PUCRS

ESCOLA POLITÉCNICA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO
MESTRADO EM CIÊNCIA DA COMPUTAÇÃO

KAYEL LUDTKE SERAFIM

BENCHMARK TPC-C APLICADO EM REPLICAÇÃO MÁQUINA DE ESTADOS

Porto Alegre
2023

PÓS-GRADUAÇÃO - *STRICTO SENSU*

Pontifícia Universidade Católica
do Rio Grande do Sul

PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO GRANDE DO SUL
ESCOLA POLITÉCNICA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

BENCHMARK TPC-C APLICADO EM REPLICAÇÃO MÁQUINA DE ESTADOS

KAYEL LUDTKE SERAFIM

Dissertação apresentada como requisito parcial à obtenção do grau de Mestre em Ciência da Computação na Pontifícia Universidade Católica do Rio Grande do Sul.

Orientador: Prof. Dr. Fernando Luís Dotti

Porto Alegre
2023

Ficha Catalográfica

S481b Serafim, Kayel Ludtke

Benchmark TPC-C aplicado em replicação máquina de estados / Kayel Ludtke Serafim. – 2023.
57.
Dissertação (Mestrado) – Programa de Pós-Graduação em Ciência da Computação, PUCRS.

Orientador: Prof. Dr. Fernando Luís Dotti.

1. Replicação Máquina de Estado. 2. Tolerância a Falhas. 3. Benchmark. I. Dotti, Fernando Luís. II. Título.

Elaborada pelo Sistema de Geração Automática de Ficha Catalográfica da PUCRS
com os dados fornecidos pelo(a) autor(a).
Bibliotecária responsável: Clarissa Jesinska Selbach CRB-10/2051

KAYEL LUDTKE SERAFIM

BENCHMARK TPC-C APLICADO EM REPLICAÇÃO MÁQUINA DE ESTADOS

Dissertação apresentada como requisito parcial para obtenção do grau de Mestre em Ciência da Computação do Programa de Pós-Graduação em Ciência da Computação, Escola Politécnica da Pontifícia Universidade Católica do Rio Grande do Sul.

Aprovado(a) em 20 de Março de 2023.

BANCA EXAMINADORA:

Prof. Dr. Avaliador Odorico Machado Mendizabal (PPGCC/UFSC)

Prof. Dr. Avaliador César Augusto Fonticielha De Rose (PPGCC/PUCRS)

Prof. Dr. Fernando Luís Dotti (PPGCC/PUCRS - Orientador)

DEDICATÓRIA

Dedico este trabalho a meus pais.

AGRADECIMENTOS

Primeiramente, gostaria de expressar minha gratidão a Deus por tornar possível a realização deste trabalho. Agradeço também à minha família pelo incentivo e apoio que me ofereceram durante todo o processo.

Gostaria de estender um agradecimento especial à minha namorada, Rafaela, por seu apoio incondicional nesta jornada. Ela sempre esteve ao meu lado, suportando meus desabafos e momentos de mau humor, com palavras de otimismo e compreensão.

Não posso deixar de agradecer ao meu professor orientador, Fernando Dotti, por sua orientação dedicada e incansável ao longo desse período.

Por fim, gostaria de expressar minha gratidão a todos que me ajudaram direta ou indiretamente na minha jornada.

BENCHMARK TPC-C APLICADO EM REPLICAÇÃO MÁQUINA DE ESTADOS

RESUMO

A disponibilidade de um sistema pode ser afetada por falhas ou ataques que exploram suas vulnerabilidades. Atualmente, cada vez mais pessoas confiam em sistemas *online* disponíveis na Internet. Para minimizar os riscos de indisponibilidade, a Replicação de Máquina de Estados (RME) é uma abordagem comum.

A RME é uma estratégia importante para fornecer serviços de alta disponibilidade. Porém, o aumento da vazão em sistemas RME é desafiador devido ao seu modelo determinístico, o que demanda esforços de pesquisa para melhorar sua vazão. Ainda assim, existe uma falta de cargas de trabalho que permitam avaliar diferentes mecanismos de RME de acordo com critérios comuns e representativos para classes de aplicações de interesse.

Com base nisso, importantes aspectos comuns foram identificados no contexto de transações *online*, e propôs-se o uso do TPC-C, o *benchmark* C do Comitê de Desempenho de Processamento de Transações, para avaliar RMEs. Sua arquitetura para o contexto RME foi discutida e implementada em uma plataforma de replicação. Resultados foram apresentados usando o modelo clássico de RME. Além disso, uma abordagem de RME paralelo foi discutida e implementada com esta carga de trabalho, e os resultados obtidos foram relatados.

Palavras-Chave: Replicação Máquina de Estado, Tolerância a Falhas, Benchmark.

TPC-C APPLIED BENCHMARK IN STATE MACHINE REPLICATION

ABSTRACT

The availability of a system can be impacted by failures or attacks that exploit its vulnerabilities. Increasingly, more people rely on available online systems on the Internet. To minimize downtime risks, State Machine Replication (SMR) is a common approach. SMR is an important strategy for providing highly available services. However, increasing capacity in SMR systems is challenging due to its deterministic model, which requires research efforts to improve its parallelism. Nevertheless, there is a lack of workloads to evaluate different SMR mechanisms according to common and representative criteria for classes of applications of interest. Based on this, important common aspects were identified in the context of online transactions, and the use of TPC-C, the Transaction Processing Performance Council's benchmark C, this work proposes to evaluate SMRs. Its architecture for the SMR context is discussed and an implementation on a replication platform is provided. TPC-C performance results are presented for the classic SMR model. Additionally, a parallel SMR approach is discussed, and implemented with this workload, and the results obtained reported.

Keywords: State Machine Replication, Fault Tolerance, Benchmark.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE SIGLAS

RME – Replicação Máquina de Estados

OLTP – *Online Transaction Processing*

HDFS – Hadoop *Distributed File System*

TPC – Transaction Processing Performance Council

TPC-C – *Transaction Processing Performance Council Benchmark C*

NAT – *Network Address Translation*

GFS – Google *File System*

SCFS – *Shared Cloud-backed File System*

SCADA – *Supervisory Control And Data Acquisition*

SDNS – *Secure Domain Name System*

DLM – *Distributed Lock Manager*

SDN – *Software defined networking*

DNS – *Domain Name System*

KVS – *Key-value store*

SUMÁRIO

1. INTRODUÇÃO

A Replicação da Máquina de Estado (RME) [30, 47] é uma abordagem simples e eficaz para criar serviços altamente disponíveis que oferecem linearizabilidade [24]. Na abordagem de RME, um conjunto de réplicas iniciam no mesmo estado e executam os mesmos comandos de clientes na mesma ordem, deterministicamente, mantendo o estado replicado e consistente.

A proposta seminal de RME [31] tem desempenho limitado, principalmente devido à redução da concorrência para garantir o determinismo. Com a crescente necessidade de serviços altamente disponíveis e fortemente consistentes, diversas abordagens foram e têm sido investigadas para escalar vazão em RME. Grande parte dos estudos voltados à escalabilidade em RME recaem sobre (i) o aumento de vazão das réplicas fazendo-se uso de paralelismo intra-réplica, por exemplo os mencionados na Seção 2.1.1; ou sobre (ii) o particionamento do estado da aplicação e provimento de diferentes conjuntos de réplicas para cada partição, aumentando a vazão para comandos em partições distintas, como por exemplo em [11, 34, 46, 46, 20, 51].

Majoritariamente, para a análise de desempenho, faz-se uso de aplicações sintéticas que permitem controlar parâmetros da carga de trabalho e, com isso, investigar seu efeito no desempenho conforme os mecanismos propostos.

No primeiro caso (i), um fator importante é a taxa de conflitos entre os comandos submetidos. Um conflito existe entre dois comandos se um deles escreve no conjunto de valores escritos ou lidos pelo outro, levando necessariamente à sequencialização da execução dos mesmos. Para o estudo de paralelismo intra-réplica utilizam-se aplicações com objetivo específico de geração de comandos com taxas controladas de conflitos. Neste contexto, a própria geração do conflito sintético tem técnicas diferentes. No segundo caso (ii), um fator importante é a taxa de comandos que acessam múltiplas partições, impondo sincronização entre partições e com isso afetando seu desempenho. Neste caso, tipicamente escolhe-se aplicações em que arbitrariamente define-se o percentual de operações em partições isoladas ou em múltiplas partições. Ainda que estas técnicas com cargas específicas permitam a avaliação dos mecanismos propostos, tipicamente a análise comparativa dos mesmos não é direta e a efetividade dos mesmos perante cargas de trabalho reais pode não estar clara.

Em síntese, considera-se que no desenvolvimento de soluções para RME existe uma lacuna de cargas de trabalho representativas de aplicações reais, disponíveis, que possam ser empregadas por diferentes proponentes para avaliar seus mecanismos propostos. Entende-se que uma carga de trabalho representativa para o estudo de desempenho em RME deva oferecer a possibilidade de estudo destes principais aspectos (i) e (ii) já mencionados. Neste sentido, são identificados aspectos comuns importantes no contexto

de processamento de transações *online*: o modelo de submissão e resposta a transações; o nível de consistência forte requerido; a noção análoga de conflito entre transações; e o uso de técnicas de particionamento. Assim, o translado de cargas de trabalho representativas advindas do contexto de transações *online* contribuem para o estudo de RME.

Neste contexto, o TPC-C é um *benchmark* C do Conselho de Desempenho de Processamento de Transação e tem como alvo os sistemas de processamento de transações *online* (OLTP), sendo aceito e utilizado para avaliação de diferentes plataformas e mecanismos para processamento de transações. Ele define a estrutura de uma base de dados e um conjunto de transações possíveis. Usuários submetem transações concorrentemente, esperando consistência forte. As taxas de cada tipo de transação e seus parâmetros também estão especificados no TPC-C. As transações acessam dados comuns, naturalmente originando conflitos. O modelo de dados preconiza a existência de armazéns (*warehouses*) e a carga especifica taxas de transações que envolvem mais de um armazém e quais são eles, fornecendo os dados para o particionamento por armazém.

Portanto, esta dissertação tem como objetivo geral propor o uso de cargas de trabalho do contexto de processamento de transações para aplicação em RME. Além disso, serão discutidas as implicações do mapeamento de uma aplicação transacional sobre RME. Os seguintes objetivos específicos serão abordados:

i. Propor uma arquitetura e implementação do TPC-C em uma plataforma de RME.

ii. Estender a proposta anterior para RME paralela por meio de uma definição de conflito entre transações, permitindo a execução simultânea de transações independentes.

iii. Realizar experimentos com o TPC-C no modelo clássico de RME e com extensões para RME paralela, implementados na biblioteca de replicação BFT-SMaRt.

O restante da dissertação está organizado da seguinte maneira: No Capítulo 2 é apresentada a fundamentação teórica; No Capítulo 3 é discutido o suporte às transações no modelo RME, assim como descreve como os principais elementos do TPC-C são mapeados para um serviço replicado e a definição da função de conflito para a integração do TPC-C em uma RME paralela; No Capítulo 4, são relatados os experimentos e os resultados alcançados. Finalmente, no Capítulo 5, são apresentadas as observações finais do trabalho.

2. FUNDAMENTAÇÃO TEÓRICA

2.1 Replicação de Máquina de Estado

A replicação é um meio essencial para alcançar alta disponibilidade, e a Replicação de Máquina de Estado (RME) [30, 47] é uma abordagem de replicação proeminente. Nesta abordagem, um serviço é definido como uma máquina de estado composta por variáveis de estado e uma função de transição de estado que é dada por um conjunto de comandos que alteram as variáveis de estado. Os comandos são enviados (entrada) pelos clientes e aplicados atomicamente, lendo e/ou escrevendo variáveis de estado e gerando uma resposta ao cliente. A execução do comando é determinística, ou seja, dado um estado de variáveis e um comando, há um próximo estado conhecido de variáveis e uma resposta conhecida para o cliente. É importante ressaltar que, por meio de um protocolo de difusão atômica [23], todas as réplicas devem iniciar no mesmo estado e executar os comandos na mesma sequência.

Com esses elementos, a RME fornece aos clientes a abstração de um serviço altamente disponível enquanto oculta a existência de múltiplas réplicas, garantindo linearizabilidade. Um sistema é linearizável se houver uma maneira de reordenar os comandos do cliente em uma sequência que (i) respeite a semântica dos comandos, conforme definido em suas especificações sequenciais, e (ii) respeite a ordem em tempo real dos comandos em todos os clientes [24, 4].

A abordagem RME é empregada em muitos sistemas importantes. Por exemplo, na pilha de software do Google com o Chubby [14], que é usado pelo gerenciador de cluster Borg, pelo Google File System (GFS) e pelo Bigtable [52, 17]. Analogamente, o Zookeeper do Apache [25] é usado pelo HDFS [48] e pelo Cassandra [29]. No entanto, a abordagem seminal RME tem desempenho limitado, pois as réplicas executam os comandos sequencialmente para garantir o determinismo e ordem total. A adição de réplicas não melhora a vazão, pelo contrário, pode prejudicar, pois mais réplicas precisam ser coordenadas durante o *broadcast* atômico ou consenso. Assim, diferentes abordagens para favorecer a vazão surgiram. De forma importante, é possível observar o paralelismo intra-réplica, mantendo o determinismo, e o uso de particionamento (*sharding*). A seguir, relata-se o uso destas técnicas.

2.1.1 RME Paralela

A abordagem paralela para RME emergiu da observação inicial de que, embora a execução de comandos concorrentes possa resultar em não determinismo, comandos

independentes podem ser executados concorrentemente sem violar a consistência [47]. Conforme já discutido, dois comandos conflitam se acessarem o estado compartilhado e pelo menos um deles alterar o estado compartilhado. Caso contrário, são independentes pois acessam diferentes partes do estado ou apenas leem as partes comuns acessadas.

A partir disso, várias abordagens foram propostas para permitir a execução concorrente de comandos [27, 28, 35, 36, 38, 3, 21, 13, 5]. Estas soluções relatam mecanismos intra-réplica para escalonar comandos para execução paralela, mantendo o determinismo entre as réplicas. Surgem diferentes meios para lidar com conflitos de comando, mostrando diferentes compensações entre sobrecarga de escalonamento e exploração de concorrência. De qualquer forma, em comum a todas estas técnicas, o aumento experimentado na vazão é altamente sensível ao grau de conflito da carga de trabalho empregada.

Em [21], uma classificação de protocolos para o escalonamento baseado em dependências é apresentada, com as seguintes abordagens:

- Protocolos de escalonamento tardio: a dependência entre requisições é identificada e tratada somente nas réplicas. Além do requisito acima mencionado em solicitações conflitantes, não há mais restrições no escalonamento. Assim sendo, os segmentos menos ocupados podem receber mais comandos se sua execução não entrar em conflito com comandos que estão sendo executados por outros segmentos.

  Nesta classe enquadram-se abordagens de RMEs paralelas para detecção de dependência baseadas na comparação de transações em pares [28] ou em lotes [38].

  O sistema CBASE [28], por exemplo, apresenta uma estrutura clássica de replicação paralela que adiciona um escalonador determinístico para as réplicas, em cada réplica é construído um grafo de dependência, localizando as dependências aos pares entre as transações em sua ordem total. Com base no grafo de dependência, o escalonador envia as transações para execução nas *threads* inativas presentes na *thread pool*. Após a execução o escalonador remove a transação do grafo e responde aos clientes.

  Além disso, pesquisas recentes [38] mostraram que comparações de transações aos pares introduzem contenção ao escalonamento e para superar esse problema, a batch CBASE [38] define as dependências por comparação em lotes. O escalonador lida e armazena lotes de comandos em vez de comandos únicos e as *threads* executam lotes inteiros. Dessa forma, comandos de um mesmo lote são executados sequencialmente na ordem estabelecida.

- Protocolos de escalonamento antecipado: parte das decisões de escalonamento é tomada antes que as requisições sejam demandadas, por exemplo através de uma classificação do tipo de requisição. Cada réplica deve respeitar restrições de escalonamento conforme a classe informada. A programação nas réplicas determina qual

thread no subconjunto definido executará a solicitação. Dessa forma, o conceito do escalonamento antecipado [2, 3] é fazer parte das decisões de escalonamento antes que os comandos cheguem ao paralelizador e, assim, reduzir a sobrecarga do escalonamento.

- Protocolos de escalonamento estático: é uma versão mais extrema do escalonamento antecipado, na qual é decidido quando *worker thread* irá executar um comando e quando esse comando deve ser executado via *broadcast*. O P-SMR [35] é um exemplo de programação estática, ele não possui paralelizador ou escalonador. As *workers threads* das réplicas fornecem e executam simultaneamente vários fluxos distintos de comandos ordenados. Para preservar a exatidão, os comandos em cada fluxo devem ser independentes. Para garantir a independência entre os fluxos entregues simultaneamente, ao contrário das abordagens anteriores onde as dependências de comando são determinadas nas réplicas, nesse caso as dependências de comando são determinadas pelos clientes, antes que os comandos sejam solicitados. O P-SMR implementa um modelo totalmente paralelo no qual os comandos independentes são ordenados, entregues e executados simultaneamente. Os comandos dependentes são ordenados por meio de grupos *multicast* dedicados e executados sequencialmente.

2.1.2 RME Particionada

O objetivo dessa abordagem é particionar o estado do serviço e replicar cada partição. O particionamento deve, tanto quanto possível, separar objetos com acessos independentes. Ou seja, que podem ser feitos concorrentemente sem impactar na consistência do serviço. Os comandos que acessam o estado de uma única partição são manipulados como na replicação clássica da máquina de estado (RME) [37]. Quando partições isoladas são acessadas, estes comandos podem ser executados paralelamente. Dessa maneira escalabilidade e linearização são garantidas. Se um comando acessa mais de uma partição, então sincronização pode ser necessária em relação a comandos acessando mesmas partições.

Existem algumas abordagens para lidar com comandos que acessam estados de diferentes partições, entretanto comandos multiparticionados, inevitavelmente, possuem desempenho limitado. Isso acontece devido à sobrecarga de ordenar e coordenar comandos entre partições para garantir uma consistência forte. Além disso, se os dados não forem distribuídos com cuidado, os desequilíbrios de carga poderão suprimir as vantagens do particionamento [37].

Alguns protocolos assumem que os serviços possuem particionamento de estado perfeito, ou seja, nenhum comando acessa os estados de duas ou mais partições. Dessa

forma, os comandos pertencem essencialmente a uma única partição e o desempenho será escalado conforme o número de partições, desde que a carga esteja equilibrada entre as mesmas [33], [40]. Há também protocolos que assumem que a carga de trabalho possa ser dividida e que os comandos sejam executados individualmente apenas nas partições envolvidas, sem a necessidade de armazenar dados em partições diferentes [39].

2.2 BFT-SMaRt

O BFT-SMaRt [9] é uma biblioteca Java para RME, que implementa um protocolo inspirado em PBFT (*Practical Byzantine Fault Tolerance)* [16] para tolerar falhas bizantinas (BFT). No entanto, o sistema também pode ser configurado para tolerar apenas faltas por parada/*crash* (CFT). Além disso, suporta a reconfiguração do conjunto de réplicas e provê suporte para durabilidade.

A vazão do BFT-SMaRt depende do tamanho da mensagem e do modo de tolerância a falhas (CFT ou BFT) [9]. No modo BFT, mais e maiores mensagens são trocadas devido aos mecanismos necessários. Além dos aspectos de comunicação, a execução também afeta o desempenho. O BFT-SMaRt[1] implementa o modelo sequencial de execução nas réplicas. Porém, no topo do BFT-SMaRt foram implementadas extensões para execução paralela[2] e particionada. Essas extensões suportam diferentes abordagens de execução discutidas em [42].

Atualmente, o BFT-SMaRt oferece uma ampla variedade de protocolos, incluindo os relacionados à reconfiguração, checkpoints e transferência de estados. Dessa forma, ele se torna uma biblioteca completa para a gestão de estados, chamada de Recuperação de Mecanismos de Estado (RME). Essa biblioteca foi criada seguindo os seguintes princípios de *design*:

Modelo de falhas ajustável: O BFT-SMaRt é capaz de tolerar falhas bizantinas não maliciosas e oferece assinaturas criptográficas para tolerar falhas maliciosas. Ele também possui um protocolo RME simplificado para tolerar apenas falhas e corrupções de mensagens [9].

Simplicidade: Para garantir a correção do protocolo, evita-se otimizações que introduzam complexidade extra na implantação, codificação ou novos casos específicos. Isso inclui evitar técnicas promissoras em desempenho ou eficiência de recursos que dificultem a renderização correta ou a implantação do sistema [9].

Modularidade: BFT-SMaRt possui um protocolo RME modular com uma primitiva de consenso bem definida, enquanto outros sistemas têm um protocolo monolítico onde o

[1] https://github.com/bft-smart/library
[2] https://github.com/parallel-SMR/library

algoritmo de consenso é integrado a RME sem separação clara. As alternativas modulares são geralmente mais fáceis de implementar e compreender [9].

Interface de Programação de Aplicativos Simples e Extensível: A API simples oferecida permite que programadores implementem serviços determinísticos seguindo o modelo de programação RME. Os clientes podem usar o método *invoke*(*command*) para enviar comandos para réplicas que os processam com o método *execute*(*command*). Se a aplicação precisar de comportamentos especializados, eles podem ser implementados com chamadas alternativas, *callbacks* ou *plug-ins* no lado do cliente ou do servidor [9].

Reconhecimento de vários núcleos: O BFT-SMaRt utiliza a arquitetura *multicore* presente em servidores para melhorar o desempenho de algumas tarefas de processamento dispendiosas no caminho crítico do protocolo. A vazão do sistema é dimensionado pelo número de *threads* de hardware suportados pelas réplicas, especialmente quando as assinaturas são habilitadas e há necessidade de maior poder computacional para sua verificação [9].

O BFT-SMaRt é um sistema que tolera f falhas bizantinas em $n \geq 3f + 1$ réplicas e permite alterar n e f em tempo de execução. Além disso, o sistema também pode ser configurado para usar apenas $n \geq 2f + 1$ réplicas para tolerar f falhas por *crash*. Clientes propensos a falhas e eventual sincronia garantem a vivacidade do sistema. *Links* confiáveis são usados para comunicação entre processos e são implementados com códigos de autenticação de mensagem (MACs) em TCP/IP. Chaves de réplica-réplica são geradas por meio de *Signed Diffie-Helman* com um par de chaves RSA por réplica, enquanto chaves de cliente-réplica são geradas com base nos *ids* dos *endpoints*. Autenticação forte de clientes está disponível se solicitações assinadas estiverem habilitadas [9].

2.2.1 Arquitetura BFT-SMaRT

A arquitetura de uma réplica de um sistema de processamento de requisições, pode ser dividida em três partes: recebimento, ordenamento e execução de requisições.

No recebimento, os clientes enviam suas requisições ao sistema, que são replicadas e enviadas para as réplicas responsáveis por atendê-las. Cada cliente possui uma *threads* dedicada e suas requisições são colocadas em filas separadas. A autenticidade das requisições é garantida por meio de assinaturas digitais [9].

No ordenamento, uma *thread proposer* inicia uma instância do consenso para definir a ordem de entrega de um lote de requisições. As réplicas se comunicam entre si por meio de outro conjunto de *threads* e existe um conjunto de *threads* responsável por receber as mensagens enviadas pelas outras réplicas. O processamento necessário para garantir a autenticidade dessas mensagens é realizado nestas *threads*. No BFT-SMaRt, o ordenamento é sequencial, ou seja, uma nova instância do consenso só é iniciada após

a instância anterior ter terminado. O objetivo de ordenar as requisições em lotes é aumentar o desempenho do sistema. Caso uma requisição não seja ordenada dentro de um determinado tempo, o sistema troca a réplica líder. Para evitar que uma réplica maliciosa force a troca de líder, é necessário que pelo menos $f + 1$ réplicas solicitem a troca [9].

Na execução de requisições, as réplicas processam as requisições ordenadas e retornam os resultados para os clientes. Os clientes verificam a validade da resposta comparando-a com $f + 1$ respostas iguais (ou apenas 1 resposta caso o sistema esteja configurado para tolerar apenas falhas por *crash*). Atualmente, foi proposto um protocolo para paralelizar a execução de múltiplas requisições em paralelo, o que aumenta ainda mais o desempenho do sistema [9].

2.3 TPC Benchmark C

2.3.1 Visão Geral

O *Transaction Processing Performance Council*[3] (TPC) reúne diferentes atores interessados no desempenho de sistemas transacionais e, portanto, como uma de suas atividades, propõe *benchmarks* para medir tais sistemas. TPC-C[4] especifica o *benchmark* C deste comitê, tendo como alvo sistemas complexos de processamento de transações *online* (OLTP) e simula um ambiente OLTP complexo através de um conjunto de operações básicas de leitura e escrita. Ele foi projetado para fornecer dados de desempenho relevantes para a comparação objetiva de dois ou mais sistemas em termos de sua capacidade de processar transações. O TPC-C retrata a atividade de uma empresa atacadista com vários distritos de vendas e armazéns associados, sendo que o tamanho da organização é determinado pelo número de armazéns. Cada armazém mantém estoque de 100 mil produtos vendidos pela empresa e cada distrito atende a 3 mil clientes.

O propósito de um *benchmark* é reduzir a diversidade de operações encontradas em uma aplicação de produção, mantendo as características de desempenho essenciais da aplicação, como o nível de utilização do sistema e a complexidade das operações. Muitas funções precisam ser realizadas para gerenciar um sistema de entrada de pedidos de produção, mas muitas delas não são de interesse principal para análise de desempenho e são omitidas no TPC-C para evitar excesso de diversidade. A empresa retratada pelo TPC-C é um fornecedor atacadista que expande seu negócio criando novos armazéns e distritos de vendas, e mantém estoques dos 100 mil itens vendidos em todos os seus armazéns.

[3] https://www.tpc.org/

[4] https://www.tpc.org/tpc_documents_current_versions/current_specifications5.asp

O diagrama a seguir ilustra a hierarquia do depósito, distrito e cliente do ambiente de negócios da TPC-C.

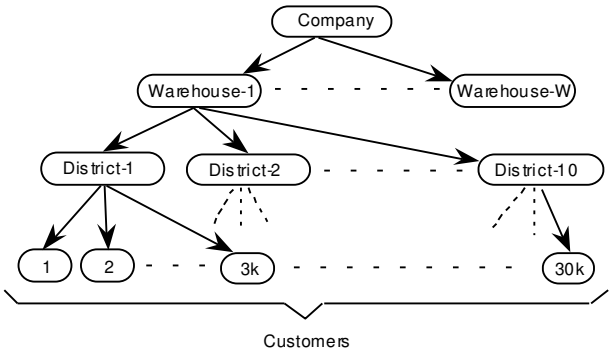


Figura 2.1: Hierarquia do ambiente de negócios do TPC-C [19].

Clientes entram em contato com a empresa para realizar novos pedidos ou solicitar o *status* de pedidos existentes. Em média, os pedidos são compostos por 10 itens, mas há casos em que um por cento deles incluem produtos que não estão disponíveis no armazém regional e precisam ser adquiridos em outro armazém. Além disso, o sistema da empresa também é utilizado para registrar pagamentos dos clientes, organizar a entrega de pedidos e monitorar os níveis de estoque para identificar possíveis faltas de suprimentos.

2.3.2 Modelagem de Dados

A modelagem do banco de dados do TPC-C é composta por nove tabelas. Os detalhes do relacionamento entre as tabelas podem ser vistos na Figura 2.2 abaixo:

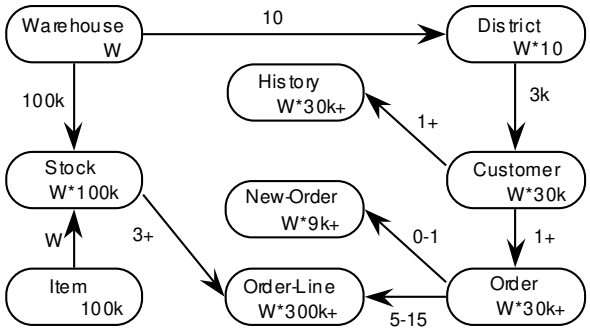


Figura 2.2: Diagrama Entidade-Relacionamento (DER) [19].

- Todos os números apresentados ilustram os requisitos populacionais do banco de dados;

- Os números dentro dos blocos da entidade representam a cardinalidade das tabelas (número de linhas). Se a tabela *warehouse*, por exemplo, possuir dez registros, então a tabela *district* terá cem registros e a tabela *customer*, por sua vez, terá trezentos mil registros. A tabela *item* é a única exclusividade nessa modelagem, pois sua cardinalidade é independente do número de registros de qualquer outra;
- Os números juntos às flechas representam as cardinalidades dos relacionamentos (número de filhos por pai);
- O símbolo de mais (+) indica que mais linhas serão adicionadas à tabela à medida que o *benchmark* for executado.

2.3.3 Transações

TPC-C envolve a mistura de cinco tipos de transações simultâneas de complexidades variadas que são executadas *online* ou em fila assíncrona. Cada transação possui características diferentes que afetam o desempenho do sistema de várias formas diferentes. Abaixo, uma breve descrição das transações:

New-Order: É responsável por registrar uma operação de compra realizada por um cliente. Ela inclui a inserção de dados nas tabelas Order, New_Order e Order_Line, bem como a atualização do estoque (na tabela Stock) para cada item adquirido. Além disso, ela realiza consultas nas tabelas Warehouse, District, Customer e Item para obter informações necessárias. É uma transação de alta frequência, que impõe uma carga média ao sistema devido às operações de leitura e escrita. Ela é a transação principal do benchmark TPC-C.

Payment: É responsável por registrar os pagamentos realizados por um cliente. Ela inclui a atualização das informações de saldo do cliente (na tabela Customer), do distrito (na tabela District) e do armazém (na tabela Warehouse). Além disso, ela insere um novo registro na tabela History. É uma transação de baixo impacto, com alta frequência de execução, composta por operações de leitura e escrita.

Delivery: É responsável por registrar a entrega de pedidos. Ela processa em lote dez novos pedidos e, para cada distrito de um armazém, realiza as seguintes operações: busca e exclusão do registro mais antigo da tabela New Order associado ao distrito em questão, e atualização de registros nas tabelas Order, Order_Line e Customer. É uma transação que envolve operações de leitura e escrita, com baixa frequência de execução, e que impõe uma carga moderada ao sistema.

Order-Status: É responsável por consultar o estado da última compra de um cliente. Ela realiza as seguintes operações: seleção de informações do cliente a partir da

tabela Customer e seleção de informações de compra a partir das tabelas Order e Order_Line. É uma transação somente-leitura, executada com baixa frequência, que impõe uma carga moderada ao sistema.

Stock-Level: É utilizada para determinar quais itens recentemente vendidos estão com estoque abaixo de um determinado limite. Ela consiste em selecionar as últimas vinte compras de um distrito a partir da tabela District e contar os itens cujo estoque está abaixo do limite estabelecido a partir das tabelas Stock e Order_Line. Essa transação é somente-leitura, o que significa que não há modificações no banco de dados, e é executada com baixa frequência, mas que impõe uma carga pesada ao sistema.

Por fim, vale destacar que o TPC-C especifica a seguinte distribuição de transações na carga de trabalho: New-Order 45%; Payment 43%; Delivery 4%; Order-Status 4%; e Stock-Level 4%.

2.4 Aplicações de RME

Para que seja possível propor aplicações e cargas de trabalho de referência para RME, é necessário entender as utilizações desse tipo de replicação de máquina de estado. Com isso, torna-se possível propor aplicações representativas de referência.

Nesta seção, é apresentado um levantamento de aplicações de RME encontradas por meio de uma busca na literatura. Embora esse levantamento não possa ser considerado exaustivo, e pretenda-se continuar com ele, já representa um levantamento razoável, pois permite inclusive derivar uma classificação das aplicações encontradas.

Conforme a Tabela 2.1, é apresentada a classificação das aplicações de RME encontradas.

Tabela 2.1: Classificação de aplicações RME.

Classificação	Aplicação	Descrição	Avaliação de desempenho
Sistema de armazenamento de arquivos	SCFS [8]	É um sistema de arquivos baseado na nuvem que ultrapassa as restrições dos serviços convencionais de armazenamento na nuvem. Seu objetivo principal é proporcionar aos clientes a capacidade de compartilhar arquivos de maneira segura e resiliente a falhas.	O SCFS foi avaliado usando parâmetros do Filebench em *back-ends* da AWS e Cloud-of-Clouds. A avaliação mostrou que o desempenho foi semelhante em todos os sistemas, exceto o S3FS e o S3QL, que apresentaram problemas de desempenho devido a falta de cache de memória principal e gravação lenta, respectivamente. As gravações foram realizadas com tamanhos de 4kB para o S3FS e 128kB para o S3QL.
Sistema de armazenamento de arquivos	DepSky [7]	É um sistema de armazenamento seguro que replica dados em várias nuvens comerciais para melhorar a disponibilidade e confidencialidade. Ele cria uma nuvem dentro de outras nuvens, permitindo que os usuários gerenciem dados e realizem operações em diferentes nuvens individuais.	DepSky usou o PlanetLab para avaliar o desempenho de provedores populares de nuvem de armazenamento. A latência foi medida por um aplicativo de *logger* que acessava seis provedores. O *logger* foi instalado em oito máquinas PlanetLab em quatro continentes, com três instâncias para diferentes tamanhos. O experimento durou dois meses.
Sistema de armazenamento transacional	Google Spanner [18]	É um banco de dados distribuído e globalmente escalável, que é replicado de forma síncrona. O Spanner é capaz de realizar transações distribuídas externamente consistentes, que podem abranger vários fragmentos de dados, e utiliza o Paxos para replicação vertical entre zonas geográficas e o TwoPhase Commit para transações atômicas.	As medições de desempenho foram realizadas em máquinas com *timeshared* e um banco de dados de teste criado com 50 grupos de Paxos. A latência permaneceu constante com o aumento de réplicas, enquanto a taxa de transferência aumentou linearmente para leituras de *Snapshot*. Para transações de gravação, a quantidade de trabalho aumentou linearmente. A avaliação de escalabilidade em duas fases foi razoável com 50 participantes, mas as latências aumentaram com 100 participantes.
Distributed lock manager	Apache ZooKeeper [25]	É um serviço de coordenação tolerante a falhas para aplicações distribuídas. Ele usa uma estrutura hierárquica de dados e uma arquitetura de pipeline sem bloqueio para fornecer coordenação e sincronização em clusters de servidores.	A avaliação de desempenho revelou que a taxa de leitura aumentou com mais servidores, enquanto a taxa de gravação diminuiu devido ao uso do *atomic broadcast*. Isso indica que o número de servidores impacta negativamente o protocolo de transmissão e que os servidores precisam garantir a persistência das transações antes de enviar confirmações de volta ao líder.
Distributed lock manager	Google Chubby [14]	É um serviço de diretório altamente disponível e tolerante a falhas, usado para armazenar informações de configuração, bloqueios distribuídos e outros dados de pequeno tamanho. Utiliza replicação ativa e eleição de líder para garantir a consistência dos dados e é amplamente utilizado pelo Google para suportar sistemas distribuídos e aplicativos de larga escala.	O artigo não apresentou métricas de avaliação de desempenho.
Serviços de rede	SMaRtLight [12]	É uma arquitetura que resolve problemas de tolerância a falhas em redes de computadores, utilizando um banco de dados compartilhado e replicado para armazenar as informações de controle do controlador primário e permitir a sincronização com o controlador de *backup*.	O CBench foi utilizado para avaliar o desempenho do SMaRtLight em redes SDN, com experimentos de dez segundos com diferentes números de *switches* e *hosts*. Para atingir o desempenho máximo, o CBench foi executado na mesma máquina que o controlador primário. Mesmo sendo uma versão preliminar, o SMaRtLight apresentou resultados promissores, com uma taxa de processamento entre 100 mil e 350 mil fluxos por segundo e taxas de *cache hits* acima de 50%.
Serviços DNS	SDNS [53]	É um esquema DNS seguro que utiliza técnicas tolerantes a intrusões bizantinas e mecanismos de votação para fornecer alta integridade, robustez e disponibilidade do serviço, mesmo na presença de falhas arbitrárias, incluindo ataques maliciosos.	Os autores não disponibilizaram avaliação de desempenho.
Sistemas SCADA	SMaRt-SCADA [41]	É um protótipo que visa integrar o Eclipse NeoSCADA e o BFT-SMaRt para criar um sistema tolerante a intrusões. O principal objetivo é assegurar a determinismo e a coordenação em um sistema que originalmente não foi projetado para ser determinístico.	O NeoSCADA e o SMaRt-SCADA foram testados em dois cenários de alarme, com 100% e 50% de ativação de alarme em cada mensagem ItemUpdate. O SMaRt-SCADA apresentou sobrecarga de 10% e 25% nos cenários de 50% e 100% de alarmes, respectivamente, refletindo a introdução de etapas adicionais de comunicação. No cenário de 100% de alarme, o número de eventos que vão para o armazenamento foi o dobro do observado no cenário de 50% de alarme.

2.4.1 Sistemas de armazenamento

Sistemas de arquivos

Shared Cloud-backed File System (2014)

O Shared Cloud-backed File System (SCFS) [8] é um sistema de arquivos com suporte na nuvem que soluciona as limitações de serviços de armazenamento e suporte em nuvem. O principal objetivo é permitir que os clientes compartilhem arquivos de maneira segura e tolerante a falhas. Além disso, o sistema melhora a durabilidade dos dados através dos recursos oferecidos pelos serviços em nuvens em conjunto com o controle de versões de arquivos.

O SCFS também visa oferecer uma API ao sistema de arquivos com consistência forte, devido ao fato de que a maioria das nuvens de armazenamento fornece apenas consistência eventual. Para garantir essa consistência o sistema faz uso de serviços de coordenação [10, 25] que preservam os metadados e a sincronização. O SCFS também aproveita a escalabilidade dos serviços em nuvem para suportar um grande volume de dados e usuários.

A Figura 2.3 apresenta os três componentes principais da arquitetura do SCFS: o armazenamento em nuvem que mantém os dados do arquivo; o serviço de coordenação para gestão dos metadados e suporte à sincronização; e o agente SCFS que executa a maior parte da funcionalidade do SCFS e corresponde ao cliente do sistema instalado na máquina do usuário.

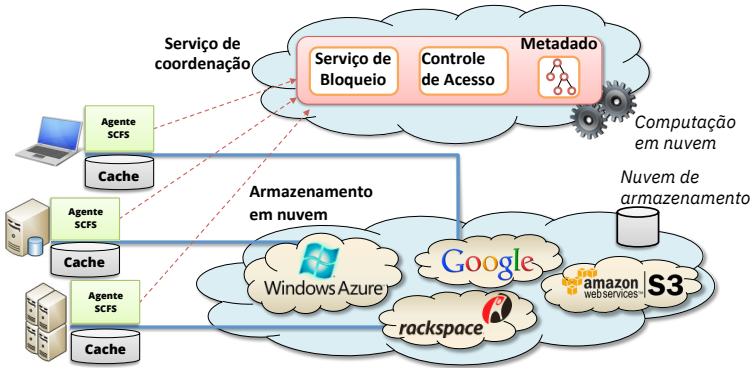


Figura 2.3: Arquitetura SCFS [8].

O desempenho é avaliado utlizando os *back-ends* da AWS e da tecnologia Cloud-of-Clouds (CoC) [1, 15, 7], operando em modos diferentes e comparando estes sistemas com outros de arquivos com *backup* em nuvem, conforme Figura 2.4.

	Blocking	*Non-blocking*	*Non-sharing*
AWS	SCFS-AWS-B	SCFS-AWS-NB	SCFS-AWS-NS
CoC	SCFS-CoC-B	SCFS-CoC-NB	SCFS-CoC-NS

Figura 2.4: Variantes do SCFS com diferentes modos e back-end [8].

A avaliação é baseada em um conjunto de parâmetros de referência [49], todos do Filebench [50]. Além disso, outros dois novos *benchmarks* foram criados: *File Synchronization Service* e *Sharing Files*. Esses *benchmarks* foram criados para simular comportamentos de interesse com dois sistemas populares de arquivos de código aberto suportados pelo S3: S3QL [43], S3FS [45] e um sistema de arquivos local baseado em FUSE-J (LocalFS).

A avaliação inicia com seis *micro-benchmarks* do Filebench: leituras sequenciais, gravações sequenciais, leituras aleatórias, gravações aleatórias, criação e cópia de arquivos. Os quatro primeiros benchmarks fazem uso intensivo de leitura e escrita e não consideram operações de abertura, sincronização ou fechamento, enquanto os dois últimos são intensivos em metadados.

Os resultados avaliados mostraram que o comportamento dos sistemas são semelhantes em relação as leituras e gravações sequenciais e aleatórias, com exceção do S3FS e S3QL. S3FS apresentou baixo desempenho devido a falta de cache da memória principal em arquivos abertos [45], enquanto o S3QL apresenta baixo desempenho em gravações aleatórias decorrente de um problema conhecido do FUSE que torna as gravações de fragmentos pequenos muito lento[44]. Os *benchmarks* executaram gravações em 4kB para S3FS e 128kB para S3QL.

Para os autores, apesar dos custos de consistência forte, o *design* é prático e oferece controle de um conjunto de vantagens e desvantagens relacionadas à segurança, consistência e custo e eficiência.

DepSky (2013)

O DepSky [7] é um sistema de armazenamento confiável e seguro que replica os dados em diversas nuvens comerciais para criar uma nuvem dentro de outras nuvens, esse sistema melhora a disponibilidade e a confidencialidade fornecidas por esses serviços comerciais. Em outras palavras, é um sistema de armazenamento virtual, que permite que seus usuários possam gerenciar os dados e realizar operações nas diferentes nuvens individuais. A Figura 2.5 ilustra a arquitetura do DepSky, em que dois clientes se comunicam com quatro servidores comerciais em nuvem diferentes.

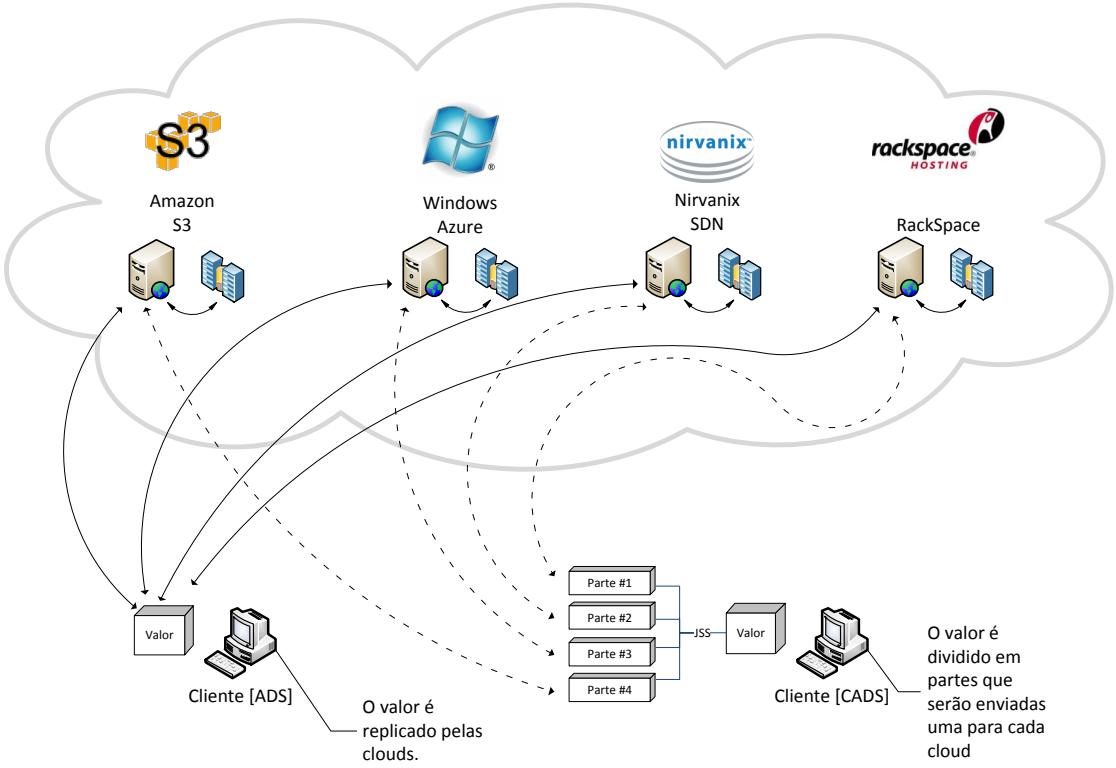


Figura 2.5: Arquitetura do DepSky (4 nuvens e 2 clientes) [7].

DepSky utiliza o PlanetLab para avaliar o desempenho de uma implantação real e simular o acesso de vários clientes ao sistema, composto por provedores populares de nuvem de armazenamento. As medições de latência foram obtidas utilizando um aplicativo de *logger* que tenta ler uma unidade de dados de seis nuvens diferentes.

Os pesquisadores implementaram o registrador em oito máquinas PlanetLab na Internet, em quatro continentes. Em cada uma dessas máquinas, três instâncias do registrador foram iniciadas para diferentes tamanhos: 100KB (uma medição a cada 5 minutos), 1MB (uma medição a cada 10 minutos) e 10MB (uma medição a cada 30 minutos). Esses experimentos ocorreram durante dois meses.

Segundo os autores, o sistema alcança os objetivos para os quais é proposto, construindo uma nuvem sobre um conjunto de nuvens de armazenamento, combinando protocolos de sistema de quorum bizantino, compartilhamento de segredo criptográfico, códigos de apagamento e a diversidade fornecida pelo uso de vários provedores de nuvem.

Armazenamento transacional

Google Spanner (2013)

O Spanner [18] é um banco de dados escalável, multi-versão, distribuído globalmente e replicado de forma síncrona da Google. É o primeiro sistema a distribuir da-

dos em escala global e dar suporte a transações distribuídas externamente consistentes. Além disso, é um banco de dados que fragmenta dados em muitos conjuntos de máquinas de estado, combina Paxos [31] para replicação vertical entre zonas geográficas com um *TwoPhase Commit* [6] para transações atômicas que abrangem fragmentos. O Spanner compartilha automaticamente os dados entre as máquinas à medida que a quantidade de dados ou o número de servidores muda e migra automaticamente os dados entre as máquinas (mesmo entre os datacenters) para equilibrar a carga e em resposta a falhas.

As medições de desempenho foram feitas em máquinas com *timeshared*, onde cada servidor de spans rodava em unidades de agendamento de 4GB de RAM e 4 núcleos. Os clientes foram executados em máquinas separadas. O banco de dados de teste foi criado com 50 grupos de Paxos com 2500 diretórios (conjunto de réplicas é coletivamente um grupo Paxos). As operações eram leituras e gravações independentes de 4kB.

Para os experimentos de latência, os clientes emitiram poucas operações para evitar filas nos servidores. Nos experimentos com uma réplica, a espera de confirmação (*commit wait*) foi de cerca de 5ms e a latência do Paxos foi de 9ms. Conforme o número de réplicas aumentava, a latência permanecia aproximadamente constante com menos desvio padrão.

Para os experimentos de vazão, os clientes emitiram operações suficientes para saturar as CPUs dos servidores. As leituras de *Snapshot* foram executadas em qualquer réplica atualizada, portanto, sua taxa de transferência aumentou quase que linearmente com o número de réplicas. As transações únicas de somente leitura são executadas apenas nos líderes e apresentou um aumento da taxa de transferência com o aumento do número de servidores spans efetivos. A taxa de transferência de gravação se beneficia do mesmo artefato experimental, mas esse benefício é superado pelo aumento linear na quantidade de trabalho realizado por gravação, à medida que o número de réplicas aumenta.

A avaliação de escalabilidade do *commit* em duas fases foi realizada em um conjunto de experimentos executados em 3 zonas, cada uma com 25 servidores spans e ao dimensionar até 50 participantes mostrou-se razoável, mas as latências começaram a aumentar visivelmente em 100 participantes.

2.4.2 Distributed lock manager (DLM)

Apache ZooKeeper (2010)

O ZooKeeper [25] é um serviço de coordenação tolerante a falhas que fornece alto desempenho. Esse serviço é inspirado no Chubby da Google, que é um serviço de bloqueio com fortes garantias de sincronização, no entanto o ZooKeeper não utiliza blo-

queios ou quaisquer outras construções de bloqueio para obter coordenação, porque essas primitivas de bloqueio podem reduzir significativamente o desempenho. Em vez disso, utiliza uma arquitetura de pipeline sem bloqueio com uma estrutura hierárquica de dados na qual clientes ou usuários do serviço ZooKeeper podem ler e gravar dados.

Para oferecer garantias de coordenação para essa estrutura de dados, o Zookepeer utiliza duas abordagens: ordenação dos clientes no modelo de filas FIFO (first in - first ou) e operações de gravação linearizáveis. O modelo FIFO impõe a ordem na qual um cliente recebe as respostas, essa garantia estabelece que todas as solicitações de um cliente são executadas na mesma ordem em que um cliente emitiu as solicitações. A linearizabilidade das operações de gravação significa que todas as alterações de estado são serializáveis e respeitam a precedência, ou seja, todas as operações de gravação são replicadas para todos os servidores ZooKeeper na mesma ordem em que foram recebidas do cliente.

A estrutura de dados hierárquica do ZooKeeper é semelhante ao sistema de arquivos, pois cada objeto de dados, chamado *znode*, pode ser acessado para leitura ou gravação usando um nome de caminho hierárquico. A figura 2.6 abaixo ilustra o conceito de sistema de arquivos como modelo de dados hierárquico.

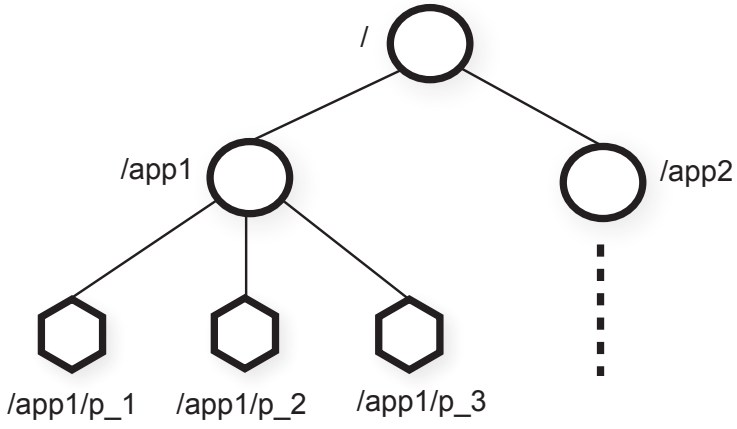


Figura 2.6: Ilustração do modelo hierárquico do ZooKeeper [25].

A Figura 2.7 apresenta a taxa de transferência à medida que modifica-se o percentual de solicitações de leitura para gravação. Cada curva corresponde a uma quantidade diferente de servidores disponibilizados para o serviço ZooKeeper.

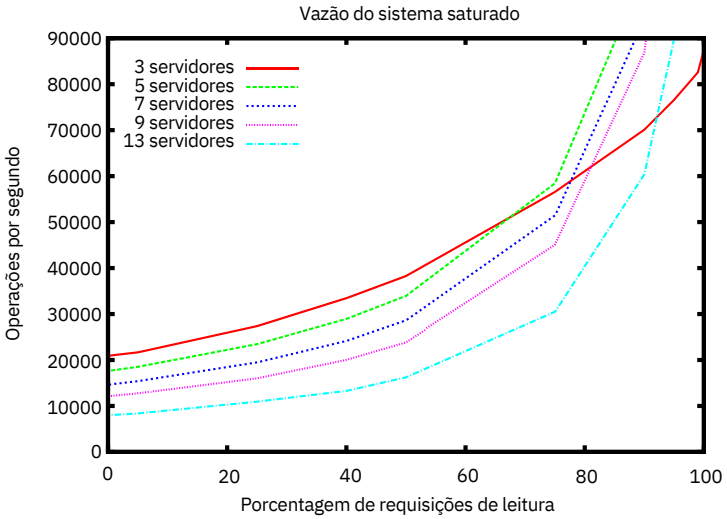


Figura 2.7: Avaliação da taxa de transferência [25].

As taxas de leitura apresentam números mais elevados à medida que novos servidores são adicionados ao serviço, fazendo com que a taxa de transferência de leitura melhore. Porém, a taxa de transferência de gravação diminui, porque as solicitações de gravação utilizam *atomic broadcast*, o que requer um processamento extra e adiciona latência às solicitações, demonstrando assim que o número de servidores tem um impacto negativo no desempenho do protocolo de transmissão. Além disso, os servidores precisam garantir que as transações sejam registradas em armazenamento não volátil antes de enviar as confirmações de volta ao líder.

Google Chubby (2006)

O Chubby [14] é um serviço de bloqueio centralizado com a finalidade de sincronizar as atividades dos clientes em sistemas distribuídos de baixo acoplamento. Os objetivos principais incluem confiabilidade, disponibilidade para um conjunto relevante de clientes e semântica de fácil compreensão. A ênfase desse sistema está no design confiável e disponível, em oposição ao alto desempenho. A taxa de transferência e armazenamento foram considerados secundários.

O Chubby possui dois componentes principais que se comunicam via RPC (*Remote Procedure Call*): um servidor mestre e uma biblioteca utilizada pelos clientes como conexão. Nesse cenário, cada aplicação interessada em obter sincronização no ambiente distribuído conecta-se ao Chubby através dessa biblioteca e essa, por sua vez, executa o protocolo de bloqueio em nome da aplicação. A Figura 2.8 abaixo apresenta a estrutura dos componentes.

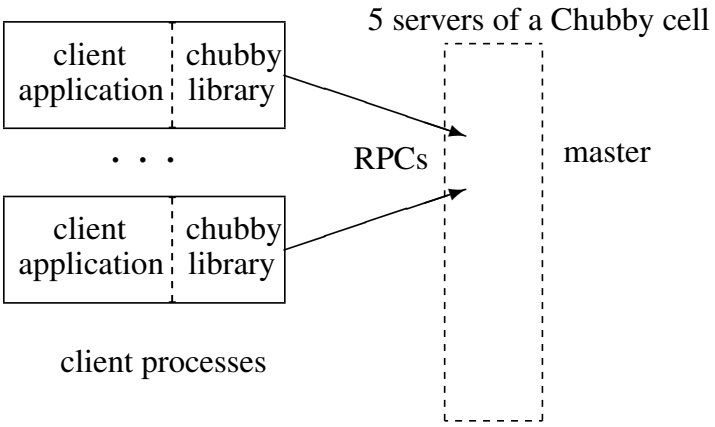


Figura 2.8: Estrutura do sistema [14].

As réplicas utilizam o protocolo Paxos [31] de consenso distribuído para eleger um mestre; o mestre deve obter votos da maioria das réplicas, além de garantir que essas réplicas não elegerão um mestre diferente por um determinado intervalo de tempo. Esse comportamento é conhecido como *master lease*, e é renovado periodicamente pelas réplicas, desde que o mestre possua a maioria dos votos. Se o mestre falhar, o protocolo de consenso é novamente executado para eleger um novo mestre.

Uma instância Chubby (também conhecida como célula do Chubby) atende a milhares de clientes, portanto, esses clientes estão se conectando a um mestre para todas as necessidades de coordenação. Os clientes enviam solicitações para o DNS (*Domain Name System*) para localizar o mestre. As réplicas respondem aos clientes que emitiram as solicitações DNS, redirecionando-os para o mestre atual. Depois que o cliente encontra o mestre, todas as solicitações vão para esse mestre. Os clientes executam o protocolo de bloqueio em nome do aplicativo e notificam o aplicativo sobre certos eventos, como *failover* principal.

O relacionamento entre uma célula do Chubby e um cliente é mantido por meio de uma sessão e o intervalo de tempo durante o qual a sessão existe é chamado de tempo de concessão. Além disso, o Chubby possui as seguintes características de *design*:

- *Coarse-grained*: os bloqueios consultivos concedidos aos arquivos são mantidos por horas ou dias, em vez de segundos ou menos; dessa maneira, é mais tolerante a falhas do servidor e economiza comunicações mais frequentes;
- Recursos de armazenamento de dados pequenos e manipulações de arquivos pequenos, além de um serviço de bloqueio;
- Permite que milhares de clientes observem mudanças. Desta maneira, o serviço de bloqueio precisa ser dimensionado para lidar com muitos clientes, embora a taxa de transação possa não ser tão alta;

- Mecanismo de notificação pelo qual o cliente sabe quando a alteração ocorre no arquivo compartilhado;
- Suporte de cache do lado do cliente para reduzir a carga do servidor Chubby para que o sistema possa manter sua escalabilidade;
- Garantias fortes de armazenamento em cache para simplificar seu uso pelo desenvolvedor.

O presente artigo não apresentou métricas de avaliação de desempenho, conforme discutido nos demais estudos.

2.4.3 Serviços de rede

SMaRtLight (2014)

O SMaRtLight [12] é uma arquitetura projetada para solucionar os problemas de tolerância a falhas em redes de computadores. Nessa arquitetura, um banco de dados compartilhado e replicado é utilizado para armazenar todas as informações de controle do controlador primário. E o controlador de *backup* lê as informações de controle do banco de dados. Esse banco de dados pode implementar efetivamente a sincronização de informações entre o controlador primário e o controlador de *backup*.

O controlador SMaRtLight possui uma arquitetura de três camadas composta por plano de dados (*data plane*), réplicas de controladores de rede e o banco de dados compartilhado. A Figura 2.9 ilustra a arquitetura SMaRtLight. Um controlador é configurado como primário, responsável pelo processamento de todas as solicitações do *switch* e os outros são configurados como *backups*. Os controladores coordenam suas ações através de um algoritmo de gerenciamento de concessão, que os controladores utilizam para detectar se um controlador primário está ativo.

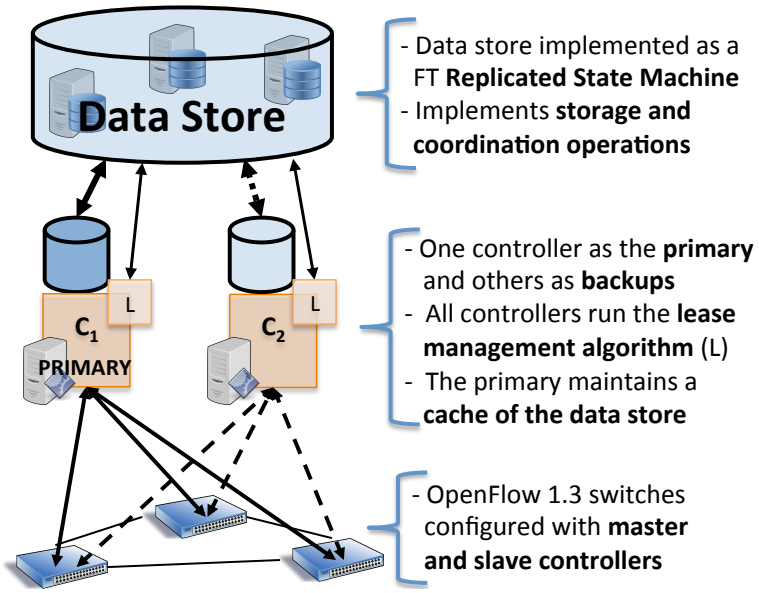


Figura 2.9: Arquitetura do SMaRtLight [12].

No caso de falha do controlador primário, é realizada uma transição para escolher um dos *backups* como um novo controlador primário. O controlador armazena todos os dados relacionados ao aplicativo em um banco de dados compartilhado, implementado através de uma RME. Essa base de dados mantém todas as informações relacionadas ao estado do aplicativo, como NIB (Base de Informações da Rede). Durante a substituição do controlador em falha, o novo controlador recebe uma atualização completa do banco de dados.

Para avaliação de desempenho, o CBench (*benchmark* de controladores) foi configurado para executar dez experimentos de dez segundos cada, com um número variável de *switches* (1-64) e um número fixo de *hosts* simulados (1.000). Para atingir o desempenho máximo, o CBench foi implantado na mesma máquina que o controlador primário, pois o tráfego gerado excedeu a largura de banda da NIC (*Network Interface Controller*).

A Figura 2.10 mostra a taxa de transferência obtida para um número variável de *switches*. Os percentuais representam a taxa de *cache hits*. A figura mostra que o sistema pode processar até 367 mil fluxos por segundo com 90% das operações absorvidas pelo cache. Para aplicações de gravação pesada com 50%, a taxa de transferência alcançada diminui para 96 mil fluxos por segundo e 55 mil fluxos por segundo com 10%.

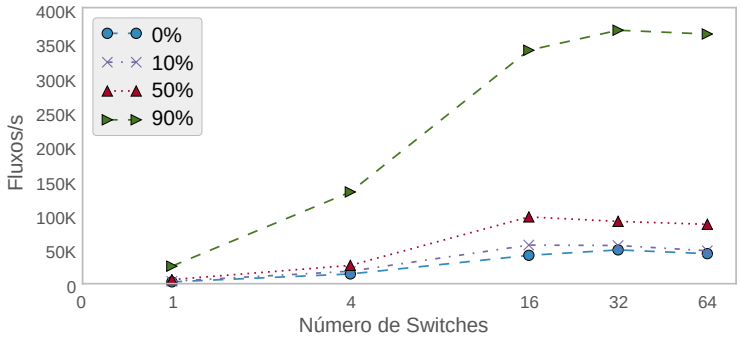


Figura 2.10: Taxa de transferência com *cache hits* diferente [12].

Para os autores, mesmo esta sendo versão preliminar do SMaRtLight, os resultados foram promissores, pois os aplicativos SDN podem ser tolerantes a falhas enquanto garantem uma taxa de processamento entre 100 mil fluxos por segundo e 350 mil fluxos por segundos, desde que ofereçam taxas de *cache hits* acima de 50%.

2.4.4 Serviços DNS

Secure Domain Name System (2008)

O Secure Domain Name System (SDNS) [53] é um esquema DNS seguro baseado em tolerância a intrusões. Esse DNS seguro utiliza técnicas tolerantes a intrusões bizantinas e mecanismos de votação, desta maneira fornece alta integridade, robustez e disponibilidade do serviço na presença de falhas arbitrárias, incluindo falhas provenientes de ataques maliciosos.

O modelo proposto pelo SDNS é formado por um grupo de servidores *proxy*, um grupo de servidores de nomes DNS, um servidor de votação e um servidor de gerenciamento de configuração, conforme ilustrado na Figura 2.11.

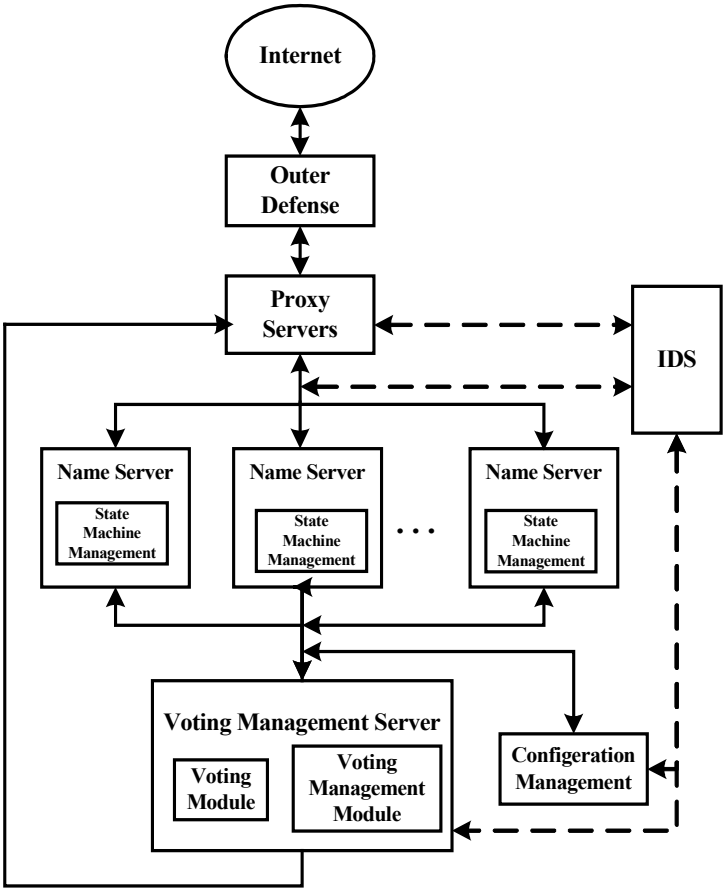


Figura 2.11: Arquitetura do sistema do SDNS [53].

O fluxo inicia através da solicitação de um usuário que chega a um servidor *proxy* por meio de uma defesa externa (e.g., um *firewall*). A entrada de todo o sistema ocorre pelo servidor *proxy*, que primeiro analisa a autenticidade da solicitação e envia a solicitação analisada para o DNS primário. Os servidores DNS, por sua vez, enviam o resultado ao servidor de votação, que em seguida transfere o resultado obtido com base no mecanismo de votação do servidor *proxy*, e por fim comunica o cliente. A comunicação e o gerenciamento entre servidores DNS usam um algoritmo tolerante a falhas bizantino aprimorado com base na RME.

Os servidores *proxy* e DNS são compostos de muitos servidores, dos quais apenas um servidor é o primário e os demais são designados para atuarem como secundários. No momento em que o sistema está em execução, o servidor primário é o responsável pelo processamento dos dados, enquanto os servidores secundários apenas monitoram a operação do mesmo. Quando o servidor primário falha, um dos servidores secundários é eleito como novo servidor primário. Esse comportamento garante o equilíbrio da carga e a confiabilidade do sistema melhora. Além disso, o IDS (*Intrusion detection System*) detecta ataques de fora e monitora a condição de execução dos servidores. Quando alguma anormalidade acontece, o IDS aciona o módulo de gerenciamento de configuração para reconfigurar o sistema.

O servidor de gerenciamento de votação consiste no módulo de gerenciamento de votação e no módulo de votação. Suas principais atribuições são:

- Receber as informações dos servidores DNS, votar de acordo com o algoritmo de votação, enviar o resultado da votação aos servidores *proxy* e analisar as informações para identificar os servidores defeituosos;
- Alterar os parâmetros do módulo de votação para melhorar o desempenho dos algoritmos de votação.

O gerenciamento de configuração é responsável pela manutenção do sistema de acordo as solicitações dos outros módulos. Além do mais, a reconfiguração pode ser dinâmica ou estática, dessa maneira o sistema melhora sua flexibilidade após uma falha ou invasão e adiciona dificuldade aos próximos ataques dos invasores.

Os autores não disponibilizaram avaliação de desempenho para essa proposta de DNS seguro.

2.4.5 Sistemas SCADA

SMaRt-SCADA (2018)

O SMaRt-SCADA [41] é um protótipo que tem como objetivo integrar o Eclipse NeoSCADA e o BFT-SMaRt, criando um sistema tolerante a intrusões. A principal finalidade desse sistema é assegurar a determinismo e a coordenação em um ambiente que, inicialmente, não foi projetado para ser determinístico.

A arquitetura proposta introduz o SMaRt-SCADA, uma solução BFT SCADA que visa a integração do Eclipse NeoSCADA [22] com o BFT-SMaRt [9]. Essa proposta foi simplificada utilizando proxies que permitem minimizar as modificações de código no sistema original. Desta maneira, cada componente original possui seu próprio *proxy* para acomodar o código BFT-SMaRt. A Figura 2.12 apresenta a arquitetura SMaRt-SCADA.

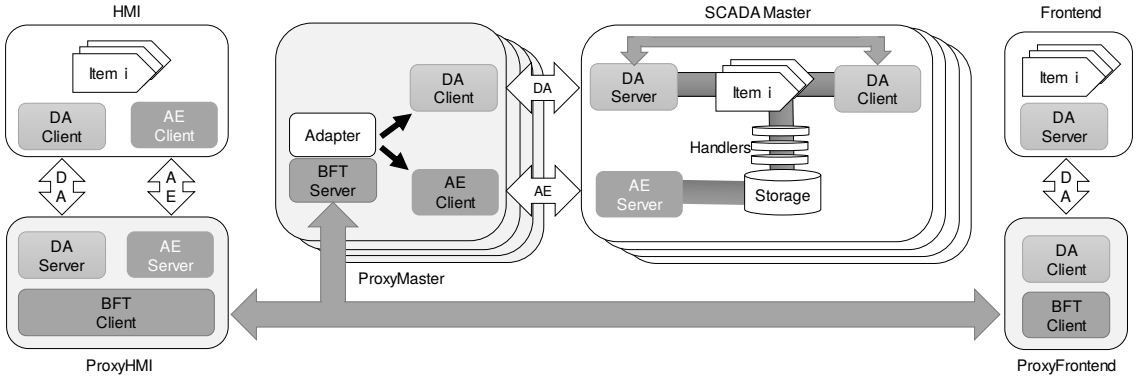


Figura 2.12: Os principais componentes do SMaRt-SCADA [41].

O *ProxyMaster* é responsável por encaminhar as mensagens do NeoSCADA provenientes do *Frontend* e da HMI para o SCADA *Master*. Cada *ProxyMaster* contém um servidor BFT, lado do servidor onde um protocolo de replicação BFT é executado. O servidor BFT se comunica com o adaptador responsável por adicionar informações a cada mensagem recebida e decidir para qual cliente a mensagem deve ser encaminhada, DA ou AE.

O *ProxyHMI* recebe as mensagens HMI e as envia por meio do cliente BFT ao *ProxyMaster*. No *ProxyMaster*, há um servidor DA e um servidor AE que simulam os servidores disponíveis no SCADA *Master*. Além disso, o *ProxyFrontend* é responsável por garantir a comunicação entre o *Frontend* e o SCADA *Master*. Esse *proxy* utiliza o cliente BFT da biblioteca para transmitir todas as mensagens provenientes do *Frontend* para o SCADA *Master*. Quando o SCADA *Master* precisa se comunicar com o *Frontend*, as mensagens são recebidas pelo *ProxyFrontend* no lado do cliente da biblioteca e encaminhadas utilizando o cliente DA.

A avaliação de desempenho foi realizada através da comparação da performance do SMaRt-SCADA com o NeoSCADA original.

O NeoSCADA foi implantado em três máquinas, cada uma contendo seu componente: *Frontend*, SCADA *Master* e HMI. Por outro lado, o SMaRt-SCADA foi implantado em seis máquinas: uma *Frontend*, quatro SCADA *Masters* e uma HMI, cada uma também contendo seu *proxy* correspondente.

A Figura 2.13 ilustra o número de mensagens processadas pelo NeoSCADA e SMaRt-SCADA. SMaRt-SCADA possui o desempenho 6% inferior devido às etapas adicionais necessárias para executar as atualizações. No NeoSCADA, cada mensagem *ItemUpdate* executa três etapas de comunicação para ir do *Frontend* ao HMI, mas no SMaRt-SCADA, a mesma operação leva 9 etapas, isso ocorre porque cada mensagem *ItemUpdate* é executada em cada SCADA *Master* após os *ProxyMasters* executarem um contrato bizantino e depois serem votados no *ProxyHMI*.

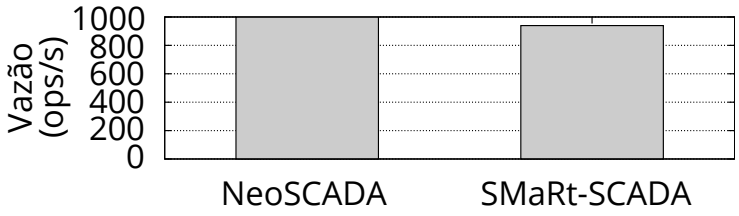


Figura 2.13: Avaliação de atualização de valor [41].

A Figura 2.14 mostra o número de mensagens processadas. As duas soluções foram executadas em dois cenários de alarme diferentes. Em um cenário, toda mensagem *ItemUpdate* aciona um alarme (100% de alarme), enquanto no outro, metade deles o faz (50% de alarme). O NeoSCADA processou todas as mensagens para as duas porcentagens de alarmes, mas o SMaRt-SCADA apresentou sobrecarga de 10% e 25% para os cenários de 50% e 100% de alarmes, respectivamente. A diminuição da taxa de transferência reflete as etapas adicionais de comunicação introduzidas pela solução. Em particular, no cenário de 100% de alarme, onde o número de eventos que vão para o armazenamento é o dobro do observado no cenário de 50% de alarme.

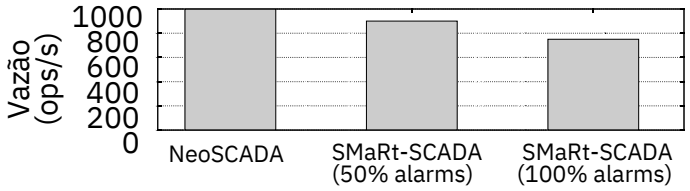


Figura 2.14: Avaliação de atualização de valor com o subsistema AE [41].

Na última experiência, o desempenho de ambas as soluções foram avaliadas para o caso de uso do valor de gravação. A HMI realiza gravações síncronas no item de um *Frontend*, ou seja, para cada operação de gravação, a HMI espera até que a operação seja concluída.

A Figura 2.15 ilustra o número de gravações que podem ser executadas nas duas soluções. O SMaRt-SCADA inicia com uma sobrecarga de 78%, consequência das 10 etapas adicionais de comunicação que a solução precisa para executar a operação de gravação. Além disso, o SCADA *Master* apresenta processamento sequencial, ou seja, não é capaz de tirar proveito dos processadores multinúcleos. Inclusive, foi possível notar que o BFT-SMaRt não é o gargalo do sistema, pois atinge uma taxa de transferência de 16k solicitações por segundo para um tamanho de mensagem semelhante (1024k bytes). No entanto, a taxa de transferência alcançada pelo SMaRt-SCADA é suficiente para acomodar uma carga de trabalho do mundo real, pois é praticamente impossível para um grupo de operadores humanos executar quase 100 comandos por segundo.

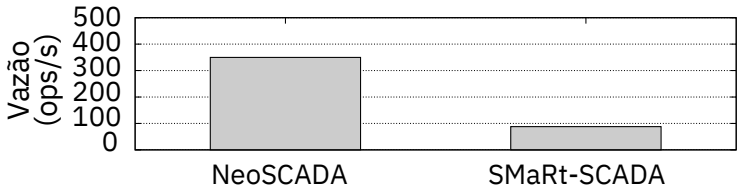


Figura 2.15: Avaliação de gravação de valor [41].

3. TPC-C SOBRE RME

3.1 Transações em RME

Ao considerar uma carga transacional sobre RME, deve-se garantir que a execução das transações ocorra de forma determinística e também adotar uma arquitetura que garanta as propriedades ACID (Atomicidade, Consistência, Isolamento e Durabilidade):

- Atomicidade: cada transação é tratada como uma unidade que tem sucesso (completo) ou então, em caso de não sucesso, não persiste nenhuma alteração;
- Consistência: cada transação inicia em um estado consistente da base de dados e deixa a mesma em um estado consistente;
- Isolamento: garante que a execução concorrente de transações deixa a base de dados em um estado equivalente a uma execução sequencial das mesmas;
- Durabilidade: uma vez que uma transação tenha sido efetivada, seus efeitos permanecerão a despeito de falhas no sistema.

Conforme o modelo de RME, os clientes submetem solicitações que são entregues totalmente ordenadas às réplicas, cabendo a elas a execução respeitando esta ordem. A Figura 3.1 apresenta um esquemático deste modelo de funcionamento. Em cada réplica, a execução de transações se desdobra em um conjunto de acessos aos itens de estado manipulados em cada transação.

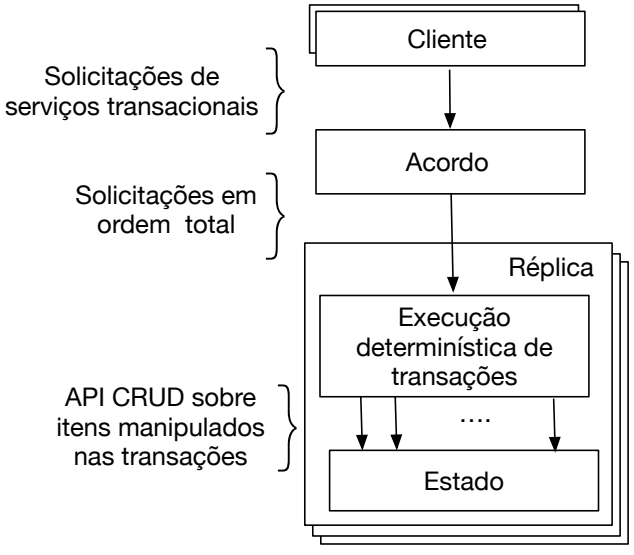


Figura 3.1: Serviço transacional sobre RME

ACID em RME sequencial.

Em bancos de dados, tipicamente o insucesso de uma transação está relacionado a dois fatores: falha no serviço; ou controle de concorrência que não permite efetivar uma transação em uma dada posição relativa às demais concorrentes. O modelo de RME propõe abordagens para tratar ambos os casos: devido à replicação, falhas são toleradas, enquanto que a ordenação total e execução conforme a ordem garantem que toda transação possa ser efetivada. Assim, garante-se atomicidade. Dado o modelo sequencial e o já exposto, a consistência também é garantida. O isolamento, no caso sequencial, é trivialmente mantido. Por fim, a durabilidade é mantida devido à tolerância a falhas da RME.

ACID em RME paralela.

Ao considerar RME paralela, é necessário levar em conta a execução concorrente de transações. Para manter o determinismo necessário na RME, o controle de concorrência das réplicas deve ser determinístico. Este é um aspecto comum abordado na literatura de RME paralela. Aqui é utilizada uma das abordagens para RME paralela, baseada na identificação de conflitos entre transações e manutenção da ordem total sempre que duas transações conflitam. Um par de transações tem conflito se uma delas modifica qualquer item lido ou escrito pela outra transação. Ao manter a ordem acordada entre transações conflitantes, garante-se que as réplicas executem deterministicamente e que as transações possam sempre ser efetivadas ao seu término. Com isso, de forma análoga ao caso sequencial, mantém-se a propriedade ACID.

ACID em RME no caso de recuperação.

Para permitir recuperação de uma réplica falha, adota-se o modelo típico de RME. Assume-se que *snapshots* de estados consistentes sejam tomados periodicamente (*checkpoints*) e o *log* de transações a partir do último *checkpoint* seja também mantido. Para a recuperação deve-se instalar o estado do último *checkpoint* e processar o *log* de transações. Note que tanto para RMEs sequenciais quanto paralelas, estes processamentos seguem conforme o caso de entrega normal das transações e, portanto, não afetam as propriedades ACID.

3.2 Integração do TPC-C em uma RME Sequencial

Conforme mencionado anteriormente, nesta pesquisa foi adotado o BFT-SMaRt como biblioteca para RME. Foi construído um serviço TPC-C (lado do servidor) e clientes complementares para geração de carga de trabalho.

Cada aspecto da especificação TPC-C foi considerado. Em algumas situações, foram utilizadas implementações disponíveis do TPC-C para avaliar como os autores interpretaram a especificação, tal como a implementação jTPCC[1] de código aberto e a produzida na tese de [32].

3.2.1 Modelo de dados sobre *key-value store* (KVS)

O armazenamento de valor-chave é adequado para armazenar dados não estruturados e semiestruturados devido à sua ausência de esquema, é um tipo de banco de dados não relacional que usa um método de chave-valor simples para armazenar dados. Diferentemente dos bancos de dados relacionais que armazenavam dados em tabelas e colunas, um banco de dados de chave-valor usa chaves individuais ou combinações de chaves para recuperar os valores associados. Juntos, eles são conhecidos como pares de valores-chave. Esses valores podem ser desde tipos de dados simples, como *strings* ou inteiros, até objetos complexos com vários valores aninhados. Cada chave é um identificador exclusivo que mapeia para um valor ou local dos dados armazenados.

O princípio desses bancos de dados é semelhante ao objeto armazenado em cache de memória distribuída em um sistema de cache. Os KVSs usam os comandos "*get*", "*put*" e "*delete*" para recuperar, armazenar e atualizar dados, pois não possui linguagem de consulta, também não possuem restrições (*constraints*) e nem chave estrangeira. Geralmente são usados para pesquisas paralelas e operações de leitura intensiva [26].

Cada linha de uma tabela é convertida em uma entrada de chave-valor. A chave primária do banco de dados e seu conteúdo são mapeados respectivamente para chave e valor no armazenamento de chave-valor. O valor armazenado é um objeto com o conteúdo específico da respectiva tabela. Para cada tabela, existe uma classe específica que descreve os objetos que representam suas linhas.

[1] http://jtpcc.sourceforge.net/

3.2.2 Serviço TPC-C

Cada réplica da RME hospeda o banco de dados TPC-C em memória, com seu estado, e responde às 5 transações possíveis (Seção 2.3).

Tabelas.

Conforme mencionado na Seção 2.3, o banco de dados consiste em 9 tabelas. Cada réplica usa um armazenamento de chave-valor subjacente para hospedar as tabelas. Cada linha de uma tabela é convertida em uma entrada de chave-valor. A chave primária do banco de dados e seu conteúdo são mapeados respectivamente para chave e valor no armazenamento de chave-valor. O valor armazenado é um objeto com o conteúdo específico da respectiva tabela. Para cada tabela, existe uma classe específica que descreve os objetos que representam suas linhas.

Para organizar a modelagem e os dados de forma equivalente ao modelo relacional proposto pelo TPC-C, utilizou-se o seguinte esquema: uma chave no armazenamento chave-valor é composta de um prefixo que é usado que descrever a entidade, e um sufixo que é a chave primária, ou composta, da respectiva entidade, de acordo com o TPC-C. Na Figura 3.2 é apresentado um exemplo para a tabela DISTRICT. O prefixo deve ser definido como “DISTRICT". Neste caso, a chave é composta e, portanto, o sufixo deve ser “D_W_ID” e “D_I”, correspondendo respectivamente ao identificador do armazém e ao identificador do distrito, conforme especificado no TPC-C. Na Figura 3.2 à direita, é apresentado um exemplo da chave no formato chave-valor, como "DISTRICT:1:1", que indica o distrito 1 do armazém 1, sendo o valor o objeto em azul.

Portanto, é possível representar uma tabela do modelo relacional em um esquema de chave-valor da seguinte forma: `table_name:primary_key_1:primary_key_2 = <value>`. Assim, todos os dados das tabelas junto com seus relacionamentos estão contidos neste esquema de chave-valor. Embora esse cenário seja um relacionamento muitos para um (existem 10 distritos para cada armazém), uma abordagem semelhante também pode ser usada para outros tipos de relacionamentos. Em relacionamentos muitos para muitos haverá uma entidade intermediária (tabela) que mapeia as chaves estrangeiras de ambas as tabelas. Nesses cenários, é possível primeiro representar as duas tabelas primárias em um esquema de chave-valor e, em seguida, representar a tabela intermediária.

DISTRICT Table Layout

Field Name	Field Definition	Comments
D_ID	20 unique IDs	*10 are populated per warehouse*
D_W_ID	2*W unique IDs	
D_NAME	variable text, size 10	
D_STREET_1	variable text, size 20	
D_STREET_2	variable text, size 20	
D_CITY	variable text, size 20	
D_STATE	fixed text, size 2	
D_ZIP	fixed text, size 9	
D_TAX	signed numeric(4,4)	*Sales tax*
D_YTD	signed numeric(12,2)	*Year to date balance*
D_NEXT_O_ID	10,000,000 unique IDs	*Next available Order number*

Primary Key: (D_W_ID, D_ID)

D_W_ID Foreign Key, references W_ID

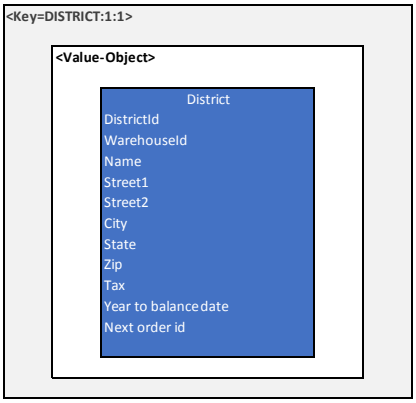


Figura 3.2: Layout (esquerda) [19] e KV (direita) da tabela DISTRICT.

Transações.

Cada uma das 5 transações TPC-C é mapeada para uma operação do serviço. Como as réplicas do serviço recebem a mesma ordem total de transações e as processam de forma determinística, as réplicas se mantêm consistentes.

3.2.3 Cliente TPC-C

Os clientes implementados geram uma carga de trabalho do TPC-C. Cada cliente é uma *thread* em *loop* fechado, que constrói a transação, envia para a RME usando *broadcast* atômico e aguarda a resposta. A especificação TPC-C estabelece distribuições de probabilidade para variáveis como tipo de transação, identificação do cliente, escolha de itens e estoque. Os clientes seguem essas distribuições ao construir as transações, além de permitir ajustes para outras distribuições.

Uma das vantagens de se utilizar essa abordagem para gerar cargas de trabalho é a flexibilidade. Além de seguir as distribuições de probabilidade estabelecidas pela especificação TPC-C, os clientes também permitem ajustes para outras distribuições. Isso permite gerar cargas de trabalho que possam ser livres de conflitos ou propensas a conflitos. Isso é particularmente útil em situações em que se deseja simular uma carga de trabalho específica, como em testes de stress ou análise de desempenho.

Depois que os lados do cliente e do servidor do TPC-C são definidos, todos os componentes necessários para implementar uma RME sequencial estarão disponíveis no BFT-SMaRt. No entanto, isso não é suficiente para o caso paralelo.

3.3 Integração do TPC-C em uma RME Paralela

Dentre as várias estratégias para escalonar comandos em uma RME paralela [42], foi escolhido neste estudo o escalonar tardio [21] por sua relativa simplicidade e alto desempenho. O escalonador tardio tem uma arquitetura intuitiva: conforme as requisições de transação chegam, um escalonador calcula as dependências relacionadas com as transações existentes (aguardando e em execução), armazenando as transações com as respectivas dependências em um grafo de dependência. Os nós representam as transações e as arestas suas dependências. Devido à entrega totalmente ordenada e, como qualquer transação pode depender apenas de transações já entregues, este grafo de dependência é direcionado e acíclico. Assim, sempre existe pelo menos uma transação que pode ser executada. Nesse caso, uma de um *pool* de *worker threads* obtêm a transação livre para executar. Após a execução, a *worker thread* remove a transação do grafo de dependência, bem como as arestas que se conectam às transações dependentes. Estas, por sua vez, podem ficar prontas para serem executadas, de forma que indutivamente a qualquer momento haja pelo menos uma transação livre para executar (ou em execução).

Para permitir o uso desta abordagem, deve-se definir uma função de conflito que, dadas duas transações, calcula se elas são conflitantes ou não. Esta função deve ser total, ou seja, dadas quaisquer transações e seus parâmetros, a função deve ser capaz de calcular um resultado. Se duas transações forem consideradas conflitantes, a execução segue a ordem total acordada em consenso, enquanto transações não conflitantes podem ser executadas em paralelo. A detecção de um conflito pode ser demorada, portanto, a precisão dessa função também é uma questão de projeto. No caso de uma detecção de conflito complexa, seu cálculo pode ser ignorado e um conflito assumido sem prejudicar a consistência. Por outro lado, apenas transações seguramente independentes podem ser consideradas independentes.

Vale a pena mencionar que a abordagem RME para sistemas transacionais é atraente, pois é possível considerar a execução das transações tolerante a falhas devido à replicação, e a detecção de conflitos garante que as transações não entrem em conflito antes da execução. Dessa forma, qualquer execução que termine é linearizável e, portanto, os protocolos de confirmação podem ser ignorados.

Para efeito de medição, foi implementado um serviço paralelo que, ao responder ao cliente, também informa se a transação conflitou com outras. Dessa forma, o cliente é capaz de calcular a parcela de suas transações que teve conflito detectado no lado do servidor. Assim, os experimentos mostram conflitos reais e não apenas a parcela de operações que podem entrar em conflito devido a itens comuns utilizados. Observa-se que um conflito real só surge se ambas as operações usarem os mesmos itens e coexistirem no lado do servidor, o que geralmente não ocorre.

3.3.1 Função de detecção de conflito para TPC-C com escalonamento tardio

A função de detecção de conflito para TPC-C foi definida utilizando uma avaliação *pairwise* das transações. A verificação foi realizada para determinar se o conjunto de gravação de uma transação acessa o conjunto de leitura ou gravação da outra transação. A Tabela 3.1 apresenta as especificações da função de conflito. Devido à simetria, a tabela apresenta apenas a parte inferior da matriz.

Tabela 3.1: Definição da função de conflito para o TPC-C.

Legenda:
✓ : sem conflito;
● : conflito possível, se as transações possuem o mesmo cliente;
■ : com conflito, independente dos parâmetros da transação.

	New-Order	Payment	Order-Status	Delivery	Stock-Level
New-Order	1. ■	-	-	-	-
Payment	2. ■	6. ●	-	-	-
Order-Status	3. ●	7. ●	10. ✓	-	-
Delivery	4. ■	8. ■	11. ●	13. ■	-
Stock-Level	5. ■	9. ✓	12. ✓	14. ✓	15. ✓

A seguir é explicado cada célula preenchida da tabela de conflitos usando como referência o número dentro das células.

1. **New-Order e New-Order:** conflitantes, pois há a possibilidade de conflito quando duas transações T1 e T2 estão tentando inserir dados para o mesmo cliente. Nessa situação, T2 deverá aguardar a conclusão de T1 antes de prosseguir. O valor de D_NEXT_O_ID reflete essa interação entre as transações, pois ele é incrementado em dois devido à execução de T1 e T2, e é um número maior que o número do pedido para T2.

2. **New-Order e Payment:** há conflito entre as transações, pois ambas realizam atualizações na tabela DISTRICT. A transação New-Order tem como objetivo incrementar e atualizar o número da próxima ordem de pedido (D_NEXT_ORDER_ID), enquanto a transação Payment atualiza as informações de saldo (D_YTD). Como ambos os procedimentos alteram informações na mesma tabela, existe uma possibilidade de interferência entre as operações. Caso essa interferência não seja tratada adequadamente, pode haver inconsistência nos dados.

3. **New-Order e Order-Status:** há conflito se a consulta ocorrer para o mesmo cliente, tendo em vista que o primeiro atualiza as informações lidas pelo segundo.

4. **New-Order e Delivery:** há conflito, pois ambos atualizam a tabela NEW_ ORDER, a transação Delivery, por exemplo, realiza exclusão do registro mais antigo da tabela NEW_ ORDER que esteja associado ao distrito.

5. **New-Order e Stock-Level:** há conflito, pois uma transação New-Order atualiza as informações de nível de estoque.

6. **Payment e Payment:** há conflito se for para o mesmo cliente, pois estas transações atualizam as mesmas informações da tabela CUSTOMER, DISTRICT e WAREHOUSE.

7. **Payment e Order-Status:** há conflito para o mesmo cliente, pois ambos acessam o saldo do cliente e a transação Payment o atualiza.

8. **Payment e Delivery:** há conflito, pois a transação Delivery entrega um lote de novos pedidos e pode acessar dados do mesmo cliente para o qual o pagamento está sendo executado. A partir dos dados contidos nestas transações, não há como detectar se elas realmente conflitam. Assim, um conflito é assumido.

9. **Payment e Stock-Level:** não há conflito, pois nenhum item lido por uma transação é escrito pela outra.

10. **Order-Status e Order-Status:** não há conflito, pois ambos são somente leitura.

11. **Order-Status e Delivery:** há conflito caso exista duas transações para um mesmo cliente, pois ocorre atualização de dados na Delivery, que são consultados na Order-Status.

12. **Order-Status e Stock-Level:** não há conflito, pois ambos são somente leitura.

13. **Delivery e Delivery:** há conflito, pois ambos atualizam a tabela NEW_ ORDER.

    Há conflito se houver mais de uma operação de escrita (como entregas) para o mesmo cliente, pois o saldo (C_BALANCE) precisará ser atualizado para refletir o valor das duas entregas simultaneamente, causando possíveis problemas de consistência.

14. **Delivery e Stock-Level:** não há conflito, pois acessam tabelas separadas.

15. **Stock-Level e Stock-Level:** não há conflito, pois ambos são somente leitura.

4. EXPERIMENTOS E RESULTADOS

Após desenvolver o *benchmark* TPC-C para operar tanto de forma sequencial quanto em paralelo, foi realizada uma série de experimentos para avaliar o desempenho dessas abordagens, utilizando uma carga de trabalho representativa de aplicações reais.

4.1 Configurações

Ambiente computacional.

O ambiente experimental foi configurado com 7 máquinas conectadas via rede comutada de 1Gbps. O software instalado nas máquinas foi Linux Ubuntu com kernel 4.15.0 (64 bits) e máquina virtual Java 64 bits versão 11.0.3. O BFT-SMaRt foi configurado com 3 réplicas hospedadas em máquinas separadas (processador AMD Opteron® com 32 núcleos físicos e 64 núcleos lógicos através de *hyper-threading* e 126 GB de RAM) para tolerar falha de até 1 réplica em modo *crash*, enquanto que até 200 clientes foram distribuídos uniformemente em outras 4 máquinas (processador Intel® Xeon® com 8 núcleos físicos e 8 GB de RAM).

Aspectos metodológicos.

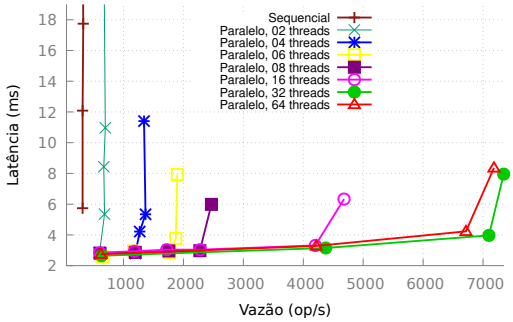
O banco de dados foi pré-populado. Os experimentos consideram apenas um armazém. De acordo com o TPC-C, um armazém possui 10 distritos. Cada distrito tem 3.000 clientes e 100.000 itens de estoque. Cada cliente tem inicialmente um pedido em andamento. Cada configuração de *worker threads*, *threads* de clientes e carga de trabalho é executada da seguinte maneira: após o início, as primeiras 100 transações de cada cliente são consideradas aquecimento; então, durante 120 segundos, as transações são submetidas e medidas de vazão, latência e conflito são feitas; após o período de medição, cada *thread* cliente ainda envia 100 transações antes de parar (para lidar com diferentes tempos de parada nos clientes e manter a mesma carga de trabalho nos servidores). Depois de concatenar todos os resultados das *threads* clientes para cada execução, obteve-se a taxa média de vazão (*throughput*), latência e conflitos.

4.2 Resultados e Análises

A seguir são reportados os resultados para três cenários diferentes: cargas de trabalho livres de conflitos, propensas a conflitos e a carga padrão do TPC-C.

Livre de Conflitos.

A carga de trabalho livre de conflitos permite entender como o serviço escala à medida que mais *worker threads* são adicionados no modelo paralelo, comparando com o caso sequencial. Esta carga é construída pelas 2 transações somente leitura do TPC-C: Order-Status e Stock-Level, sendo a primeira uma transação de peso pesado e a segunda uma transação de peso médio, em partes iguais.



# worker threads	1	2	4	6
Threads em ktxns/seg	0.3	0.6	1.2	1.9
Delay em ms	5.7	2.8	2.8	3.7
Speedup	1	2	4	6.3
# worker threads	8	16	32	64
Threads em ktxns/seg	2.3	4.2	7.1	6.7
Delay em ms	2.9	3.3	3.9	4
Speedup	7.7	14	23.7	22.3

Legenda:
ktxns: k de *kilo* sendo x 1000; txns sendo transações;

Figura 4.1: Cenário livre de conflitos (50% *Order-Status* e 50% *Stock-Level*).

A Figura 4.1 apresenta os resultados para este experimento. Pode-se observar que a vazão escala com o número de *worker threads* até 32, gerando um *speedup* de 23.7 quando comparado com o modelo sequencial. De 32 a 64, o *hardware* disponível utiliza *hyper-threading*, sendo esta uma possível interpretação para o desempenho ter estabilizado.

Propenso a Conflitos.

Para investigar o impacto dos conflitos na vazão, foram analisados dois cenários. O primeiro é derivado do caso livre de conflitos, introduzindo transações de pagamento, tendo a seguinte configuração: 45% de Order-Status, 45% de Stock-Level e 10% de Payment. Nos vários experimentos deste cenário, observou-se que 5% a 8% das transações conflitavam com outras já entregues (sendo processadas ou aguardando a execução). Conforme ilustrado na Figura 4.2, esses níveis de conflito fizeram com que a vazão caísse para menos de 1900 txns/seg, enquanto que no cenário sem conflito este valor ficou próximo de 7000 txns/seg.

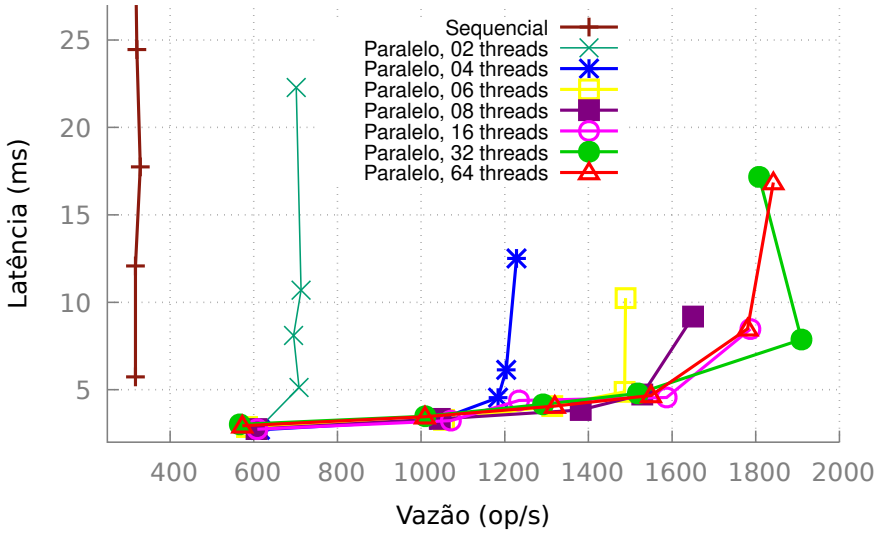


Figura 4.2: Conflito medido de 5 a 8%.

O segundo cenário introduz ainda as transações de novos pedidos e entregas, resultando na seguinte configuração: 40% de Order-Status, 40% de Stock-Level, 10% de Payment, 5% de New-Order e 5% de Delivery. Neste cenário, foi observado um índice de 18% a 24% de conflitos, o que fez a vazão cair para menos de 600 txns/s (Figura 4.3).

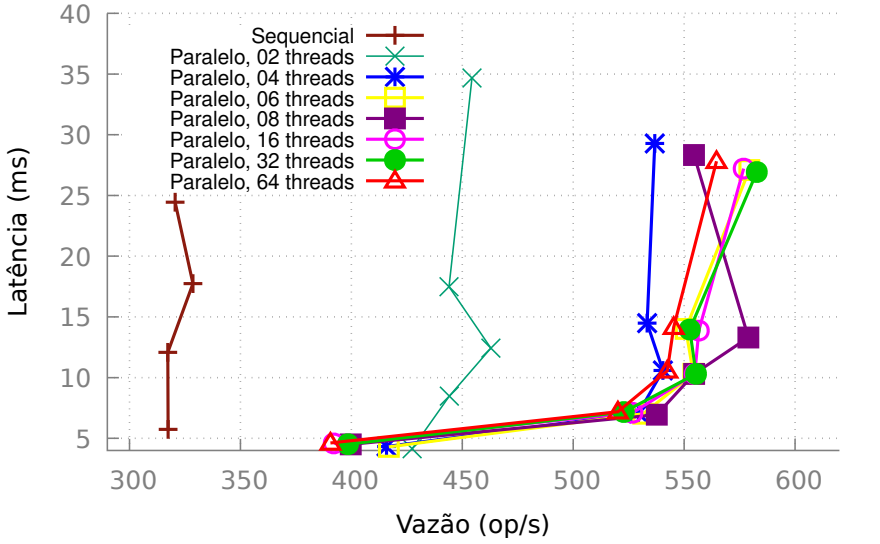


Figura 4.3: Conflito medido de 18 a 24%.

Carga de trabalho padrão do TPC-C.

Por fim, os clientes foram ajustados para gerar a carga de trabalho exatamente como especificada no TPC-C: 45% de New-Order; 43% de Payment; 4% de Delivery; 4% de Order-Status; e 4% de Stock-Level. A Figura 4.4 apresenta os resultados para este cenário, onde a taxa de conflito aumentou consideravelmente. Com apenas 2 clientes gerando transações (o primeiro ponto de cada linha no gráfico), foram observadas taxas de conflito no intervalo de 47% a 53%. Quando a carga de trabalho é ligeiramente au-

mentada (segundo ponto de cada linha no gráfico), a vazão também aumenta. Note que a latência também aumenta, mas permanece abaixo de 10ms. Para estes casos, o conflito observado foi próximo de 77%.

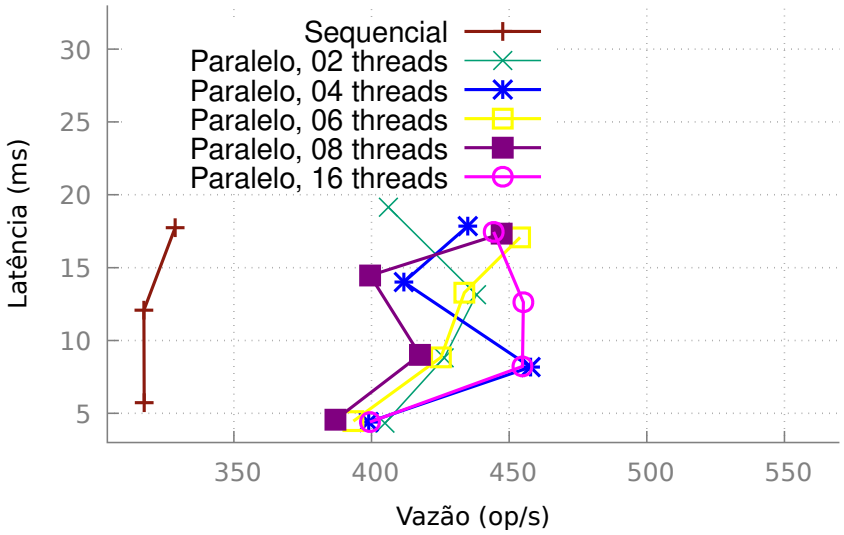


Figura 4.4: Carga de trabalho padrão do TPC-C.

A partir deste ponto, o serviço fica saturado e aumentar a carga de trabalho não aumenta a vazão, isso significa que mais transações estão enfileiradas e, portanto, o conflito observado também tende a aumentar. Com efeito, para latências superiores a 15 ms, os conflitos observados se aproximam de 82%.

5. CONSIDERAÇÕES FINAIS

A função de conflito detecta se existe alguma possibilidade de um item acessado por uma transação ser atualizado pela outra. Embora possível, em vários casos nenhum conflito real acontecerá. Esses falsos positivos derivam da definição da função, que tem como entrada apenas os parâmetros das transações submetidas e sempre que as informações não forem suficientes para garantir a ausência de conflito, um conflito é assumido. Portanto, a carga de trabalho TPC-C mostrou altas taxas de conflito medidas, levando a uma alta sequencialização na execução.

Vários aspectos podem contribuir aqui. Uma investigação de falsos positivos e refinamento dessa função é um passo natural. Além disso, estruturas alternativas para hospedar dados em réplicas podem levar a tempos de atendimento menores, reduzindo a população do escalonador de transações e, consequentemente, diminuindo os conflitos.

O código fonte do TPC-C para a plataforma BFT-SMaRt, bem como os resultados dos experimentos realizados, estão disponíveis em https://github.com/kayelserafim/bft-smart-tpcc.

5.1 Trabalhos futuros

Como o BFT-SMaRt possui vários aprimoramentos para favorecer o desempenho da RME, vale a pena considerar uma investigação mais aprofundada dessa carga de trabalho com outros mecanismos integrados. A implementação TPC-C atual oferece suporte a vários armazéns e esse cenário pode ser valioso para avaliar os mecanismos existentes para RME particionada.

REFERÊNCIAS BIBLIOGRÁFICAS

[1] Abu-Libdeh, H.; Princehouse, L.; Weatherspoon, H. “Racs: A case for cloud storage diversity”. In: Proceedings of the 1st ACM Symposium on Cloud Computing, 2010, pp. 229–240.

[2] Alchieri, E.; Dotti, F.; Mendizabal, O. M.; Pedone, F. “Reconfiguring parallel state machine replication”. In: Proceedings of the 36th IEEE Symposium on Reliable Distributed Systems, 2017, pp. 104–113.

[3] Alchieri, E.; Dotti, F.; Pedone, F. “Early scheduling in parallel state machine replication”. In: Proceedings of the ACM Symposium on Cloud Computing, 2018, pp. 82–94.

[4] Attiya, H.; Welch, J. “Distributed Computing: Fundamentals, Simulations, and Advanced Topics”. John Wiley & Sons, Inc., 2004, 432p.

[5] Batista, E.; Alchieri, E.; Dotti, F.; Pedone, F. “Early scheduling on steroids: Boosting parallel state machine replication”, *Journal of Parallel and Distributed Computing*, vol. 163, mai. 2022, pp. 269–282.

[6] Berger, M.; Honda, K. “The two-phase commitment protocol in an extended π-calculus”, *Electronic Notes in Theoretical Computer Science*, vol. 39–1, fev. 2003, pp. 21–46.

[7] Bessani, A.; Correia, M.; Quaresma, B.; André, F.; Sousa, P. “Depsky: Dependable and secure storage in a cloud-of-clouds”, *ACM Transactions on Storage*, vol. 9–4, nov. 2013, pp. 1–33.

[8] Bessani, A.; Mendes, R.; Oliveira, T.; Neves, N.; Correia, M.; Pasin, M.; Verissimo, P. “Scfs: A shared cloud-backed file system”. In: Proceedings of the USENIX Conference on USENIX Annual Technical Conference, 2014, pp. 169–180.

[9] Bessani, A.; Sousa, J.; Alchieri, E. E. “State machine replication for the masses with bft-smart”. In: Proceedings of the 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, 2014, pp. 355–362.

[10] Bessani, A. N.; Alchieri, E. P.; Correia, M.; Fraga, J. S. “Depspace: A byzantine fault-tolerant coordination service”, *SIGOPS Operating Systems Review*, vol. 42–4, abr. 2008, pp. 163–176.

[11] Bezerra, C. E.; Pedone, F.; Van Renesse, R. “Scalable state-machine replication”. In: Proceedings of the 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, 2014, pp. 331–342.

[12] Botelho, F.; Bessani, A.; Ramos, F. M.; Ferreira, P. “On the design of practical fault-tolerant sdn controllers”. In: Proceedings of the 3rd European Workshop on Software Defined Networks, 2014, pp. 73–78.

[13] Burgos, A.; Alchieri, E.; Dotti, F.; Pedone, F. “Exploiting concurrency in sharded parallel state machine replication”, *IEEE Transactions on Parallel and Distributed Systems*, vol. 33–9, set. 2022, pp. 2133–2147.

[14] Burrows, M. “The chubby lock service for loosely-coupled distributed systems”. In: Proceedings of the 7th Symposium on Operating Systems Design and Implementation, 2006, pp. 335–350.

[15] Băsescu, C.; Cachin, C.; Eyal, I.; Haas, R.; Sorniotti, A.; Vukolić, M.; Zachevsky, I. “Robust data sharing with key-value stores”. In: Proceedings of the IEEE/IFIP International Conference on Dependable Systems and Networks, 2012, pp. 1–12.

[16] Castro, M.; Liskov, B. “Practical byzantine fault tolerance and proactive recovery”, *ACM Transactions on Computer Systems*, vol. 20–4, nov. 2002, pp. 398–461.

[17] Chang, F.; Dean, J.; Ghemawat, S.; Hsieh, W. C.; Wallach, D. A.; Burrows, M.; Chandra, T.; Fikes, A.; Gruber, R. E. “Bigtable: A distributed storage system for structured data”, *ACM Transactions on Computer Systems*, vol. 26–2, jun. 2008, pp. 1–26.

[18] Corbett, J. C.; Dean, J.; Epstein, M.; Fikes, A.; Frost, C.; Furman, J. J.; Ghemawat, S.; Gubarev, A.; Heiser, C.; Hochschild, P.; Hsieh, W.; Kanthak, S.; Kogan, E.; Li, H.; Lloyd, A.; Melnik, S.; Mwaura, D.; Nagle, D.; Quinlan, S.; Rao, R.; Rolig, L.; Saito, Y.; Szymaniak, M.; Taylor, C.; Wang, R.; Woodford, D. “Spanner: Google’s globally distributed database”, *ACM Transactions on Computer Systems*, vol. 31–3, ago. 2013, pp. 1–26.

[19] Council, T. P. P. “Tpc benchmark c - standard specification - revision 5.11”, Relatório Técnico, Transaction Processing Performance Council, 2010, 132p.

[20] Cowling, J.; Liskov, B. “Granola: Low-Overhead distributed transaction coordination”. In: Proceedings of the USENIX Annual Technical Conference, 2012, pp. 223–235.

[21] Escobar, I. A.; Alchieri, E.; Dotti, F. L.; Pedone, F. “Boosting concurrency in parallel state machine replication”. In: Proceedings of the 20th International Middleware Conference, 2019, pp. 228–240.

[22] Foundation, E. “Neoscada”. Capturado em: https://www.eclipse.org/eclipsescada, Julho 2020.

[23] Hadzilacos, V.; Toueg, S. “A modular approach to fault-tolerant broadcasts and related problems”, Relatório Técnico, Cornell University, 1994, 83p.

[24] Herlihy, M. P.; Wing, J. M. “Linearizability: A correctness condition for concurrent objects”, *ACM Transactions on Programming Languages and Systems*, vol. 12–3, jul. 1990, pp. 463–492.

[25] Hunt, P.; Konar, M.; Junqueira, F. P.; Reed, B. “Zookeeper: Wait-free coordination for internet-scale systems”. In: Proceedings of the USENIX Conference on USENIX Annual Technical Conference, 2010, pp. 1–11.

[26] Jang, J.; Cho, Y.; Jung, J.; Jeon, G. “Enhancing lookup performance of key-value stores using cuckoo hashing”. In: Proceedings of the Research in Adaptive and Convergent Systems, 2013, pp. 487–489.

[27] Kapritsos, M.; Wang, Y.; Quema, V.; Clement, A.; Alvisi, L.; Dahlin, M. “All about eve: Execute-verify replication for multi-core servers”. In: Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation, 2012, pp. 237–250.

[28] Kotla, R.; Dahlin, M. “High throughput byzantine fault tolerance”. In: Proceedings of the International Conference on Dependable Systems and Networks, 2004, pp. 575–584.

[29] Lakshman, A.; Malik, P. “Cassandra: a decentralized structured storage system”, *ACM SIGOPS Operating Systems Review*, vol. 44–2, abr. 2010, pp. 35–40.

[30] Lamport, L. “Time, clocks, and the ordering of events in a distributed system”, *Communications of the ACM*, vol. 21–7, jul. 1978, pp. 558–565.

[31] Lamport, L. “The part-time parliament”, *ACM Transactions on Computer Systems*, vol. 16–2, mai. 1998, pp. 133–169.

[32] Le, L. H. “Scaling state machine replication”, Tese de Doutorado, Università della Svizzera italiana, 2020, 112p.

[33] Le, L. H.; Bezerra, C. E.; Pedone, F. “Dynamic scalable state machine replication”. In: Proceedings of the 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, 2016, pp. 13–24.

[34] Li, Z.; Van Roy, P.; Romano, P. “Enhancing throughput of partially replicated state machines via multi-partition operation scheduling”. In: Proceedings of the 16th IEEE International Symposium on Network Computing and Applications, 2017, pp. 1–10.

[35] Marandi, P. J.; Bezerra, C. E.; Pedone, F. “Rethinking state-machine replication for parallelism”. In: Proceedings of the 34th IEEE International Conference on Distributed Computing Systems, 2014, pp. 368–377.

[36] Marandi, P. J.; Pedone, F. “Optimistic parallel state-machine replication”. In: Proceedings of the 33rd IEEE International Symposium on Reliable Distributed Systems, 2014, pp. 57–66.

[37] Marandi, P. J.; Primi, M.; Pedone, F. “High performance state-machine replication”. In: Proceedings of the 41st IEEE/IFIP International Conference on Dependable Systems Networks, 2011, pp. 454–465.

[38] Mendizabal, O. M.; De Moura, R. S. T.; Dotti, F. L.; Pedone, F. “Efficient and deterministic scheduling for parallel state machine replication”. In: Proceedings of the IEEE International Parallel and Distributed Processing Symposium, 2017, pp. 748–757.

[39] Mu, S.; Nelson, L.; Lloyd, W.; Li, J. “Consolidating concurrency control and consensus for commits under conflicts”. In: Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation, 2016, pp. 517–532.

[40] Nogueira, A.; Casimiro, A.; Bessani, A. “Elastic state machine replication”, *IEEE Transactions on Parallel and Distributed Systems*, vol. 28–9, set. 2017, pp. 2486–2499.

[41] Nogueira, A.; Garcia, M.; Bessani, A.; Neves, N. “On the challenges of building a bft scada”. In: Proceedings of the 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, 2018, pp. 163–170.

[42] Pedone, F.; Alchieri, E.; Dotti, F.; Marandi, P.; Mendizabal, O. “Boosting state machine replication with parallel execution”. In: Proceedings of the 8th Latin-American Symposium on Dependable Computing, 2018, pp. 77–86.

[43] Rath, N. “S3ql - a full-featured file system for online data storage”. Capturado em: https://github.com/s3ql/s3ql, Julho 2020.

[44] Rath, N. “S3ql 1.13.2 documentation: Known issues”. Capturado em: http://www.rath.org/s3ql-docs/issues.html, Julho 2020.

[45] S3, A. “S3fs - fuse-based file system backed by amazon s3”. Capturado em: https://github.com/s3fs-fuse/s3fs-fuse, Julho 2020.

[46] Schiper, N.; Sutra, P.; Pedone, F. “P-store: Genuine partial replication in wide area networks”. In: Proceedings of the 29th IEEE Symposium on Reliable Distributed Systems, 2010, pp. 214–224.

[47] Schneider, F. B. “Implementing fault-tolerant services using the state machine approach: A tutorial”, *ACM Computing Surveys*, vol. 22–4, dez. 1990, pp. 299–319.

[48] Shvachko, K.; Kuang, H.; Radia, S.; Chansler, R. “The hadoop distributed file system”. In: Proceedings of the 26th IEEE symposium on mass storage systems and technologies, 2010, pp. 1–10.

[49] Tarasov, V.; Bhanage, S.; Zadok, E.; Seltzer, M. “Benchmarking file system benchmarking: It *is* rocket science”. In: Proceedings of the 13th USENIX Conference on Hot Topics in Operating Systems, 2011, pp. 1–9.

[50] Tarasov, V.; Zadok, E.; Shepler, S. “Filebench: A flexible framework for file system benchmarking”. Capturado em: https://www.usenix.org/system/files/login/articles/login_spring16_02_tarasov.pdf, Julho 2020.

[51] Thomson, A.; Diamond, T.; Weng, S.-C.; Ren, K.; Shao, P.; Abadi, D. J. “Calvin: Fast distributed transactions for partitioned database systems”. In: Proceedings of the ACM SIGMOD International Conference on Management of Data, 2012, pp. 1–12.

[52] Verma, A.; Pedrosa, L.; Korupolu, M.; Oppenheimer, D.; Tune, E.; Wilkes, J. “Large-scale cluster management at google with borg”. In: Proceedings of the 10th European Conference on Computer Systems, 2015, pp. 1–17.

[53] Zhou, W.; Chen, L. “A secure domain name system based on intrusion tolerance”. In: Proceedings of the International Conference on Machine Learning and Cybernetics, 2008, pp. 3535–3539.

Pontifícia Universidade Católica do Rio Grande do Sul
Pró-Reitoria de Pesquisa e Pós-Graduação
Av. Ipiranga, 6681 – Prédio 1 – Térreo
Porto Alegre – RS – Brasil
Fone: (51) 3320-3513
E-mail: propesq@pucrs.br
Site: www.pucrs.br