Benchmark TPC-C Aplicado em Replicação Máquina de Estados*

Kayel L. Serafim[1], **Eduardo Alchieri**[2], **Fernando Dotti**[1]

[1]Escola Politécnica – Pontifícia Universidade Católica do Rio Grande do Sul (PUCRS)
Caixa Postal 1429 – 90.619-900 – Porto Alegre – RS – Brasil

[2]Universidade de Brasília (UnB) – Brasília, DF – Brasil

Abstract. *State Machine Replication (SMR) is an important approach to providing highly availabile services. Due to its deterministic model, scaling throughput in SMR is a challenge. However, there is a lack of common workloads to evaluate SMR, which are representative to significant application classes. We identify important and common aspects in the context of online transaction processing and discuss the use of such workloads to evaluate SMR. Due to its acceptance, we propose the use of benchmark C from the Transaction Processing Performance Committee, called TPC-C, for SMR evaluation. We discuss its architecture for the context of SMR and implemented it on a replication platform. Results for the sequential SMR model are reported. Moreover, a parallel SMR approach is discussed for TPC-C, implemented and results reported.*

Resumo. *A Replicação Máquina de Estados (RME) é uma abordagem importante para prover serviços de alta disponibilidade. No entanto, o aumento da vazão em RME é um desafio devido ao seu modelo determinístico. Uma lacuna importante é a falta de cargas de trabalho que permitam avaliar diferentes mecanismos para RME sob os mesmos referenciais, representativos de aplicações de interesse. Identificaram-se aspectos comuns relevantes no contexto de transações online, e propõe-se o uso do benchmark TPC-C do Transaction Processing Performance Comittee para avaliar RMEs. A arquitetura é discutida e implementada em uma plataforma de replicação, no contexto de RME. Os resultados obtidos foram reportados utilizando o modelo clássico de RME. Além disso, foi discutida e implementada uma abordagem para RME paralela com essa carga de trabalho, e os resultados também foram reportados.*

1. Introdução

A replicação da máquina de estado (RME) [Lamport 1978, Schneider 1990] é uma abordagem simples e eficaz para criar serviços altamente disponíveis que oferecem linearizabilidade [Herlihy and Wing 1990]. Na abordagem de RME, um conjunto de réplicas iniciam no mesmo estado e executam comandos de clientes na mesma ordem, deterministicamente, mantendo o estado replicado e consistente.

A proposta seminal de RME 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.

*Apoio: Conselho Nacional de Desenvolvimento Científico e Tecnológico - CNPQ; Fundação de Amparo à Pesquisa do Estado Do Rio Grande do Sul - FAPERGS PqG 07/21.

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 2.2; 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 [Bezerra et al. 2014, Li et al. 2017, Schiper et al. 2010] [J. C. Corbett and et al 2012, Cowling and Liskov 2012, Thomson et al. 2012].

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 on-line: 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 on-line 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 on-line (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.

Resumidamente, este artigo apresenta as seguintes contribuições:

i. Propõe o uso de cargas de trabalho do contexto de processamento de transações

para aplicação em RME, discutindo implicações do mapeamento de uma aplicação transacional sobre RME;
ii. Propõe uma arquitetura e uma implementação do TPC-C sobre uma plataforma de RME;
iii. Estende o resultado anterior para RME paralela através de uma definição de conflito entre transações, permitindo a execução paralela de transações independentes;
iv. Relata experimentos usando o TPC-C com o modelo clássico de RME e com extensões para RME paralela, obtidos através da implementação destas propostas na biblioteca de replicação BFT-SMaRt.

O restante do artigo está organizado da seguinte forma. A Seção 2 fornece a fundamentação teórica. A Seção 3 discute sobre o suporte a transações no modelo RME, enquanto que a Seção 4 descreve como os principais elementos do TPC-C são mapeados para um serviço replicado. Na seção 5, apresenta-se a definição da função de conflito para a integração do TPC-C em uma RME paralela. A Seção 6 descreve os experimentos e resultados alcançados. Finalmente, a Seção 7 apresenta as observações finais do artigo.

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) [Lamport 1978, Schneider 1990] 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 através de um protocolo de *broadcast* atômico [Hadzilacos and Toueg 1994], todas as réplicas entregam e executam 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 [Herlihy and Wing 1990, Attiya and Welch 2004].

A abordagem RME é empregada em muitos sistemas importantes. Por exemplo, na pilha de software do Google com o Chubby [Burrows 2006], que é usado pelo gerenciador de cluster Borg, pelo Google File System (GFS) e pelo Bigtable [Verma et al. 2015, Chang et al. 2008]. Analogamente, o Zookeeper do Apache [Hunt et al. 2010] é usado pelo HDFS [Shvachko et al. 2010] e pelo Cassandra [Lakshman and Malik 2010]. No entanto, a abordagem seminal RME tem desempenho limitado, pois as réplicas executam os comandos sequencialmente para garantir o determinismo e a 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 primeiro pois este artigo faz uso destas técnicas.

2.2. Replicação de Máquina de Estado 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 [Schneider 1990]. 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 [Kapritsos et al. 2012, Kotla and Dahlin 2004, Marandi et al. 2014, Marandi and Pedone 2014, Mendizabal et al. 2017, Alchieri et al. 2018, Escobar et al. 2019, Burgos et al. 2022, Batista et al. 2022]. 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.

2.3. TPC Benchmark C

O *Transaction Processing Performance Council*[1] (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[2] especifica o *benchmark* C deste comitê, tendo como alvo sistemas complexos de processamento de transações on-line (OLTP) e simula suas atividades com uma mistura de transações de somente leitura e de atualização intensiva. A carga de trabalho foi concebida no contexto de um fornecedor atacadista, centrada na atividade de processamento de pedidos e fornecendo um *design* de banco de dados lógico. A empresa está organizada em vários armazéns distribuídos. Cada armazém cobre 10 distritos e cada distrito atende a 3.000 clientes. Cada armazém possui estoques para os 100.000 itens vendidos pela empresa.

O banco de dados do TPC-C consiste em 9 tabelas definidas usando a abordagem entidade-relacionamento, sujeitas a uma série de regras de implementação: WAREHOUSE, DISTRICT, CUSTOMER, HISTORY, ORDER, NEW_ORDER, ORDER_LINE, STOCK_LEVEL. As seguintes transações são definidas:

NEW-ORDER: A transação NEW-ORDER insere um pedido completo no banco de dados. Representa uma transação de leitura e gravação de peso médio de alta frequência, sendo uma das principais transações da carga de trabalho.

PAYMENT: Esta transação atualiza o saldo do cliente e reflete o pagamento nas vendas do distrito e armazém. É também uma transação de leitura e gravação de alta frequência, mas leve.

[1] `https://www.tpc.org/`

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

DELIVERY: Esta é uma transação de leitura e gravação de baixa frequência que processa um lote de 10 novos pedidos, ainda não processados. Cada pedido é totalmente processado (entregue) dentro do escopo de uma transação de banco de dados de leitura/gravação.

ORDER-STATUS: ORDER-STATUS é uma transação somente leitura de peso médio e baixa frequência, que consulta o status do último pedido de um determinado cliente.

STOCK-LEVEL: É uma transação pesada de somente de leitura e com baixa frequência, que contabiliza o número de itens vendidos recentemente com nível de estoque abaixo de um determinado limite.

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. BFT-SMaRt

O BFT-SMaRt [Bessani et al. 2014] é uma biblioteca Java para RME, que implementa um protocolo inspirado em PBFT (*Practical Byzantine Fault Tolerance)* [Castro and Liskov 2002] para tolerar falhas bizantinas (BFT). No entanto, o sistema também pode ser configurado para tolerar apenas falhas 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) [Bessani et al. 2014]. 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[3] implementa o modelo sequencial de execução nas réplicas. Porém, no topo do BFT-SMaRt foram implementadas extensões para execução paralela[4] e particionada. Essas extensões suportam diferentes abordagens de execução discutidas em [Pedone et al. 2018].

3. 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 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.

[3] `https://github.com/bft-smart/library`

[4] `https://github.com/parallel-SMR/library`

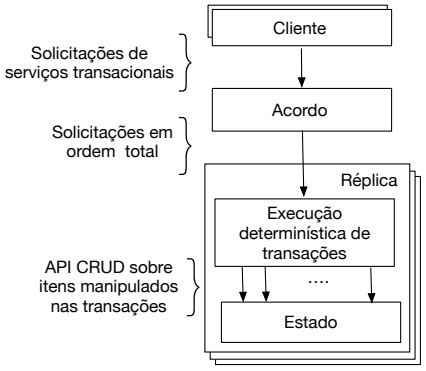


Figura 1. Serviço transacional sobre RME

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.

4. 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[5] de código aberto e a produzida na tese de [Le 2020].

4.1. 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 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 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.

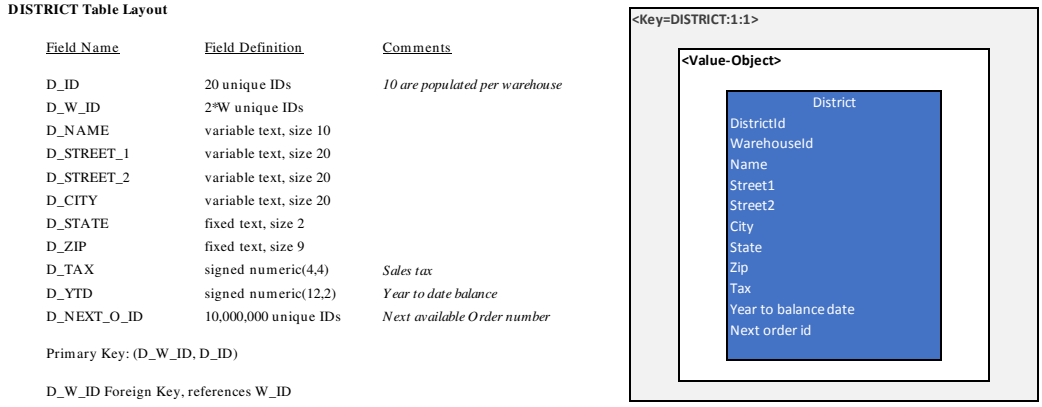


Figura 2. Layout (esquerda) e KV (direita) da tabela DISTRICT.

Em geral, qualquer tabela no modelo relacional pode ser representada 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

[5] `http://jtpcc.sourceforge.net/`

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.

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.

4.2. 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 atribui distribuições de probabilidade a todas as variáveis possíveis, como: tipo de transação; identificação do cliente por id ou sobrenome; escolha do cliente; itens de estoque, entre outros. Os clientes seguem essas distribuições enquanto constroem as transações a serem enviadas. Além disso, permitem que qualquer variável mencionada seja ajustada para outras distribuições. Esse recurso é usado para gerar cargas de trabalho livres de conflitos ou propensas a conflitos.

Depois que os lados do cliente e do servidor do TPC-C são definidos, todas as peças para implantar uma RME sequencial estão disponíveis no BFT-SMaRt. No entanto, isso não é suficiente para o caso paralelo.

5. Integração do TPC-C em uma RME Paralela

Dentre as várias estratégias para escalonar comandos em uma RME paralela [Pedone et al. 2018], foi escolhido neste estudo o escalonar tardio [Escobar et al. 2019] 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.

5.1. 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 1 apresenta as especificações da função de conflito. Devido à simetria, a tabela apresenta apenas a parte inferior da matriz.

	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. ✓

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

Tabela 1. Definição da função de conflito para o TPC-C.

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 atualizam o número sequencial do pedido D_NEXT_O_ID presente na tabela DISTRICT. Adicionalmente, estas transações também atualizam os itens de estoque, podendo gerar conflitos.
2. **NEW-ORDER e PAYMENT:** há conflito, pois ambas atualizam a tabela DISTRICT, a transação NEW-ORDER incrementa e atualiza o próximo número da ordem do pedido D_NEXT_ORDER_ID, enquanto a PAYMENT atualiza as informações de saldo D_YTD.
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, entre outros.

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 várias colunas da tabela CUSTOMER.
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 no DELIVERY, que são consultados no 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.
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.

6. Experientos e Resultados

Tendo desenvolvido o *benchmark* TPC-C para funcionar tanto sequencialmente quanto em paralelo, foi realizado uma série de experimentos para avaliar o comportamento destas abordagens utilizando uma carga de trabalho representativa de aplicações reais.

6.1. 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).

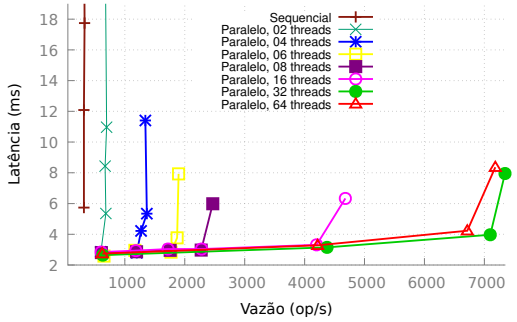
Aspectos metodológicos. O banco de dados foi pré-populado. Os experimentos consideram apenas um armazém. De acordo com o TPC-C, um armazém possui 10 distritos. Cada distrito tem 3.000 clientes e 100.000 itens de estoque. Cada cliente tem inicialmente um pedido em andamento. Cada configuração de *worker 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.

6.2. 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
Thr. 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
Thr. 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 da tabela: ktxns – k de *kilo* sendo x 1000; txns sendo transações.

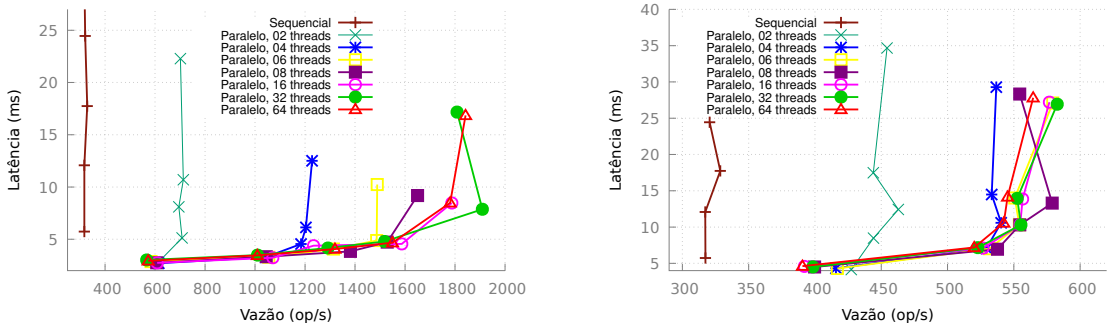
Figura 3. Cenário livre de conflitos (50% ORDER-STATUS e 50% STOCK-LEVEL).

A Figura 3 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(a), 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.

O segundo cenário introduz as transações de novos pedidos e entregas, levando a 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(b)).

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 5 apresenta os resultados para este cenário, onde a taxa de conflito aumentou



(a) Conflito medido de 5 a 8%

(b) Conflito medido de 18 a 24%

Figura 4. Cenários com diferentes taxas de conflitos.

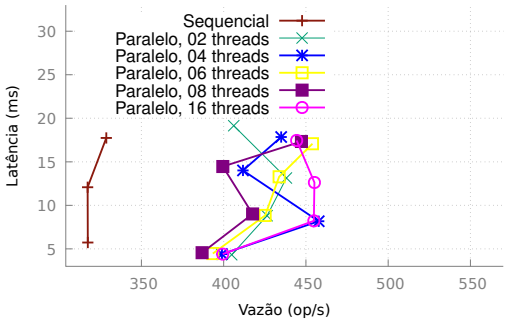


Figura 5. Carga de trabalho padrão do TPC-C.

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 aumentada (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%. 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%.

7. 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 para isso. 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.

As implementações do TPC-C para a plataforma BFT-SMaRt, bem como dos experimentos aqui reportados, estão disponíveis em `https://github.com/kayelserafim/bft-smart-tpcc`.

7.1. 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

Alchieri, E., Dotti, F., and Pedone, F. (2018). Early scheduling in parallel state machine replica. In *ACM SoCC*.

Attiya, H. and Welch, J. (2004). *Distributed Computing: Fundamentals, Simulations, and Advanced Topics*. Wiley-Interscience.

Batista, E., Alchieri, E., Dotti, F., and Pedone, F. (2022). Early scheduling on steroids: Boosting parallel state machine replication. *Journal of Parallel and Distributed Computing*, 163:269–282.

Bessani, A., Sousa, J. a., and Alchieri, E. E. P. (2014). State machine replication for the masses with bft-smart. In *Proceedings of the 2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, DSN '14, pages 355–362, Washington, DC, USA. IEEE Computer Society.

Bezerra, C. E., Pedone, F., and Renesse, R. V. (2014). Scalable state-machine replication. In *DSN*, pages 331–342.

Burgos, A., Alchieri, E., Dotti, F., and Pedone, F. (2022). Exploiting concurrency in sharded parallel state machine replication. *IEEE Transactions on Parallel and Distributed Systems*, 33(9):2133–2147.

Burrows, M. (2006). The chubby lock service for loosely-coupled distributed systems. In *OSDI*.

Castro, M. and Liskov, B. (2002). Practical byzantine fault-tolerance and proactive recovery. *ACM Transactions on Computer Systems*, 20(4):398–461.

Chang, F., Dean, J., Ghemawat, S., Hsieh, W. C., Wallach, D. A., Burrows, M., Chandra, T., Fikes, A., and Gruber, R. E. (2008). Bigtable: A distributed storage system for structured data. *ACM Trans. Comput. Syst.*, 26(2):1–26.

Cowling, J. and Liskov, B. (2012). Granola: Low-Overhead distributed transaction coordination. In *USENIX Annual Technical Conference*. USENIX Association.

Escobar, I., Alchieri, E., Dotti, F., and Pedone, F. (2019). Boosting concurrency in parallel state machine replication. In *Middleware*.

Hadzilacos, V. and Toueg, S. (1994). A modular approach to the specification and implementation of fault-tolerant broadcasts. Technical report, Department of Computer Science, Cornell.

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

Hunt, P., Konar, M., Junqueira, F. P., and Reed, B. (2010). Zookeeper: wait-free coordination for internet-scale systems. In *ATC*, volume 8.

J. C. Corbett, J. D. and et al, M. E. (2012). Spanner: Google's globally distributed database. In *OSDI*.

Kapritsos, M., Wang, Y., Quema, V., Clement, A., Alvisi, L., and Dahlin, M. (2012). All about eve: execute-verify replication for multi-core servers. In *OSDI*.

Kotla, R. and Dahlin, M. (2004). High throughput byzantine fault tolerance. In *DSN*.

Lakshman, A. and Malik, P. (2010). Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review*, 44(2):35–40.

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

Le, L. H. (2020). *Scaling State Machine Replication*. PhD thesis, Università della Svizzera italiana.

Li, Z., Van Roy, P., and Romano, P. (2017). Enhancing throughput of partially replicated state machines via multi-partition operation scheduling. In *2017 IEEE 16th International Symposium on Network Computing and Applications (NCA)*, pages 1–10.

Marandi, P. J., Bezerra, C. E. B., and Pedone, F. (2014). Rethinking state-machine replication for parallelism. In *ICDCS*.

Marandi, P. J. and Pedone, F. (2014). Optimistic parallel state-machine replication. In *SRDS*.

Mendizabal, O. M., Moura, R. T. S., Dotti, F. L., and Pedone, F. (2017). Efficient and deterministic scheduling for parallel state machine replication. In *IPDPS*.

Pedone, F., Alchieri, E., Dotti, F., Marandi, P., and Mendizabal, O. (2018). Boosting state machine replication with parallel execution. In *Anais do VIII Latin-American Symposium on Dependable Computing*, pages 77–86, Porto Alegre, RS, Brasil. SBC.

Schiper, N., Sutra, P., and Pedone, F. (2010). P-store: Genuine partial replication in wide area networks. In *Symposium on Reliable Distributed Systems (SRDS)*.

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

Shvachko, K., Kuang, H., Radia, S., and Chansler, R. (2010). The hadoop distributed file system. In *MSST*.

Thomson, A., Diamond, T., Weng, S.-C., Ren, K., Shao, P., and Abadi, D. J. (2012). Calvin: Fast distributed transactions for partitioned database systems. In *Proceedings of ACM SIGMOD International Conference on Management of Data*, SIGMOD '12.

Verma, A., Pedrosa, L., Korupolu, M., Oppenheimer, D., Tune, E., and Wilkes, J. (2015). Large-scale cluster management at google with borg. In *EuroSys*.