inferior a 6.5 GB. Já a técnica de *checkpoints* particionados necessita uma quantidade menor de memória, pois são realizadas cópias de partições do estado da aplicação, tendo sido necessário configurar a *heap* de memória em 4.5 GB para que o sistema pudesse operar normalmente. Os resultados com o uso da técnica de Cópia na Escrita para os próximos cenários avaliados não são discutidos nesta seção, mas podem ser consultados nos Apêndices A e B. Em geral, a técnica traz uma melhoria de desempenho em ambas as abordagens, e pode ser considerada uma otimização ortogonal à proposta pelo particionamento de *checkpoints*.

A Figura 14 apresenta o resultado para o cenário $0r0c$, onde todas operações geradas pelos clientes são de escrita. Pode-se observar que, devido à ausência de conflitos, a técnica de *checkpointing* particionado apresenta maior vazão média comparando com o modelo tradicional. Ainda, percebe-se que a vazão média da técnica de *checkpointing* particionado pouco se altera com variações no período de *checkpoint*.

A Figura 15 mostra os resultados obtidos com a carga de trabalho $0r100c$. Este cenário indica o caso mais custoso em termos de sincronização, visto que 100% dos comandos envolvem mais de uma partição. Pode-se notar que a vazão média é menor em relação ao caso anterior, com a execução da carga de trabalho $100r0c$. Este fato deve-se aos comandos de escrita, que são mais custosos que leituras e à existência de conflito entre

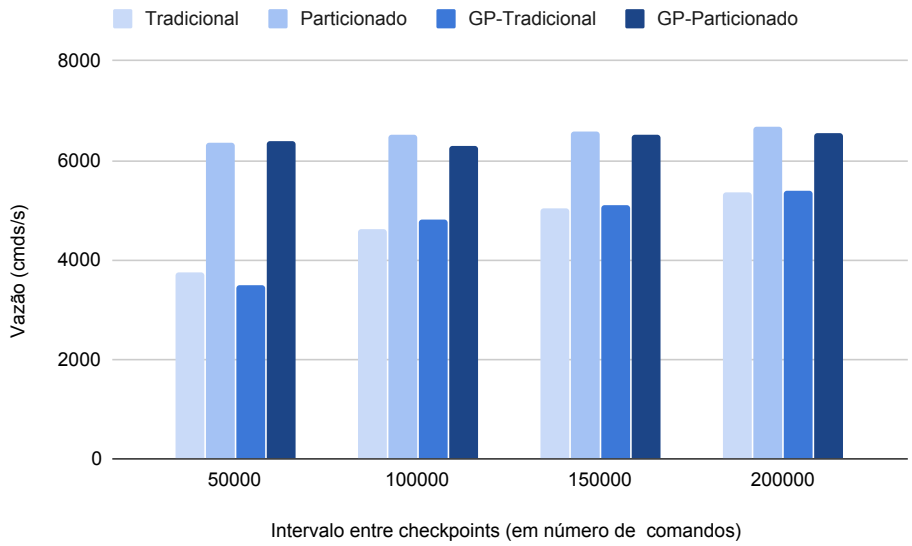


Figura 14 – Vazão média para diferentes intervalos de *checkpoint*, utilizando a carga de trabalho $0r0c$ e 4 partições.

partições, forçando o sincronismo entre *threads* executoras.

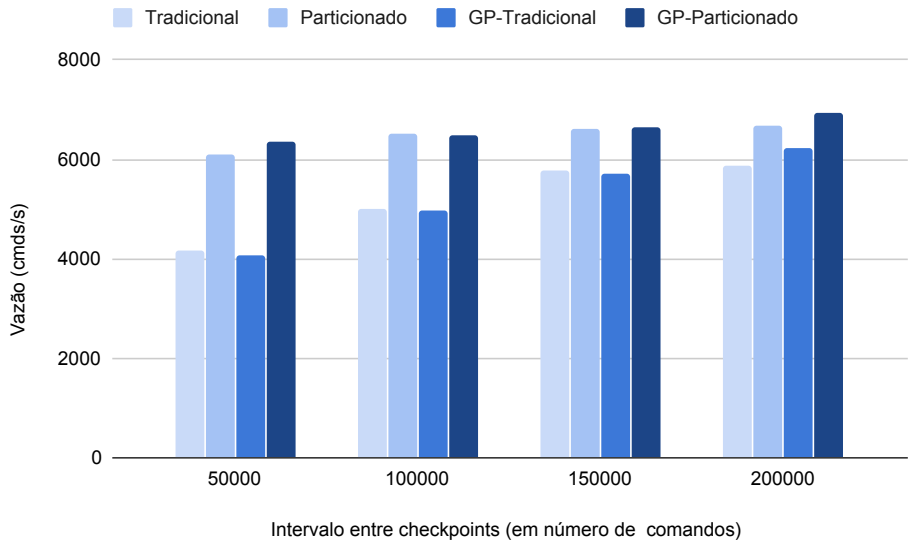


Figura 15 – Vazão média para diferentes intervalos de *checkpoint*, usando a carga de trabalho $0r100c$ e 4 partições.

Conforme observado na figura, a estratégia de *checkpointing* particionado (indicada por "Particionado") apresenta desempenho superior à técnica tradicional (indicada por "Tradicional"). Apesar da necessidade de antecipação de *checkpoints* de outras partições a cada intervalo de *checkpoints*, a técnica de *checkpoints* particionados apresenta ganho, pois as porções do estado são armazenadas em paralelo. Pode-se observar que a técnica de *checkpoints* particionados, onde conflitos envolvem grupos disjuntos de partições, o desempenho foi levemente superior ao da técnica com checkpoints particionados para os

mesmos intervalos Em geral, mesmo com a ocorrência de conflitos, é possível observar ganhos de forma consistente ao utilizar a técnica de *checkpointing* particionado. Porém, pode-se relatar que, para intervalos mais espaçados as vazões são semelhantes. Entretanto, o custo para recuperação com intervalos mais espaçados é maior, pois uma maior quantidade de comandos encontra-se nos *logs*. Dessa forma, a técnica com *checkpoiting* particionado é mais eficiente do que o modelo tradicional.

São apresentados, na Figura 17, os resultados obtidos para a carga de trabalho $90r1c$. Neste cenário 90% das operações são de leituras e foi adicionada uma taxa de 1% de conflito entre as operações de escrita. É notável que a vazão média seja um pouco menor em comparação ao melhor caso, devido à existência de operações de escrita e conflitos entre partições. Ainda assim, é perceptível que a vazão média da técnica com *checkpoints* particionados é superior ao modelo tradicional apesar da antecipação de *checkpoints* de mais de uma partição. A estratégia com *checkpointing* particionado envolvendo grupos disjuntos obteve desempenho semelhante à abordagem sem este agrupamento, indicando que o fato de executar *checkpoints* de partições em paralelo já é benéfico ao desempenho do sistema, mesmo que novos comandos tenham que aguardar para execução.

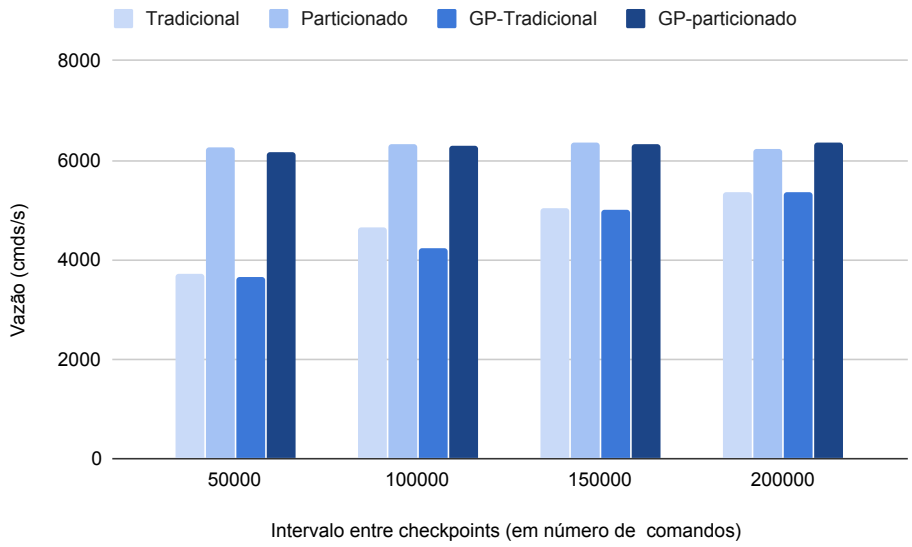


Figura 16 – Vazão média para diferentes intervalos de *checkpoint*, usando a carga de trabalho $0r0c$ e 8 partições.

Com o objetivo de avaliar como o número de *threads* executoras e partições afetam o desempenho da técnica de *checkpointing* particionado, são apresentados os resultados obtidos utilizando 8 *threads* para as cargas de trabalho $0r0c$ e $90r1c$ nas Figuras 16 e 18, respectivamente. Ao comparar o caso $0r0c$, nos cenários com 4 e 8 partições (veja Figuras 14 e 16, respectivamente), pode-se observar que o aumento no número de partições não apresentou aumento na vazão média do sistema. Este resultado é de fato esperado, pois a carga gerada durante os experimentos correspondia a aproximadamente 70% do necessário

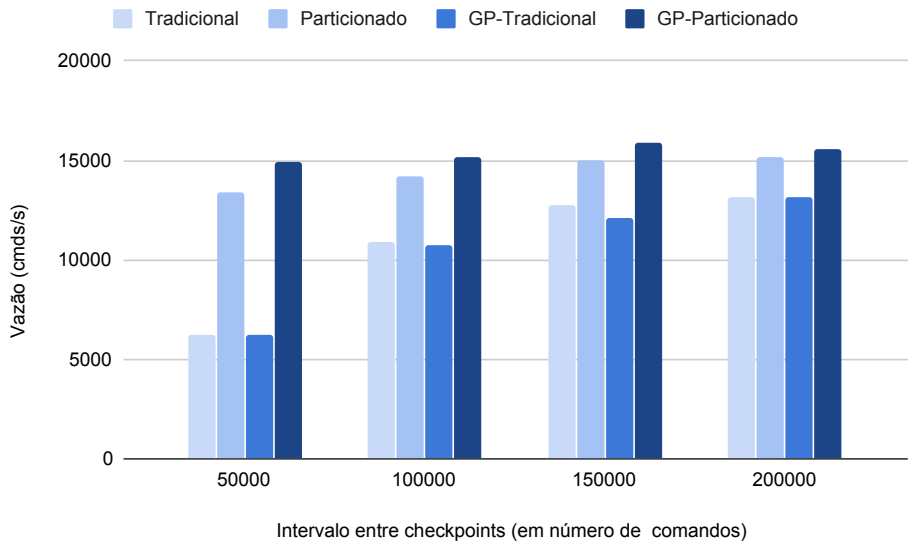


Figura 17 – Vazão média para diferentes intervalos de *checkpoint*, usando a carga de trabalho $90r1c$ e 4 partições.

para atingir o ponto de saturação em um ambiente configurado com 4 *threads* de execução. Portanto, ao executar o sistema com 8 *threads*, as réplicas estão aptas a processar cargas ainda maiores sem degradação do desempenho. Além disso, o cenário testado não exige sincronização entre *threads*. Com isso, a carga estipulada neste experimento não causa saturação em nenhum dos cenários, com 4 ou 8 *threads*.

O número de partições tem impacto na vazão em cenários onde existem conflitos, como pode ser observado ao comparar as Figuras 17 e 18. As figuras exibem os resultados da carga de trabalho $90r1c$ e com o sistema configurado com 4 e 8 partições, respectivamente. A vazão do serviço não se alterou para o caso onde os conflitos podem envolver todas as partições. Isto significa que mesmo com o sistema configurado em 8 partições, todas as partições tiveram seus *checkpoints* antecipados. Contudo é possível notar um ganho de desempenho quando os conflitos envolvem grupos disjuntos. Ao aumentar o grau de paralelismo do sistema existem mais requisições sendo executadas durante a realização dos *checkpoints*. Ainda a quantidade de partições sendo armazenadas é menor, fazendo com que o procedimento de *checkpointing* demore menos tempo para executar. Este resultado reforça a necessidade de que os dados da aplicação sejam particionados de forma a minimizar o conflito entre diferentes partições. Por exemplo, técnicas de particionamento de estado deveriam ser capazes de concentrar os conflitos em grupos disjuntos de partições, ou de implementar estratégias de reparticionamento dinâmico, visando agrupar dados com maior probabilidade de conflito em mesmas partições. Outros trabalhos indicam comparativos entre estratégias de particionamento em sistemas de alta vazão (LI; XU; KAPITZA, 2018; TROMBETA; MENDIZABAL, 2020). Embora benéficas para a nossa abordagem, estratégias para o reparticionamento do estado da aplicação fogem do escopo

deste trabalho.

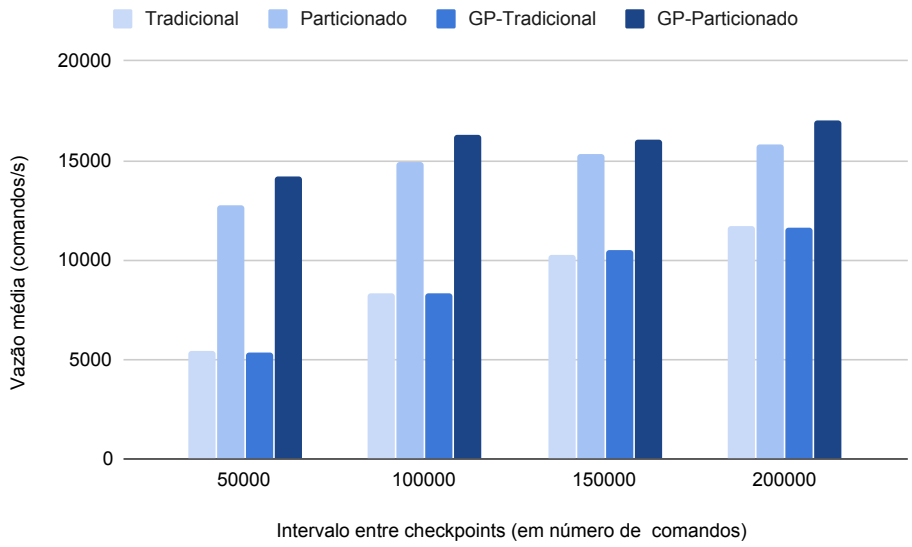


Figura 18 – Vazão média para diferentes intervalos de *checkpoint*, usando a carga de trabalho $90r1c$ e 8 partições.

A Figura 19 apresenta a latência de resposta das requisições considerando intervalo de 50000 comandos entre *checkpoints* e carga de trabalho $100r0c$. A latência é medida por requisição ao longo da execução, com valores indicados em milissegundos. Os valores de latência coletados foram medidos por um único processo cliente. Os picos de latência representam os instantes em que uma requisição ficou aguardando a realização de *checkpoint*. Por exemplo, na técnica tradicional, o salvamento de um checkpoint leva pelo menos 6s para finalizar (observe os picos próximos aos instantes 9s, 17s e 26s no gráfico). Este tempo de salvamento do estado afeta diretamente a latência observada pelos clientes. Pode-se observar que a latência observada com a técnica de *checkpointing* particionado, durante a realização de *checkpoints*, são menores em comparação a técnica tradicional. Este ganho é obtido pois a cada *checkpoint* apenas parte do estado da aplicação é armazenado. Além disso, requisições que não envolvam partições de estado sendo salvas podem executar em paralelo com a execução do procedimento de *checkpointing*.

A Figura 20 apresenta o comportamento da vazão, em comandos executados, ao longo do tempo, medido em segundos. A linha pontilhada indica a execução da técnica tradicional, enquanto a linha contínua a vazão da técnica com checkpointing particionado. Note que para a estratégia tradicional a vazão média cai para 0 no instante $5s$ e permanece por um período de aproximadamente $6s$. Durante este período a réplica do serviço armazena seu estado e para de processar novas requisições. Na estratégia de *checkpointing* particionado é possível observar um decaimento na vazão, entretanto a réplica do serviço ainda é capaz de processar novas requisições. Pode-se perceber que a técnica de *checkpoints* particionados produz menores picos de latência de resposta pois são realizados *checkpoints* de porções

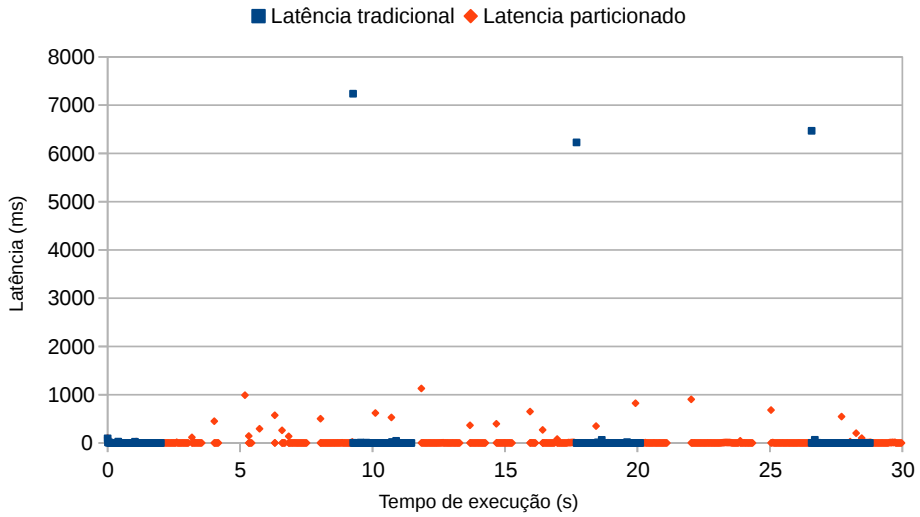


Figura 19 – Latência de resposta para intervalo de *checkpoint* a cada 50000 comandos, utilizando a carga de trabalho $90r1c$ e 4 partições.

do estado em instantes de tempo diferentes.

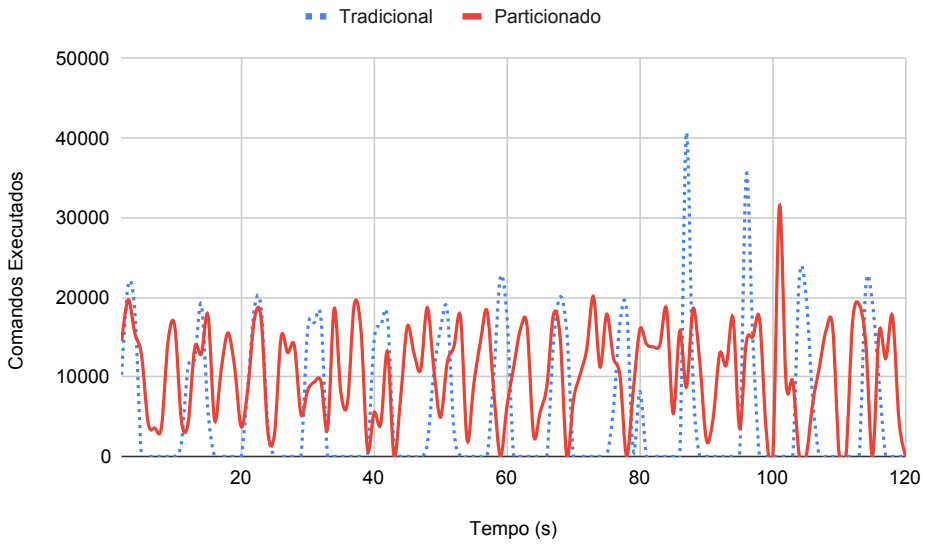


Figura 20 – Vazão do serviço ao longo do tempo com *checkpoints* a cada 50000 comandos, utilizando a carga de trabalho $90r1c$ e 4 partições.

A Tabela 2 apresenta uma simulação sobre o percentual do estado armazenado a cada *checkpoint*. Foi observada a quantidade total de conflitos com a partição P_1, considerado que a partição P_1 seria salva no *checkpoint*. Com base na matriz de conflitos mantida pela escalonador (vide Algoritmo 3) foi calculado o total de partições que tiveram comandos conflitando com a partição P_1, de forma direta ou transitiva. Na tabela são apresentados o percentual total do estado armazenado a cada *checkpoint*, utilizando intervalos entre *checkpoints* de 50000, 100000, 150000 e 200000 comandos. A coluna "Número de partições" apresenta as configurações de particionamento utilizadas. Para este

experimento foi utilizada a carga de trabalho $90r1c$, ou seja, 1% das requisições geradas envolvem pares de partições. Pode-se observar que com taxa de conflitos de 1%, para

Tabela 2 – Simulação do número de partições envolvidas antes da execução de um *checkpoint* para configurações com 4 partições e carga de trabalho $90r1c$.

Número de partições	**Percentual do estado armazenado por intervalo Δ_{cp}**			
	Δ_{cp} = 50000	Δ_{cp} = 100000	Δ_{cp} = 150000	Δ_{cp} = 200000
4	100	100	100	100
8	100	100	100	100
16	100	100	100	100
32	93,75	100	100	100
64	57,81	92,19	98,44	100
128	1,72	69,53	86,41	95,00
256	0,47	5,78	24,22	60,71

4 e 8 partições, todo o estado da aplicação precisa ser armazenado independentemente do intervalo entre *checkpoints* utilizado. Isto ocorre devido à incidência de comandos conflitantes envolvendo todas as partições antes da realização do *checkpoint*. Para 16 partições e intervalo de 50000 requisições, 87.5% do estado foi armazenado, enquanto que nos demais intervalos houve antecipação no *checkpoint* de todas as partições que compõem o estado da aplicação. Essa observação ressalta a vantagem em utilizar intervalos entre *checkpoints* curtos. Observa-se, também, que o aumento no número de partições ocasiona uma maior dispersão nos conflitos envolvendo pares de partições. Consequentemente, menos partições precisam ser armazenadas a cada *checkpoint*, possibilitando que mais comandos possam ser processados durante a realização de um *checkpoint*.

Como pode-se observar, mesmo com a incidência de uma pequena taxa da conflitos, a alta vazão a qual o sistema está submetido faz com que o salvamento do estado de uma partição force a antecipação do salvamento de outras partições. Com isso, a vazão das técnicas de *checkpointing* tradicional (indicada por "Tradicional") e particionado (indicada por "Particionado") são muito semelhantes em configurações com poucas partições, pois em ambos os casos, todo o estado é salvo a cada intervalo de *checkpoint*. Ao assumir que os conflitos envolvem grupos disjuntos de partições, observa-se um aumento considerável na vazão média do serviço (ver vazão indicada por "GP-Particionado"), sobretudo para intervalos entre *checkpoints* pequenos.

Para avaliar como o particionamento do estado afeta o procedimento de recuperação, foram executados 10 experimentos. Na Figura 21 é apresentado o resultado para uma dessas execuções. Para os resultados obtidos foi avaliado o intervalo de confiança com nível de 95%. A Tabela 3 apresenta os resultados obtidos para cada execução, o desvio padrão e o intervalo de confiança utilizando a estratégia tradicional. Já a Tabela 4 apresenta os mesmos indicadores para as 10 execuções realizadas com a estratégia de *checkpointing* particionado. Cada barra vertical indica o tempo total necessário para realiza

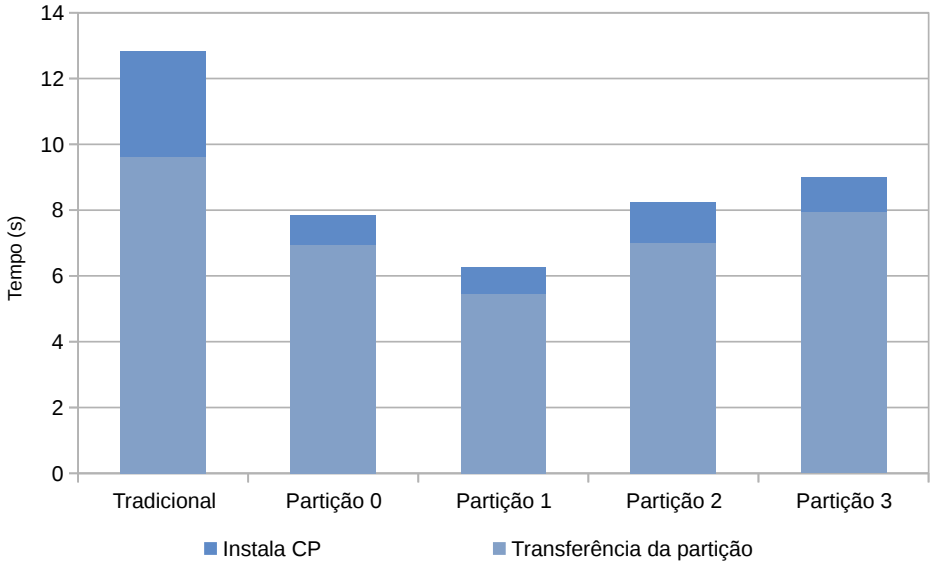


Figura 21 – Recuperação utilizando *checkpoints* particionados.

o procedimento de recuperação. Para o caso particionado são apresentados os resultados de cada partição separadamente para melhor visualização. O tempo total de recuperação foi dividido em 2 períodos. O primeiro período, *Transferência da partição*, apresenta o tempo necessário para requisitar uma partição de uma réplica correta e receber esta partição. Enquanto o período de tempo, *instala CP*, indica o tempo necessário para instalação do *checkpoint* recebido.

Execução	Recebimento do estado	Instalação do checkpoint
1	10,7960	3,4106
2	10,9550	3,0090
3	11,3671	3,0802
4	16,1088	2,9051
5	15,9596	3,2413
6	18,1926	2,8858
7	15,2954	3,0234
8	14,8672	3,0578
9	16,4710	2,7238
10	14,0183	3,1158
Média	14,4031	3,0453
Desvio padrão	2,565720391	0,1910
Intervalo de Confiança	±1,5902	±0,1184

Tabela 3 – Tempo de recuperação para a estratégia tradicional.

Pode-se perceber que a técnica de *checkpoints* particionados proporciona menor tempo necessário para recuperação. Pode-se observar que o tempo necessário para que a última partição (partição 4) finalize a recuperação é menor que o da técnica tradicional.

	Média		**Desvio Padrão**		**Intervalo de Confiança**	
	Transferência	Instalação	Transferência	Instalação	Transferência	Instalação
Partição 0	8,0625	1,2590	0,9345	0,1743	±0,5791	±0,1080
Partição 1	7,7747	1,2226	1,1395	0,2548	±0,7062	±0,1579
Partição 2	7,2822	1,2062	1,3911	0,2895	±0,8622	±0,1794
Partição 3	8,6639	8,6639	0,9839	0,2783	±0,6098	±0,1725

Tabela 4 – Tempo de recuperação para a estratégia de *checkpointing* particionado.

Enquanto a recuperação tradicional necessitou de aproximadamente 13 segundos, a técnica de *checkpoints* particionados precisou cerca de 9 segundos para finalizar o procedimento de recuperação. Pode-se observar a sobreposição de tarefas, por exemplo, a recuperação das partições 0 e 1 são finalizadas enquanto os *checkpoints* das partições 2 e 3 ainda não foram recebidos. Tal sobreposição de tarefas implica em menor tempo de recuperação. É notável que o tempo necessário para receber o estado completo, no modelo tradicional, é semelhante ao tempo de recebimento de uma partição. Este efeito acontece pois embora sejam transmitidos fragmentos do estado as *threads* responsáveis por receberem as partições competem acesso ao meio de acesso físico. Contudo, o desempenho poderia ser melhorado utilizando mais de um barramento de rede, tanto para recebimento, quanto para envio de fragmentos do estado. Desta forma seria possível diminuir o número de *threads* enviando e recebendo através de uma mesma interface de rede.

6 Conclusão

Atualmente a grande demanda sobre disponibilidade e resposta de sistemas providos através da Internet torna necessária a utilização de técnicas que mantenham estes serviços responsivos. Com o uso de técnicas de replicação, é possível que um sistema mantenha-se operacional a despeito de uma certa quantidade de falhas das réplicas do serviço, além de oferecer melhor escalabilidade ao serviço replicado.

Neste trabalho foi apresentado o modelo de Replicação Máquina de Estados Paralela (RMEP). Com esse modelo é possível garantir o funcionamento correto do serviço replicado e aumentar a vazão de sistemas permitindo a execução paralela de alguns comandos. Conforme discutido no Capítulo 2, embora existam diversas propostas de modelo de RMEP, pouco tem se discutido sobre aspectos de durabilidade, como *checkpointing* e recuperação neste contexto. Neste dissertação, foi feito um levantamento do estado da arte e foram discutidas estratégias para implementação de mecanismos de recuperação, em especial *checkpointing*. Foram discutidas diferentes abordagens que buscam minimizar o impacto causado no desempenho do serviço durante a realização de um *checkpoint*.

Como resultado da pesquisa realizada, foi apresentada uma nova abordagem que busca reduzir o impacto no desempenho do serviço, através do particionamento do estado da aplicação e da defasagem no salvamento de estados para as diferentes partições. O desempenho desta nova abordagem foi avaliado frente ao modelo tradicional de *checkpointing*. Foram observados os resultados para vazão média do serviço e o impacto causado na latência de resposta ao cliente considerando diferentes cargas de trabalho, que variam desde os casos sem conflitos, que são largamente favorecidos pela técnica, até onde todas as requisições envolvem acessos à todas as partições. Neste segundo caso, o salvamento do estado de uma partição antecipa o salvamento das demais. Ainda assim, a execução paralela dos procedimentos de *checkpointing* demonstrou menor interferência na execução das réplicas. Portanto, a estratégia proposta mostrou-se mais eficiente do que a técnica tradicional, trazendo ganhos em todos os cenários avaliados.

Apesar de apresentar ganhos em comparação a técnica tradicional, a técnica de *checkpoints* particionados necessita que os dados da aplicação sejam particionados de maneira que conflitos entre partições sejam minimizados. Conforme apresentado, foi possível identificar aumento na vazão do serviço ao agrupar conflitos entre no máximo duas partições. Esta observação sugere que o uso de técnicas de reparticionamento de estado poderiam contribuir para manter requisições conflitantes restritas a um subconjunto de partições e, assim, oferecendo melhor desempenho. Este aspecto foge do escopo de pesquisa do trabalho, mas pode ser investigado de forma complementar ao estudo apresentando.

6.1 Trabalhos Futuros

Neste trabalho foram fixados alguns parâmetros de configuração do sistema. O primeiro deles é o tamanho das partições em que o estado da aplicação foi dividido. A técnica de *checkpointing* particionado apresentada considera que os estados sejam particionados em tamanhos iguais (em *MB*) e que estes tamanhos não se modifiquem. Não foi avaliado como o desempenho da técnica é afetado quando os tamanhos das partições variam conforme a execução. Considerar partições de tamanhos diferentes implicaria em tempos mais longos para realização dos *checkpoints* das partições maiores.

A carga de trabalho gerada para avaliação de desempenho é uniformemente distribuída, ou seja, a quantidade de requisições que operam sobre as partições do estado do serviço são semelhantes. Observar cenários onde a carga de trabalho não seja uniformemente distribuída pode apresentar desvantagens na técnica de *checkpoints* particionados. Partições modificadas mais constantemente realizariam *checkpoints* com maior frequência e como a carga estaria mais concentrada nestas partições o desempenho da técnica poderia ser reduzido. Constatar a redução no desempenho da técnica de *checkpoints* particionados reforçaria a utilização de mecanismos de reparticionamento de estado. Através do reparticionamento do estado seria possível distribuir de maneira mais uniforme a carga sobre as partições do serviço.

Referências

ALCHIERI, E. et al. Boosting state machine replication with concurrent execution. In: IEEE. *2018 Eighth Latin-American Symposium on Dependable Computing (LADC)*. [S.l.], 2018. p. 77–86. Citado 3 vezes nas páginas 17, 22 e 23.

ALCHIERI, E. et al. Reconfiguring parallel state machine replication. In: IEEE. *2017 IEEE 36th Symposium on Reliable Distributed Systems (SRDS)*. [S.l.], 2017. p. 104–113. Citado 5 vezes nas páginas 17, 20, 23, 50 e 51.

BELLARE, M.; MICCIANCIO, D. A new paradigm for collision-free hashing: Incrementality at reduced cost. In: SPRINGER. *International Conference on the Theory and Applications of Cryptographic Techniques*. [S.l.], 1997. p. 163–192. Citado na página 32.

BESSANI, A.; SOUSA, J.; ALCHIERI, E. E. State machine replication for the masses with bft-smart. In: IEEE. *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. [S.l.], 2014. p. 355–362. Citado 2 vezes nas páginas 17 e 20.

BESSANI, A. N. et al. On the efficiency of durable state machine replication. In: *USENIX Annual Technical Conference*. [S.l.: s.n.], 2013. p. 169–180. Citado 3 vezes nas páginas 18, 19 e 28.

BURROWS, M. The chubby lock service for loosely-coupled distributed systems. In: *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*. Berkeley, CA, USA: USENIX Association, 2006. (OSDI '06), p. 335–350. ISBN 1-931971-47-1. Disponível em: <http://dl.acm.org/citation.cfm?id=1298455.1298487>. Citado na página 17.

CASTRO, M.; LISKOV, B. Proactive recovery in a byzantine-fault-tolerant system. In: USENIX ASSOCIATION. *Proceedings of the 4th conference on Symposium on Operating System Design & Implementation-Volume 4*. [S.l.], 2000. p. 19. Citado 2 vezes nas páginas 19 e 31.

CHEN, J. et al. Copss: An efficient content oriented publish/subscribe system. In: IEEE. *2011 ACM/IEEE Seventh Symposium on Architectures for Networking and Communications Systems*. [S.l.], 2011. p. 99–110. Citado na página 49.

CHENG, D. et al. Adaptive scheduling of parallel jobs in spark streaming. In: IEEE. *IEEE INFOCOM 2017-IEEE Conference on Computer Communications*. [S.l.], 2017. p. 1–9. Citado na página 49.

CLEMENT, A. et al. Upright cluster services. In: *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*. New York, NY, USA: ACM, 2009. (SOSP '09), p. 277–290. ISBN 978-1-60558-752-3. Disponível em: <http://doi.acm.org/10.1145/1629575.1629602>. Citado 2 vezes nas páginas 19 e 33.

CNBC. *Internal documents show how Amazon scrambled to fix Prime Day glitches*. 2018. Disponível em: <https://www.cnbc.com/2018/07/19/amazon-internal-documents-what-caused-prime-day-crash-company-scramble.html>. Acesso em: 25 jul. 2020. Citado na página 17.

Coelho, P.; Pedone, F. Geographic state machine replication. In: *2018 IEEE 37th Symposium on Reliable Distributed Systems (SRDS)*. [S.l.: s.n.], 2018. p. 221–230. Citado na página 49.

CORBETT, J. C. et al. Spanner: Google's globally-distributed database. In: *OSDI*. [S.l.: s.n.], 2012. Citado na página 17.

CURINO, C. et al. Schism: a workload-driven approach to database replication and partitioning. *Proceedings of the VLDB Endowment*, VLDB Endowment, v. 3, n. 1-2, p. 48–57, 2010. Citado na página 49.

DEAN, J. Designs, lessons and advice from building large distributed systems. 2009. Citado na página 17.

ELNOZAHY, E. N. et al. A survey of rollback-recovery protocols in message-passing systems. *ACM Computing Surveys (CSUR)*, ACM, v. 34, n. 3, p. 375–408, 2002. Citado 2 vezes nas páginas 18 e 19.

EUGSTER, P. T. et al. The many faces of publish/subscribe. *ACM computing surveys (CSUR)*, ACM New York, NY, USA, v. 35, n. 2, p. 114–131, 2003. Citado na página 49.

GUNARATHNE, T.; QIU, J.; FOX, G. Iterative mapreduce for azure cloud. *CCA11 Cloud Computing and Its Applications, Chicago, ILL*, 2011. Citado na página 17.

HERLIHY, M. P.; WING, J. M. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, ACM, v. 12, n. 3, p. 463–492, 1990. Citado na página 17.

HUNT, P. et al. Zookeeper: Wait-free coordination for internet-scale systems. In: BOSTON, MA, USA. *USENIX annual technical conference*. [S.l.], 2010. v. 8, n. 9. Citado 2 vezes nas páginas 17 e 19.

KAPRITSOS, M. et al. All about eve: Execute-verify replication for multi-core servers. In: *OSDI*. [S.l.: s.n.], 2012. v. 12, p. 237–250. Citado 5 vezes nas páginas 17, 22, 23, 26 e 30.

KOTLA, R.; DAHLIN, M. *High throughput Byzantine fault tolerance*. [S.l.]: IEEE, 2004. Citado 7 vezes nas páginas 17, 18, 22, 23, 25, 29 e 36.

KUMAR, K. A.; DESHPANDE, A.; KHULLER, S. Data placement and replica selection for improving co-location in distributed environments. *arXiv preprint arXiv:1302.4168*, 2013. Citado na página 49.

LAMPORT, L. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, ACM, v. 21, n. 7, p. 558–565, 1978. Citado na página 31.

LAMPORT, L. The part-time parliament. *ACM Trans. Comput. Syst.*, ACM, New York, NY, USA, v. 16, n. 2, p. 133–169, maio 1998. ISSN 0734-2071. Disponível em: <http://doi.acm.org/10.1145/279227.279229>. Citado 2 vezes nas páginas 19 e 22.

LAMPORT, L. et al. Paxos made simple. *ACM Sigact News*, v. 32, n. 4, p. 18–25, 2001. Citado na página 17.

LE, L. H. et al. Dynastar: Optimized dynamic partitioning for scalable state machine replication. Citado na página 49.

LI, B.; XU, W.; KAPITZA, R. Dynamic state partitioning in parallelized byzantine fault tolerance. In: IEEE. *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)*. [S.l.], 2018. p. 158–163. Citado 3 vezes nas páginas 36, 49 e 61.

MARANDI, P. J.; BEZERRA, C. E.; PEDONE, F. Rethinking state-machine replication for parallelism. In: IEEE. *2014 IEEE 34th International Conference on Distributed Computing Systems*. [S.l.], 2014. p. 368–377. Citado 6 vezes nas páginas 22, 23, 27, 29, 36 e 49.

MARANDI, P. J.; PEDONE, F. Optimistic parallel state-machine replication. In: IEEE. *2014 IEEE 33rd International Symposium on Reliable Distributed Systems*. [S.l.], 2014. p. 57–66. Citado na página 17.

MARANDI, P. J.; PRIMI, M.; PEDONE, F. Multi-ring paxos. In: IEEE. *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2012)*. [S.l.], 2012. p. 1–12. Citado na página 27.

MENDIZABAL, O. M.; DOTTI, F. L.; PEDONE, F. High performance recovery for parallel state machine replication. In: IEEE. *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*. [S.l.], 2017. p. 34–44. Citado 2 vezes nas páginas 19 e 30.

MENDIZABAL, O. M. et al. Checkpointing in parallel state-machine replication. In: SPRINGER. *International Conference on Principles of Distributed Systems*. [S.l.], 2014. p. 123–138. Citado na página 29.

MENDIZABAL, O. M. et al. Efficient and deterministic scheduling for parallel state machine replication. In: IEEE. *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. [S.l.], 2017. p. 748–757. Citado 2 vezes nas páginas 26 e 36.

ONGARO, D.; OUSTERHOUT, J. In search of an understandable consensus algorithm. In: *2014 {USENIX} Annual Technical Conference ({USENIX}{ATC} 14)*. [S.l.: s.n.], 2014. p. 305–319. Citado 2 vezes nas páginas 17 e 22.

QUAMAR, A.; KUMAR, K. A.; DESHPANDE, A. SWORD: scalable workload-aware data placement for transactional workloads. In: *Proceedings of the 16th International Conference on Extending Database Technology*. [S.l.: s.n.], 2013. p. 430–441. Citado na página 49.

RANDELL, B. System structure for software fault tolerance. *Ieee transactions on software engineering*, IEEE, n. 2, p. 220–232, 1975. Citado na página 19.

RAO, J.; SHEKITA, E. J.; TATA, S. Using paxos to build a scalable, consistent, and highly available datastore. *Proceedings of the VLDB Endowment*, VLDB Endowment, v. 4, n. 4, p. 243–254, 2011. Citado na página 34.

SACHS, K. et al. Benchmarking publish/subscribe-based messaging systems. In: SPRINGER. *International Conference on Database Systems for Advanced Applications*. [S.l.], 2010. p. 203–214. Citado na página 49.

SCHNEIDER, F. B. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys (CSUR)*, ACM, v. 22, n. 4, p. 299–319, 1990. Citado 3 vezes nas páginas 17, 22 e 31.

SINGH, A. et al. Zeno: eventually consistent byzantine-fault tolerance. In: USENIX ASSOCIATION. *Proceedings of the 6th USENIX symposium on Networked systems design and implementation*. [S.l.], 2009. p. 169–184. Citado na página 31.

TAMIR, Y.; SEQUIN, C. H. Error recovery in multicomputers using global checkpoints. In: *13th International Conference on Parallel Processing*. [S.l.: s.n.], 1984. p. 32–41. Citado na página 19.

TROMBETA, J.; MENDIZABAL, O. Simulador para medição de paralelismo em algoritmos de escalonamento para replicação máquina de estados paralela. In: *Anais da XX Escola Regional de Alto Desempenho da Região Sul*. Porto Alegre, RS, Brasil: SBC, 2020. p. 49–52. ISSN 2595-4164. Disponível em: <https://sol.sbc.org.br/index.php/eradrs-/article/view/10753>. Citado na página 61.

VIEL, E.; UEDA, H. Data stream partitioning re-optimization based on runtime dependency mining. In: IEEE. *2014 IEEE 30th International Conference on Data Engineering Workshops*. [S.l.], 2014. p. 199–206. Citado na página 49.

WANG, L. et al. pipscloud: High performance cloud computing for remote sensing big data management and processing. *Future Generation Computer Systems*, Elsevier, v. 78, p. 353–368, 2018. Citado na página 17.

WANG, Y.-M. Space reclamation for uncoordinated checkpointing in message-passing systems. ph. d. thesis. 1993. Citado na página 19.

WANG, Y.-M.; FUCHS, W. K. *Lazy checkpoint coordination for bounding rollback propagation*. [S.l.], 1993. Citado na página 19.

WHITE, B. et al. An integrated experimental environment for distributed systems and networks. *ACM SIGOPS Operating Systems Review*, ACM New York, NY, USA, v. 36, n. SI, p. 255–270, 2002. Citado na página 53.

ZHENG, W. et al. Fast databases with fast durability and recovery through multicore parallelism. In: *11th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 14)*. [S.l.: s.n.], 2014. p. 465–477. Citado 2 vezes nas páginas 19 e 35.

Apêndices

APÊNDICE A – Resultados para testes com 4 partições

Neste apêndice são apresentados os resultados obtidos para os experimentos, utilizando 4 *threads* de execução, realizados durante este trabalho. A Tabela 5 descreve o ambiente configurado para cada caso de execução. A coluna "Carga de trabalho" descreve a carga de trabalho utilizada, enquanto a coluna "Cópia na escrita", indica se foi utilizada a técnica de cópia na escrita. Não é feita nenhuma análise sobre os dados apesentados, sendo mostrados aqui com o objetivo de relatar os experimentos realizados.

Figura	Carga de trabalho	Cópia na escrita
22	100r0c	Não
23	100r0c	Sim
24	90r1c	Não
25	90r1c	Sim
26	50r50c	Não
27	50r50c	Sim
28	0r50c	Não
29	0r50c	Sim
30	0r100c	Não
31	0r100c	Sim
32	0r0c	Não
33	0r0c	Sim

Tabela 5 – Resultados dos experimentos utilizando 4 partições.

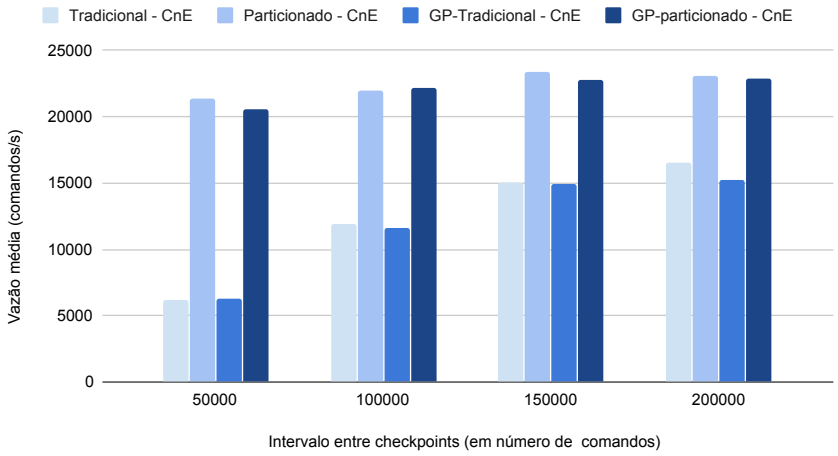


Figura 22 – Vazão média para diferentes intervalos de *checkpoint* sem utilizar CnE - 100r0c

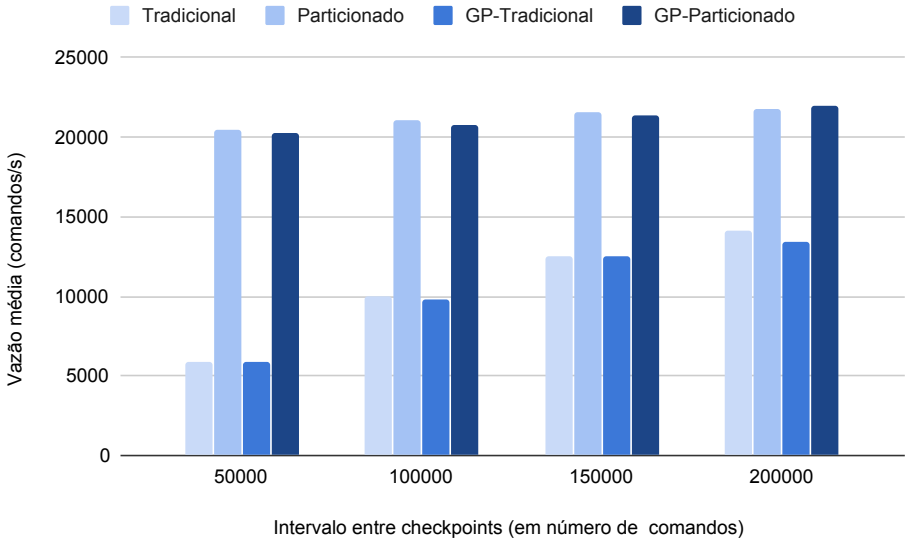


Figura 23 – Vazão média para diferentes intervalos de *checkpoint* utilizando CnE com a carga de trabalho 100r0c e 4 partições

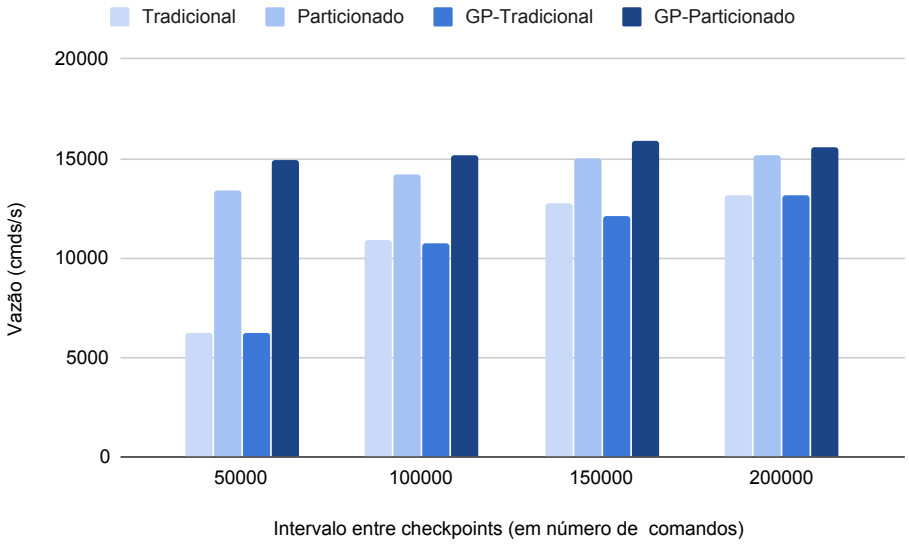


Figura 24 – Vazão média para diferentes intervalos de *checkpoint* sem utilizar CnE - 90r1c

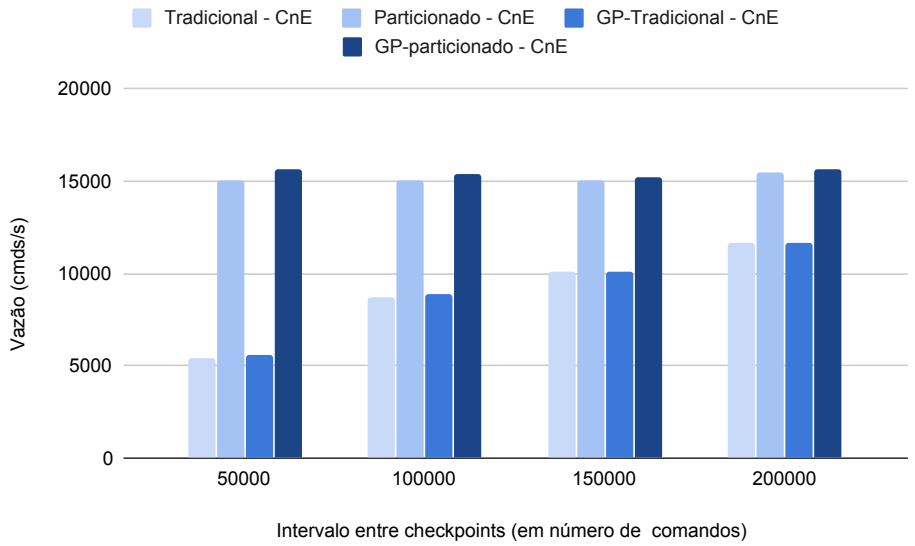


Figura 25 – Vazão média para diferentes intervalos de *checkpoint* utilizando CnE - 90r1c

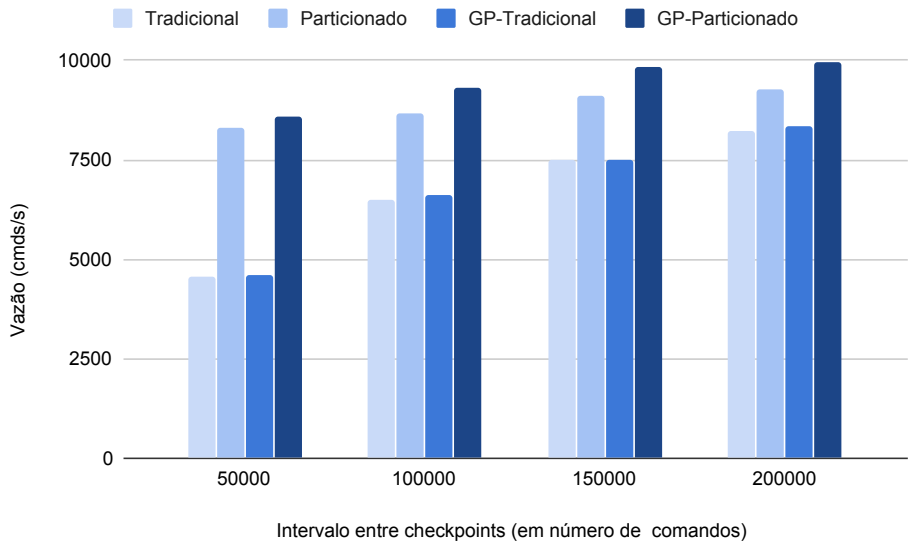


Figura 26 – Vazão média para diferentes intervalos de *checkpoint* sem utilizar CnE - 50r50c

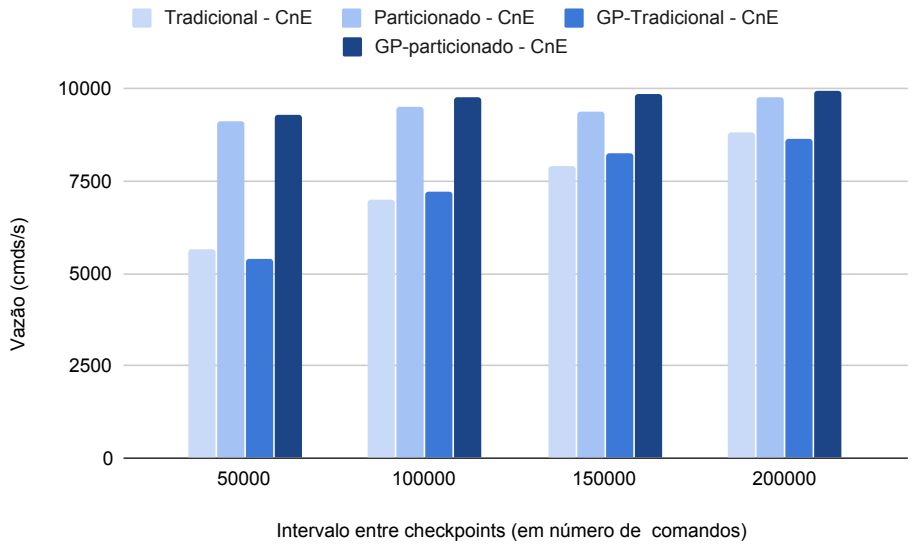


Figura 27 – Vazão média para diferentes intervalos de *checkpoint* utilizando CnE - 50r50c

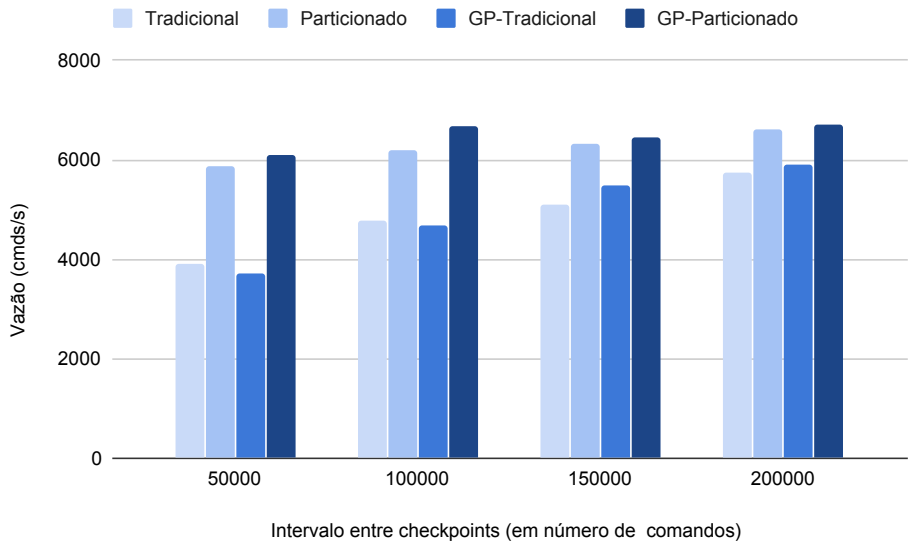


Figura 28 – Vazão média para diferentes intervalos de *checkpoint* sem utilizar CnE - 0r50c

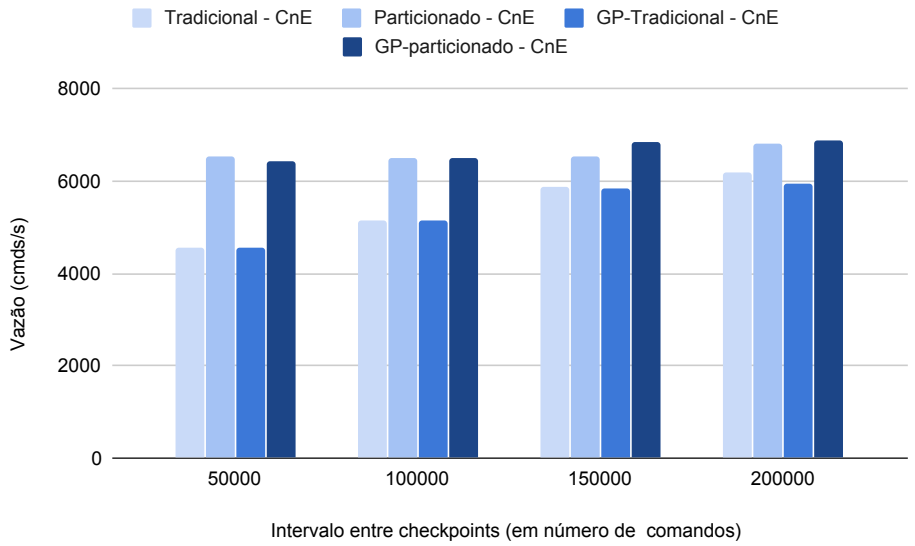


Figura 29 – Vazão média para diferentes intervalos de *checkpoint* utilizando CnE - 0r50c

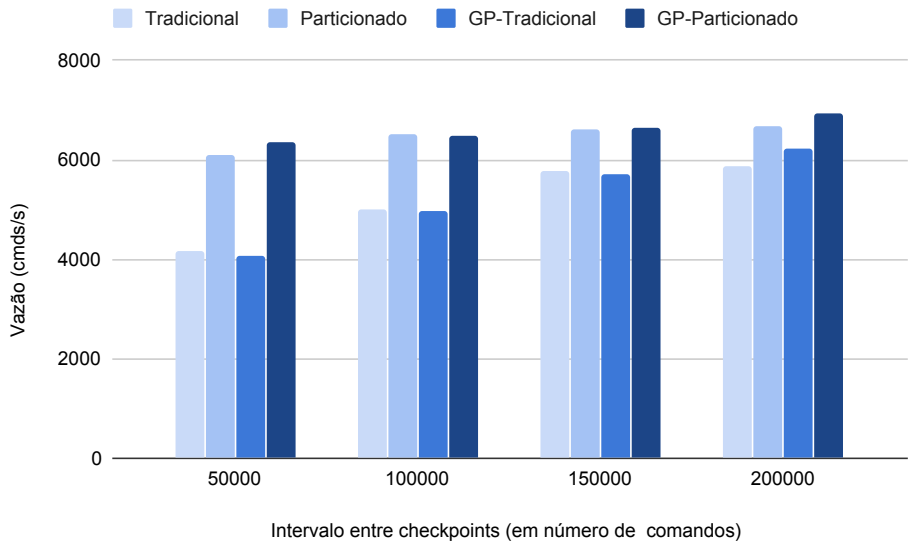


Figura 30 – Vazão média para diferentes intervalos de *checkpoint* sem utilizar CnE - 0r100c

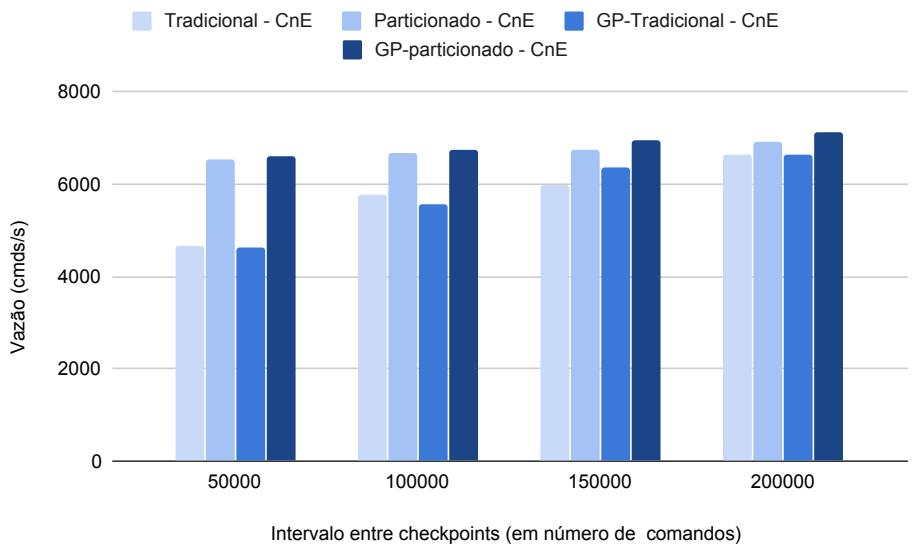


Figura 31 – Vazão média para diferentes intervalos de *checkpoint* utilizando CnE - 0r100c

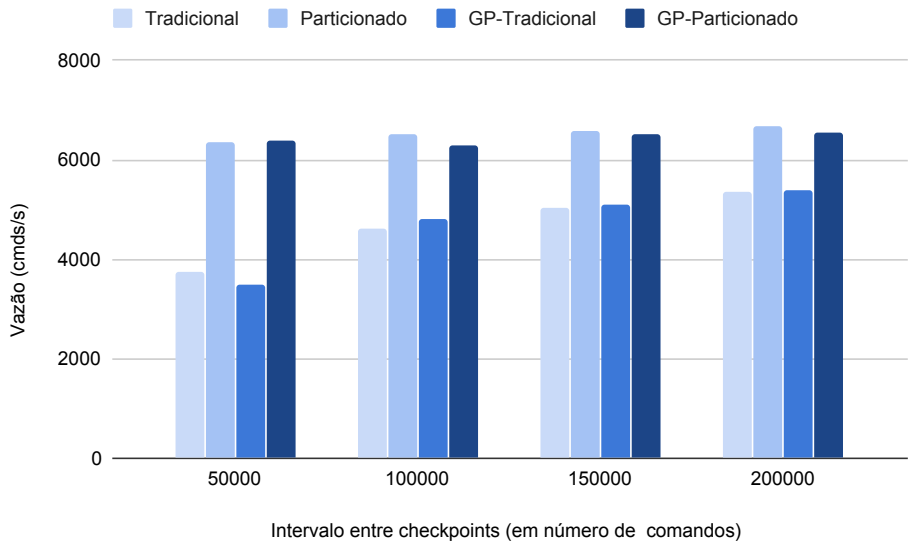


Figura 32 – Vazão média para diferentes intervalos de *checkpoint* sem utilizar CnE - 0r0c

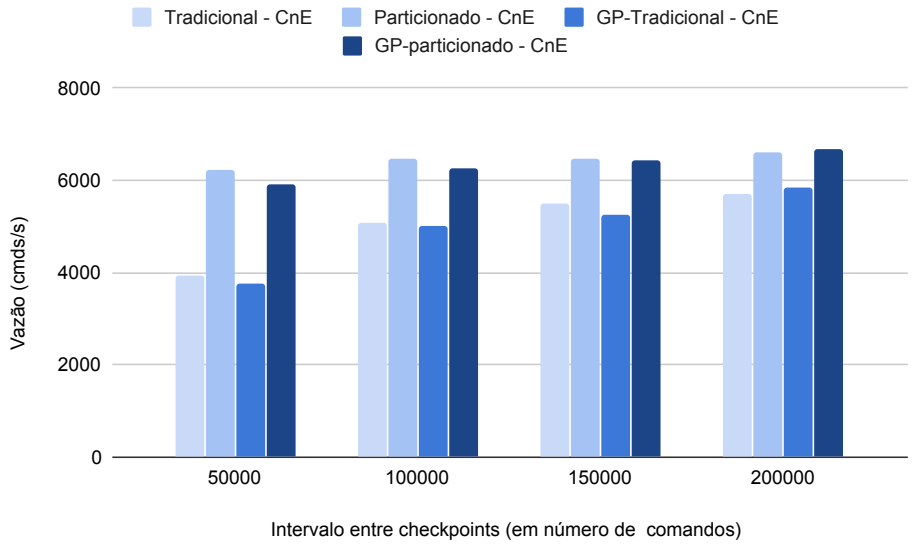


Figura 33 – Vazão média para diferentes intervalos de *checkpoint* utilizando CnE - 0r0c

APÊNDICE B – Resultados para testes com 8 partições

Neste apêndice são apresentados os resultados obtidos para os experimentos, utilizando 8 *threads* de execução, realizados durante este trabalho. A Tabela 6 descreve o ambiente configurado para cada caso de execução. A coluna "Carga de trabalho" descreve a carga de trabalho utilizada, enquanto a coluna "Cópia na escrita", indica se foi utilizada a técnica de cópia na escrita.

Não é feita nenhuma análise sobre os dados apesentados, sendo mostrados aqui com o objetivo de relatar os experimentos realizados.

Figura	Carga de trabalho	Cópia na escrita
34	100r0c	Não
35	100r0c	Sim
36	90r1c	Não
37	90r1c	Sim
38	50r50c	Não
39	50r50c	Sim
40	0r50c	Não
41	0r50c	Sim
42	0r100c	Não
43	0r100c	Sim
44	0r0c	Não
45	0r0c	Sim

Tabela 6 – Resultados dos experimentos utilizando 8 partições.

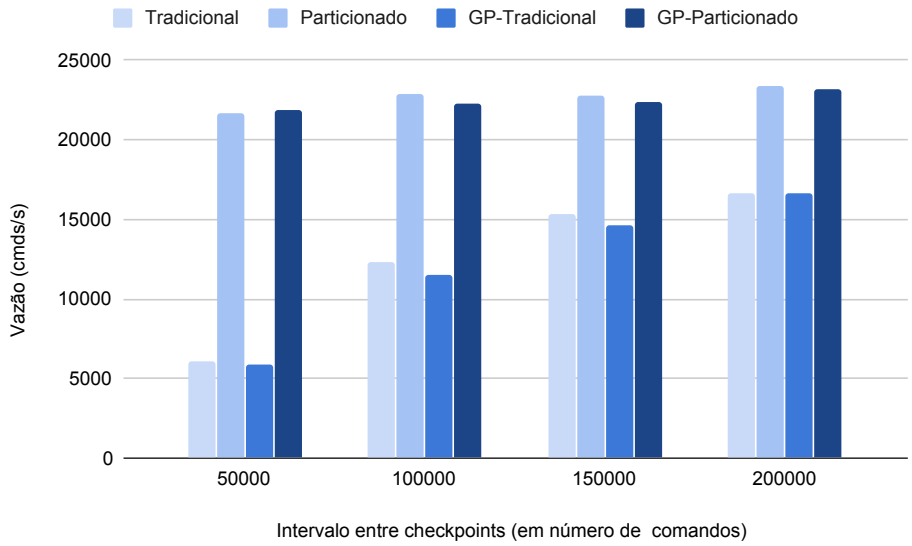


Figura 34 – Vazão média para diferentes intervalos de *checkpoint* sem utilizar CnE - 100r0c

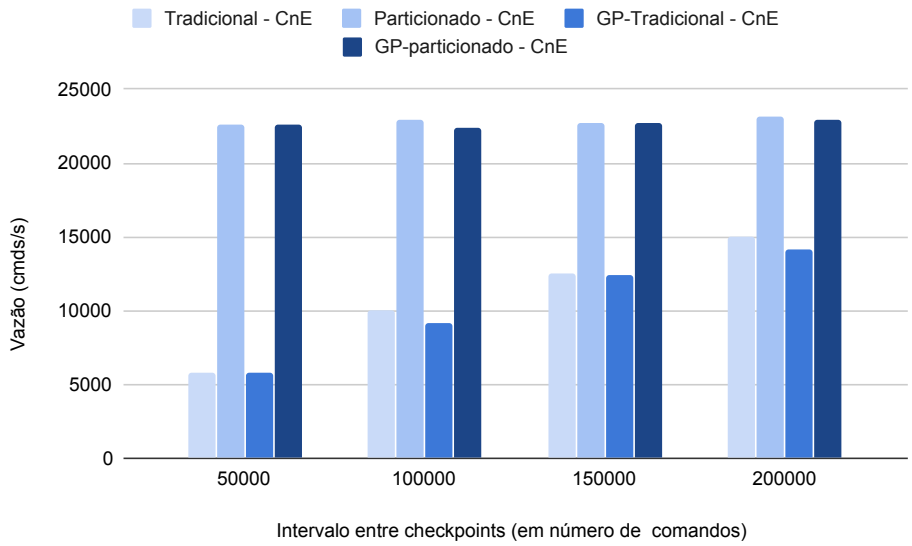


Figura 35 – Vazão média para diferentes intervalos de *checkpoint* utilizando CnE - 100r0c

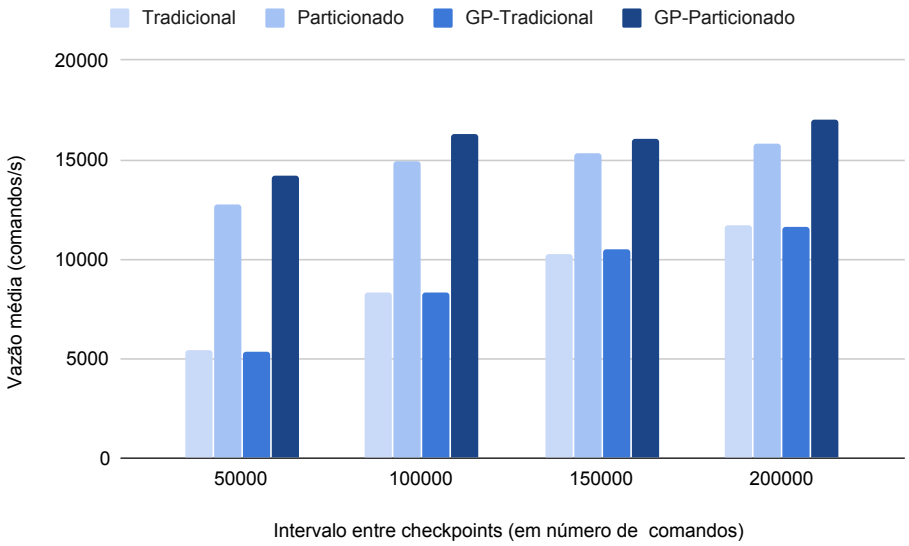


Figura 36 – Vazão média para diferentes intervalos de *checkpoint* sem utilizar CnE - 90r1c

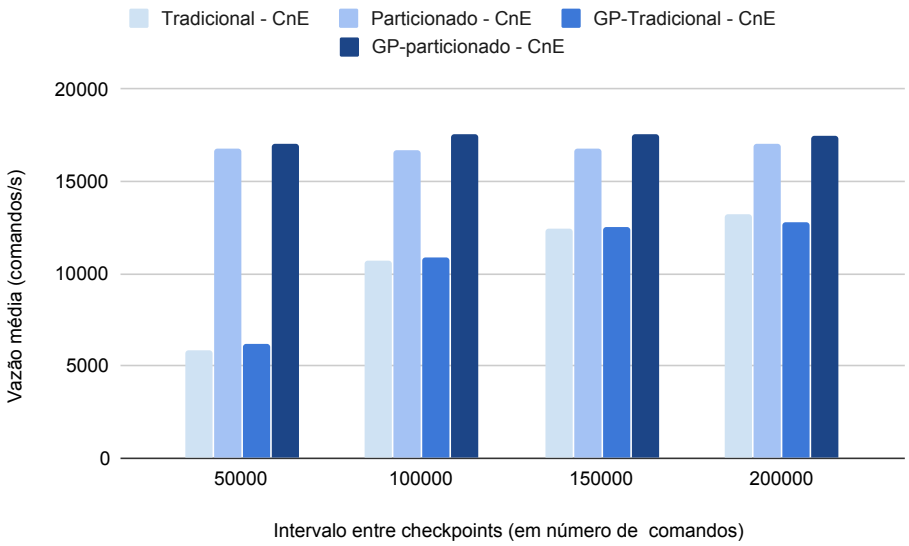


Figura 37 – Vazão média para diferentes intervalos de *checkpoint* utilizando CnE - 90r1c

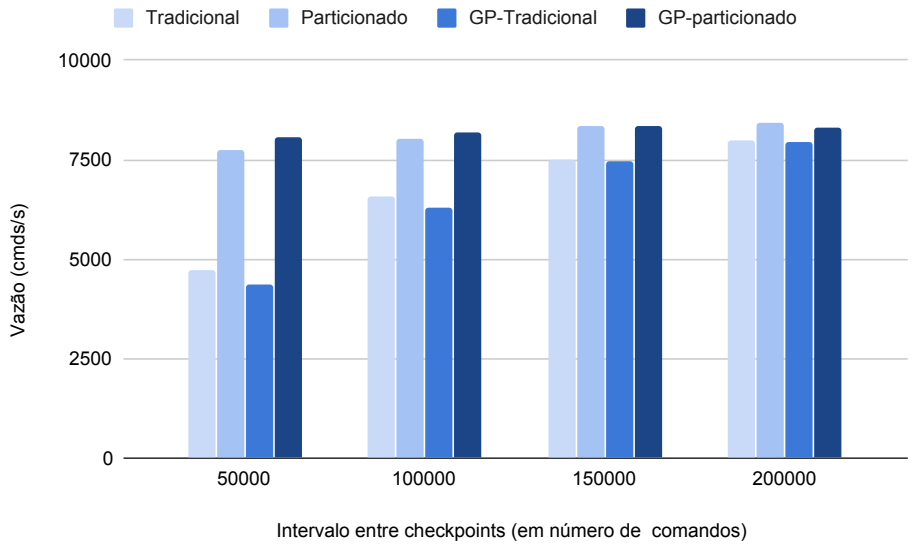


Figura 38 – Vazão média para diferentes intervalos de *checkpoint* sem utilizar CnE - 50r50c

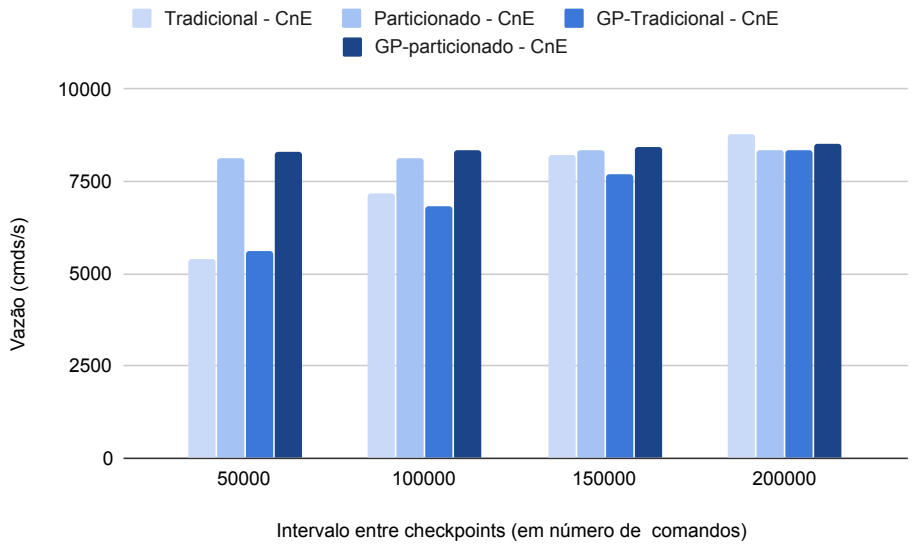


Figura 39 – Vazão média para diferentes intervalos de *checkpoint* utilizando CnE - 50r50c

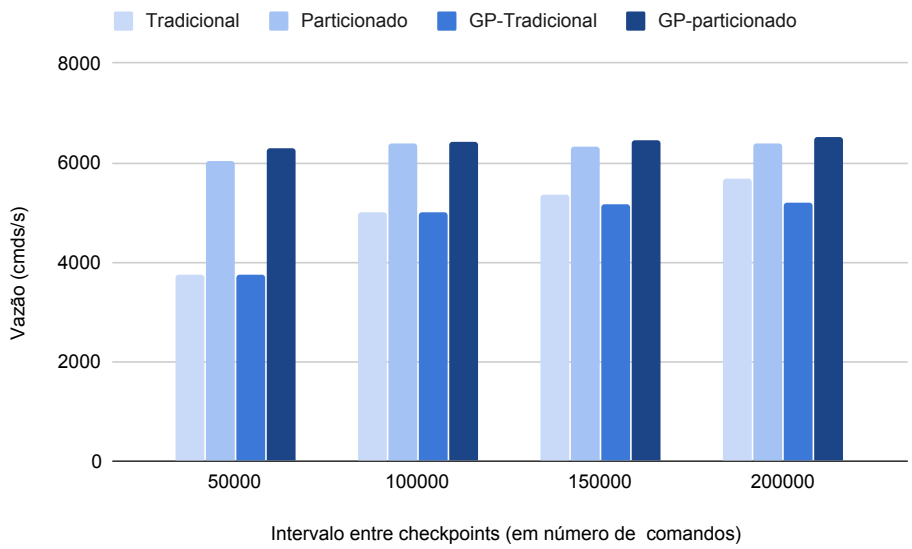


Figura 40 – Vazão média para diferentes intervalos de *checkpoint* sem utilizar CnE - 0r50c

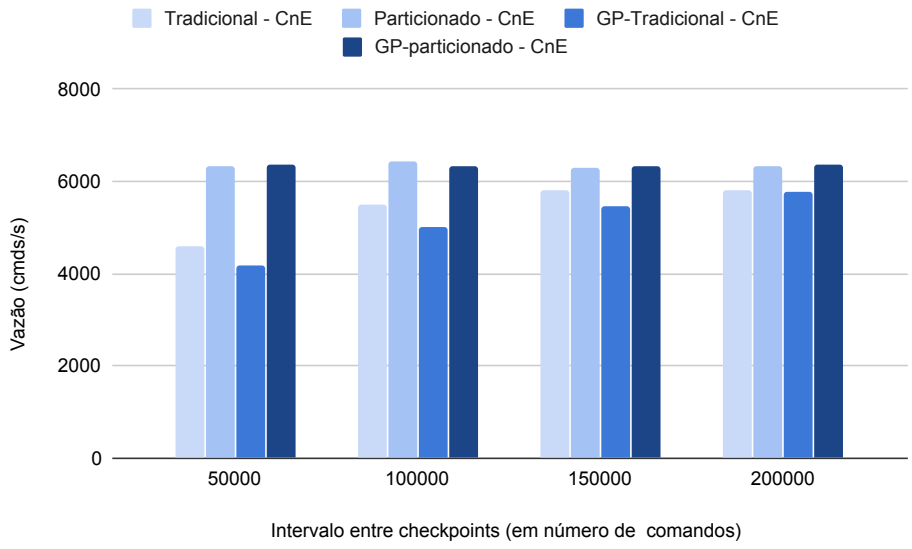


Figura 41 – Vazão média para diferentes intervalos de *checkpoint* utilizando CnE - 0r50c

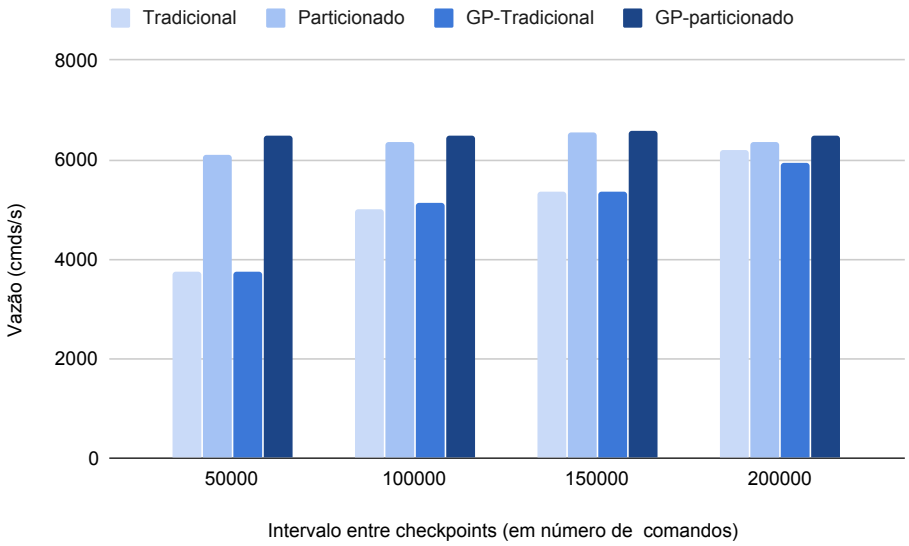


Figura 42 – Vazão média para diferentes intervalos de *checkpoint* sem utilizar CnE - 0r100c

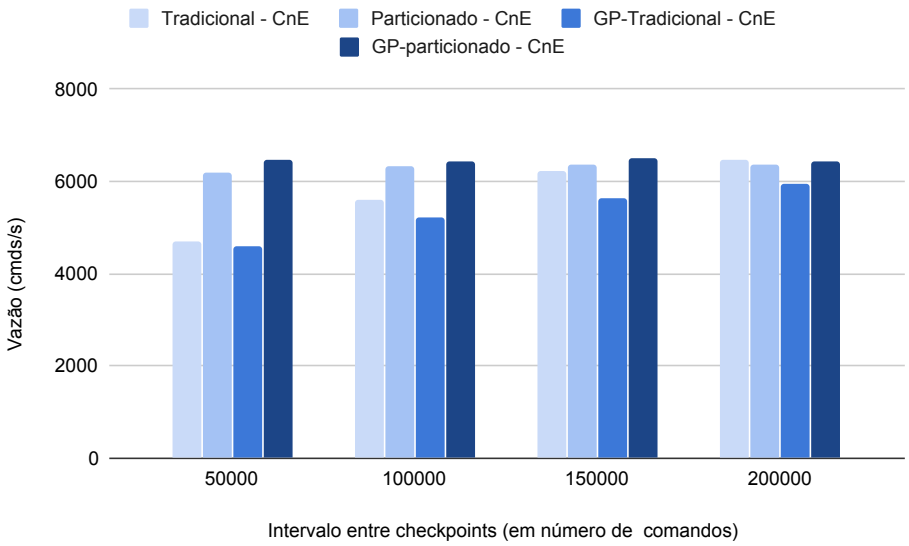


Figura 43 – Vazão média para diferentes intervalos de *checkpoint* utilizando CnE - 0r100c

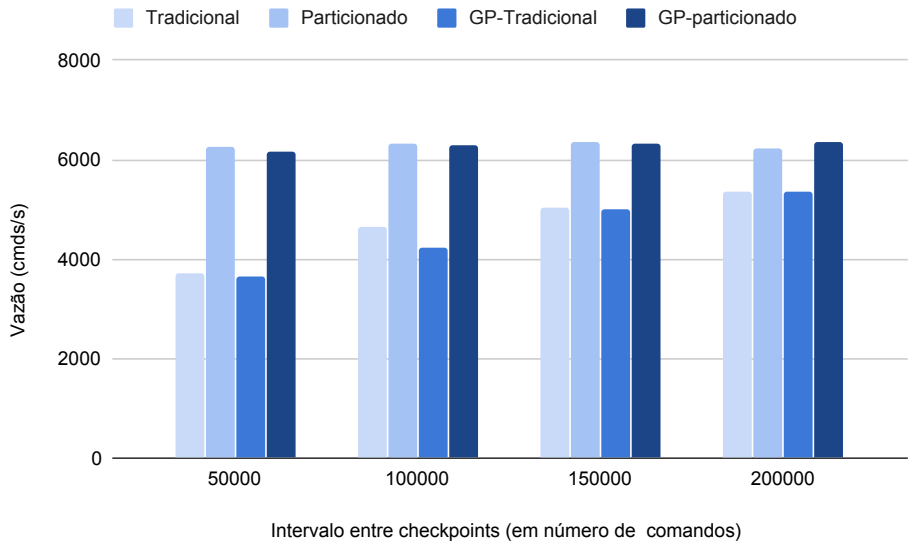


Figura 44 – Vazão média para diferentes intervalos de *checkpoint* sem utilizar CnE - 0r0c

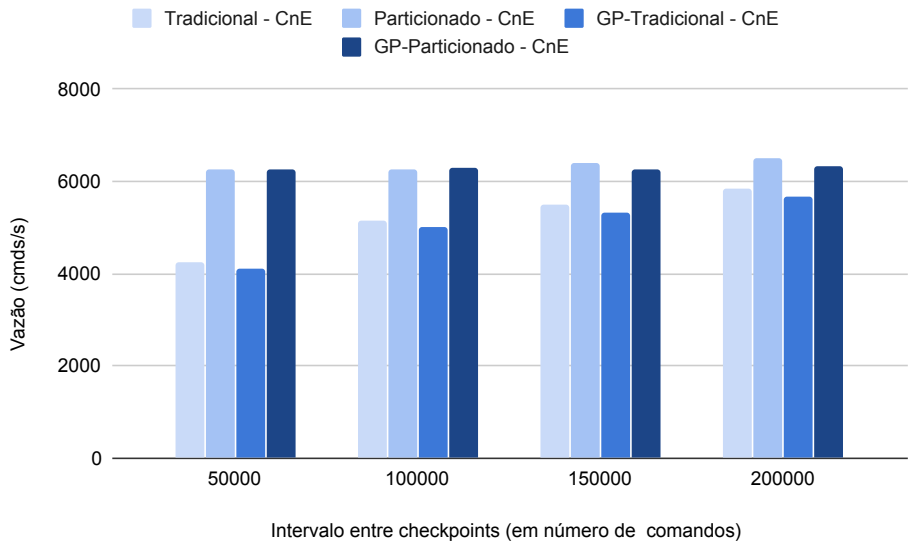


Figura 45 – Vazão média para diferentes intervalos de *checkpoint* utilizando CnE - 0r0c