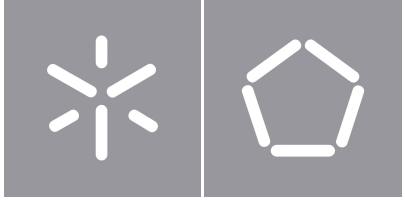
Universidade do Minho
Escola de Engenharia

Paulo Silva Sousa

Simulação de sistemas distribuídos de gestão de bases de dados

agosto 2023



Universidade do Minho
Escola de Engenharia

Paulo Silva Sousa

Simulação de sistemas distribuídos de gestão de bases de dados

Dissertação de Mestrado
Mestrado em Engenharia Informática

Sistemas Distribuídos
Trabalho efetuado sob a orientação de
José Orlando Roque Nascimento Pereira
Ricardo Manuel Pereira Vilaça

agosto 2023

Direitos de Autor e Condições de Utilização do Trabalho por Terceiros

Este é um trabalho académico que pode ser utilizado por terceiros desde que respeitadas as regras e boas práticas internacionalmente aceites, no que concerne aos direitos de autor e direitos conexos.

Assim, o presente trabalho pode ser utilizado nos termos previstos na licença abaixo indicada.

Caso o utilizador necessite de permissão para poder fazer um uso do trabalho em condições não previstas no licenciamento indicado, deverá contactar o autor, através do RepositóriUM da Universidade do Minho.

Licença concedida aos utilizadores deste trabalho:

CC BY

https://creativecommons.org/licenses/by/4.0/

Agradecimentos

A realização desta dissertação não teria sido possível sem o apoio importante de diversas pessoas.

Aos meus orientadores, professor José Orlando Pereira e Ricardo Vilaça, agradeço por todo o acompanhamento que me deram durante este ano. A vossa orientação foi essencial para conseguir concluir a dissertação.

Aos meus pais e ao meu irmão agradeço por me terem apoiado ao longo de todo o meu percurso escolar e académico.

Aos meus amigos agradeço por sempre me apoiaram nos momentos mais difíceis que fui passando ao longo do curso. Agradeço também por todos os momentos que passamos ao longo destes 5 anos, ficarão lembranças eternas.

Declaração de Integridade

Declaro ter atuado com integridade na elaboração do presente trabalho académico e confirmo que não recorri à prática de plágio nem a qualquer forma de utilização indevida ou falsificação de informações ou resultados em nenhuma das etapas conducente à sua elaboração.

Mais declaro que conheço e que respeitei o Código de Conduta Ética da Universidade do Minho.

Universidade do Minho, Braga, agosto 2023

Paulo Silva Sousa

Assinado por: **Paulo Silva Sousa**
Num. de Identificação: 14354810
Data: 2023.08.07 17:33:59+01'00'

Resumo

Hoje em dia, graças à existência de várias aplicações em grande escala com acesso a grandes quantidades de informação, bases de dados monolíticas não são capazes de satisfazer as suas necessidades, quer a nível de disponibilidade, de escalabilidade ou de performance. Deste modo, necessitamos de sistemas distribuídos de gestão de bases de dados para conseguir satisfazer estas aplicações. Destes sistemas, são particularmente interessantes aqueles que se destinam a um grande número de servidores espalhados por diferentes zonas geográficas, devido à urgência de os aproximar das populações para obter uma melhor escalabilidade do sistema e uma melhor performance. Estes sistemas estão geralmente divididos em duas famílias: uma que dá prioridade à coerência dos dados e uma que dá prioridade à disponibilidade do serviço.

Apesar do interesse que estes sistemas despertam, existe um grande custo associado ao seu teste no mundo real, sendo necessário recorrer a modelos de simulação para reproduzir o seu comportamento. Além disso, estes sistemas contém bastantes diferenças entre eles, sendo muitas vezes difícil de comparar as suas vantagens e desvantagens em contexto real.

Nesta tese desenvolvemos o *SAGeo*, um simulador de bases de dados geo-replicadas configurável, capaz de avaliar e comparar o desempenho relativo de diversas bases de dados distribuídas. Para além disso, configuramos este simulador para três algoritmos de bases de dados diferentes e apresentamos comparações de resultados de diversas simulações realizadas.

Palavras-chave Base de Dados, Sistemas Distribuídos, Base de Dados Distribuída Geo-replicada, Simulação, Simulação de Eventos Discretos

Abstract

Now a days, thanks to the existence of various large scale applications with access to large amounts of information, monolithic databases can not satisfy their needs in terms of availability, scalability and performance. In that way, distribuited database systems are needed to solve those application issues. From those systems, those who have a large number of servers in different geographic locations are particularly intersting, due to the urge to get them closer to the populations in order to obtain a better scalability and a better performance. These systems are usually divided in two families: one that prioritizes data consistency and one that prioritizes service availability.

Despite the interest that these systems arouse, there is a large temporal and monetary cost associated with testing them in the real world, and it is necessary to resort to simulation models to reproduce their behavior. Futhermore, these systems are very different from each other, making it often difficult to compare their advantages and disadvantages in a real context.

In this thesis we developed *SAGeo*, a configurable simulator for geo-replicated databases, capable of evaluating and comparing the relative performance of different distributed databases. Moreover, we configured these simulator to three different database algoritms and presented comparisons between several simulation results.

Keywords Database, Distributed Systems, Distributed Geo-replicated Database, Simution, Discrete Event Simulation

Índice

Índice de Figuras

Índice de Tabelas

Índice de Código Fonte

Acrónimos

AP Available and Partition Tolerant System (Sistemas de Disponibilidade e Tolerância a Partições).

API Application Programming Interface (Interface de Programação da Aplicação).

CA Consistent and Available System (Sistemas de Disponibilidade e Corência).

COPS Clusters of Order-Preserving Servers (Grupo de Servidores Preservantes de Ordem).

COPS-GT Clusters of Order-Preserving Servers - Get Transaction (Grupo de Servidores Preservantes de Ordem - Transação *Get*).

CP Consistent and Partition Tolerant System (Sistemas de Corência e Tolerância a Partições).

CPU Central Processing Unit (Unidade de Processamento Central).

DBSM Database State Machine (Máquina de Estados de Base de Dados).

DES Discrete Event Simulation (Simulação de Eventos Discretos).

PDBSM Partial Database State Machine (Máquina de Estados de Base de Dados Parcial).

PDES Parallel Discrete Event Simulation (Simulação Paralela de Eventos Discretos).

RAM Random Access Memory (Memória de Acesso aleatório).

SGBDD Sistema de Gestão de Bases de Dados Distribuídas.

SQL Structured Query Language (Linguagem de Consulta Estruturada).

Glossário

Atomic Broadcast Transmissão em que todos os processos que não falham num sistema de vários processos recebem o mesmo conjunto de mensagens na mesma ordem.

fragmentação Processo de particionar a base de dados horizontalmente em várias bases de dados mais pequenas.

gargalo Parte do sistema não tem capacidade para processar a quantidade de dados pretendida, afetando a performance total do sistema.

locking Mecanismo para garantir apenas um processo pode aceder a uma parte dos dados de cada vez, de modo a garantir a exatidão do sistema [Kle16].

replicação Processo de replicar dados em diversas bases de dados.

Capítulo 1

Introdução

1.1 Contexto e Motivação

Com a evolução da tecnologia, cada vez mais geramos e consumimos grandes quantidades de dados em diversas atividades do nosso dia a dia, sejam elas relacionadas com o trabalho ou entretenimento. Para guardar e organizar todos estes dados dependemos de bases de dados. Todas as bases de dados têm o mesmo objetivo: organizar uma coleção de dados de forma a que estes possam ser facilmente acedidos e manipulados. Assim, os utilizadores conseguem recolher, adicionar, atualizar e eliminar dados de uma forma fácil e eficiente [RGG03].

Com o crescimento do número de aplicações em grande escala a quantidade de dados que necessitam de ser armazenados nestas bases de dados, bem como o número de clientes que realizam pedidos a estas, aumentou exponencialmente. Deste modo, o aumento da taxa de chegada de pedidos irá levar, eventualmente, a um esgotamento dos recursos disponíveis por uma base de dados centralizada e a uma degradação do desempenho da mesma. Uma solução para este problema é escalar a base de dados verticalmente, adicionando mais recursos ao servidor até satisfazer o desempenho desejado. Contudo, esta solução é apenas temporária, uma vez que existe um limite para a quantidade de recursos que podem ser adicionados. Além disso, as aplicações desenvolvidas atualmente exigem um serviço altamente disponível para poderem corresponder com as necessidades do mercado e bases de dados centralizadas não toleram faltas. Surge então a necessidade de utilizar bases de dados distribuídas, que são conjuntos de servidores repartidos numa rede de computadores, que se relacionam entre si, de modo a melhorar o desempenho, a disponibilidade e a escalabilidade do serviço [V96, RGG03].

Uma das grandes vantagens da utilização de uma base de dados distribuída é o aumento da capacidade para tratar elevadas quantidades de dados, pois se o sistema estiver a atingir um **gargalo** estas são escaláveis horizontalmente, sendo apenas necessário adicionar novos servidores à rede. Os dados são depois distribuídos por todos os servidores recorrendo a métodos como **fragmentação** e **replica-**

ção. Outra vantagem da utilização destas bases de dados, caso haja **replicação**, é a capacidade de tolerar faltas, onde no caso de uma falta de um servidor, esta consegue continuar a execução e servir o utilizador sem interrupção, pois os pedidos deste poderão continuar a ser atendidos por outros servidores do sistema. Por último, no caso de bases de dados distribuídas geo-replicadas, conseguimos ter um grande número de servidores espalhados por diferentes zonas geográficas, aproximando-os das populações para reduzir a latência dos pedidos dos clientes à base de dados, aumentando a localidade geográfica e melhorar a disponibilidade [SPAL11].

Estes sistemas de bases de dados geo-replicados estão geralmente divididos em duas famílias de acordo com o comportamento na presença de partições da rede [GL02]: uma primeiro que dá prioridade à disponibilidade dos dados perante partições e que oferece menor latência de resposta a pedidos, e uma segunda que dá prioridade à coerência de dados.

Estes algoritmos servem dois propósitos diferentes e seguem abordagens de implementação muito distintas. Além disso, mesmo dentro da mesma família de algoritmos, existe um vasto leque de implementações diferentes devido à constante evolução dos mesmos. Devido à grande variedade de abordagens é muitas vezes difícil perceber as diferenças entre eles e quantificar as vantagens e desvantagens relativas, sendo necessária uma ferramenta capaz de os comparar.

Normalmente, um protótipo de um sistema distribuído de base de dados que é proposto é testado usando um conjunto de servidores dispersos geograficamente num sistema de computação em nuvem, incluindo uma comparação pontual com um sistema antecessor. No entanto, apesar deste método ser útil para demonstrar a viabilidade de uma proposta no mundo real, tem várias limitações.

1.2 Problema

Primeiramente, a escala a que os testes são realizados em termos de número de servidores é limitada, devido ao custo da infraestrutura necessária para os suportar, impedindo assim a avaliação de condições relevantes [LSS+17]. Além disso, a implementação em nuvem não dispõe de implementações de múltiplos sistemas de modo a comparar diretamente os compromissos de cada um. Por último, a reprodução de experiências na nuvem é difícil de atingir [LAdS+15, SDQR10]. Surge assim a necessidade de modelar um sistema de simulação capaz de ultrapassar as dificuldades da implementação em nuvem.

Simulação de sistemas de bases de dados é o processo de criar um modelo computacional aproximado do *design* do modelo verdadeiro, de modo a simular o seu desempenho e escalabilidade. Pode ser também utilizado para encontrar algum **gargalo** e ajudar a resolvê-lo. Este modelo pode depois ser

executado com uma grande variedade de parâmetros, cobrindo muitos casos diferentes a que a base de dados se possa encontrar, tornando esta o mais robusta possível para poder estar num ambiente de produção [Law15]. Além disso, com um modelo de simulação podemos testar casos hipotéticos de um sistema, que no mundo real seriam demasiado custosos para analisar, como por exemplo aumentar consideravelmente o número de nós de um sistema.

Existem diversos simuladores de bases de dados distribuídas, contudo, apenas permitem avaliar o desempenho de um algoritmo em particular. Surge assim a necessidade de um simulador capaz de avaliar e comparar diferentes algoritmos de bases de dados distribuídas, de modo ser possível perceber as vantagens e limitações de cada um.

1.3 Objetivos e Resultados

Esta tese tem como objetivo a simulação de sistemas distribuídos de gestão de bases de dados para avaliar e comparar o desempenho relativo de diferentes bases de dados distribuídas perante variações nos parâmetros de simulação, assim como ajudar na validação dos algoritmos e encontrar potenciais problemas. Deste modo, o propósito do modelo não é tentar replicar resultados absolutos obtidos pelos sistemas em contexto real, mas sim analisar potenciais comportamentos relativos perante diferentes compromissos na simulação.

O modelo *SAGeo* foi desenvolvido utilizando o paradigma de simulação orientado ao processo, no qual o tempo em que não se está a fazer uma transação na base de dados será avançado, de modo a reduzir significativamente o tempo de simulação e obter o simulador o mais escalável possível.

Como primeiro resultado, configuramos o modelo para o algoritmo de bases de dados *Wren* [SDZ18] e utilizamo-lo para reproduzir os resultados encontrados no artigo original, onde é comparado o desempenho do algoritmo com e sem a utilização de *cache*.

Como segundo resultado, utilizamos o modelo configurado para os algoritmos de bases de dados **DBSM** [PGS03] e *Spanner* [CDE+13] para comparar o desempenho destes dois sistemas em relação à utilização de execução otimista por parte do **DBSM** e a utilização de fragmentação e geo-replicação por parte do *Spanner*.

Por último, como terceiro resultado, utilizamos o modelo configurado para os algoritmos *Wren* e *Spanner* para comparar o desempenho destes dois sistemas perante partições de rede e latências entre centros de dados crescentes.

O código fonte do modelo está disponível como código aberto: `https://github.com/29medium/`

SAGeo.git.

1.4 Estrutura da Tese

O Capítulo 2 aborda o estado da arte e faz uma revisão da literatura. Este capítulo está dividido em duas secções, uma primeira secção que trata de bases de dados distribuídas (2.1), especificamente as arquiteturas **DBSM**, *Spanner*, **COPS** e *Wren*; e uma segunda secção sobre simulação de um modo geral (2.2), simuladores e modelos de bases de dados distribuídas.

O Capítulo 3 aborda, numa primeira secção, o modelo de simulação de bases de dados genérico desenvolvido, bem como este pode ser configurado para diferentes algoritmos. Além disso, esta capítulo também aborda o método de introdução dos parâmetros de simulação e a extração de resultados. Além disso, numa segunda secção, aborda a implementação das arquiteturas **DBSM**, *Spanner* e *Wren* no nosso modelo de simulação configurável, onde é apresentada a arquitetura de cada uma das soluções.

O Capítulo 4 aborda, primeiramente, a validação do modelo de simulação. Além disso, são comparados os resultados obtidos com as diferentes arquiteturas de modo a expor as diferenças dos algoritmos.

Por último, o Capítulo 5 conclui a dissertação.

Capítulo 2

Estado da Arte e Revisão da Literatura

2.1 Sistemas de Gestão de Bases de Dados Distribuídas

Uma base de dados distribuída é um conjunto de bases de dados distribuídas numa rede, sendo estas logicamente relacionadas entre si, em vez de centralizadas num só computador. O seu objetivo é aumentar a disponibilidade e o desempenho, comparativamente a um sistema centralizado. Uma base de dados distribuída é gerida por um **Sistema de Gestão de Bases de Dados Distribuídas** (**SGBDD**). Já este é um sistema com uma interface comum para aceder aos dados [V96].

Existem várias vantagens na utilização de um sistema distribuído de bases de dados comparativamente a um sistema centralizado [RGG03]:

- **Disponibilidade -** Quando um nodo da rede falha, o sistema continua a poder ser acedido através de outros nodos.
- **Desempenho -** Ao termos a carga de trabalho distribuída por vários nodos, o número de pedidos que cada um recebe é menor do que num modelo com apenas um computador, aumentando o desempenho do sistema. Além disso, no caso de um sistema de bases de dados geo-replicado, os clientes poderão aceder um servidor mais perto da sua localização e reduzir o tempo de comunicação com este.
- **Escalabilidade -** Um sistema distribuído pode escalar com a adição de um nodo à rede.

Por outro lado, nem todos os aspetos de uma base de dados distribuída são positivos. Os dados de todo o sistema podem não ser coerentes num dado instante de tempo, sendo que diferentes computadores são atualizados em tempos diferentes. Isto pode levar a problemas de integridade dos dados, levando-nos a fazer escolhas sobre a arquitetura da nossa base de dados, em que terão de ser feitos compromissos sobre algumas das funcionalidades, como por exemplo em relação à disponibilidade e coerência, ou em relação à utilização de **locking** otimista ou conservador.

Na Figura 1 e na Figura 2 temos o exemplo de uma arquitetura centralizada e de uma arquitetura distribuída, respetivamente. Na arquitetura centralizada, os pedidos do cliente são atendidos por um servidor apenas e os dados são guardados apenas numa base de dados. Já na arquitetura distribuída, os pedidos dos clientes podem ser atendidos por qualquer um dos vários servidores disponíveis, de modo a distribuir a carga de pedidos, e os dados são replicados por diversas bases de dados.

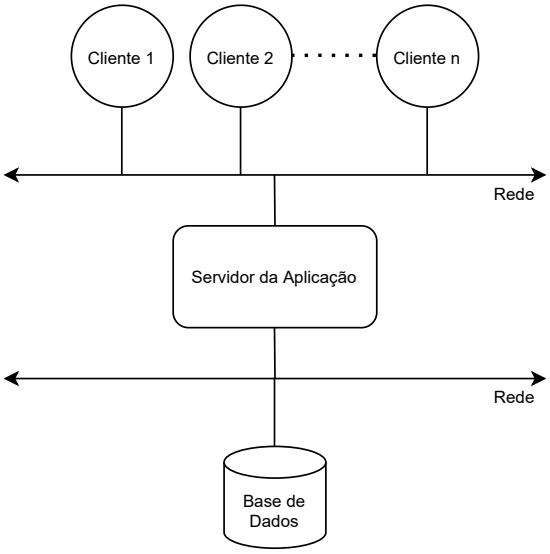


Figura 1: Arquitetura Centralizada [V11]

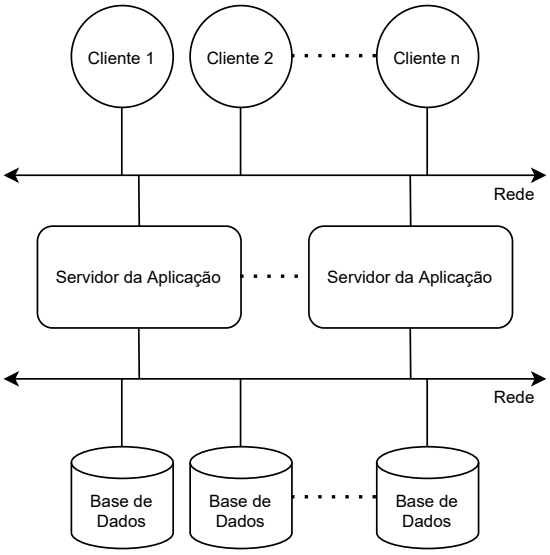


Figura 2: Arquitetura Distribuída [V11]

2.1.1 Teorema CAP

O Teorema *CAP* foi introduzido por *Eric Brewer* em 1999, no contexto de um serviço web distribuído, e indica que esse serviço tem de negociar entre coerência, disponibilidade e tolerância a partições da rede [GL12]:

- **Coerência -** Coerência é a propriedade de cada servidor devolver a resposta correta a cada pedido.
- **Disponibilidade -** Disponibilidade é a propriedade de cada pedido feito por um cliente ao sistema receber uma resposta, sem garantia de que contém a escrita mais recente.
- **Tolerância a partições -** Tolerância a partições da rede é a capacidade de um sistema continuar operacional mesmo ocorrendo uma falha de comunicação entre dois ou mais nodos desse mesmo sistema [Edu23].

Inicialmente, quando o teorema foi desenvolvido, existia a fórmula "2 de 3", em que apenas 2 das 3 propriedades descritas em cima poderiam ser satisfeitas, como podemos observar na Figura 3. No entanto, como a maioria dos sistemas não podia sacrificar tolerância a partições, como nos sistemas **CA**, a negociação decorria entre sacrificar coerência ou disponibilidade [Bre12].

Figura 3: Teorema CAP [Zha20]

Por exemplo, imaginemos um sistema distribuído que representa um banco, com dois servidores de bases de dados, cada um responsável por uma caixa multibanco. Se existir uma partição de rede em que as duas caixas multibanco não podem comunicar uma com a outra: se descartarmos coerência, o serviço irá estar disponível, mas operações que ocorrem durante a partição podem levar a dados incoerentes quanto esta estiver resolvida; se descartarmos disponibilidade, o serviço não estará disponível durante a partição, mas é garantido que quando esta for resolvida os dados estão coerentes.

Contudo, com o passar dos anos, após terem sido desenvolvidos muitos sistemas baseados neste teorema, concluiu-se que a fórmula "2 de 3"é enganadora, visto que, apesar de não ser possível ter coerência e disponibilidade perfeita sendo tolerante a partições, os sistemas de hoje em dia são desenvolvidos de modo a tentar maximizar os dois [Bre12].

Seguindo o exemplo anterior do banco, o sistema pode sacrificar alguma coerência e alguma disponibilidade, mas tentar chegar a uma solução melhor que a anterior. Por exemplo, o sistema pode permitir depósitos e consultas de saldo, mas não permitir levantamentos. Assim, disponibilidade de alguns serviços é garantida, bem como alguma coerência.

Surgem assim duas famílias de algoritmos de sistemas distribuídos de bases de dados de acordo com o comportamento na presença de partições de rede [GL02]: uma primeira que dá prioridade à disponibilidade dos dados perante partições de rede e que oferece menor latência de resposta a pedidos (sistema **AP**), e uma segunda que dá prioridade à coerência de dados (sistema **CP**).

Em particular, em relação aos sistemas geo-replicados que privilegiam a disponibilidade, existe uma grande variedade de algoritmos que foram apresentados ao longo dos anos que apresentam diversas evoluções. Um dos primeiros sistemas a ser apresentados com estas características foi o *Dynamo* da *Amazon* [DHJ+07], um sistema replicado e particionado com coerência eventual. Esta proposta foi melhorada com a introdução de coerência causal [LFKA11] e transações [TZB+17]. Estas caraterísticas foram também combinadas entre si [ATB+16] e otimizadas para permitir **replicação** parcial, em que nem todos os itens de dados estão presentes em todos os centros de dados [BRVR17, SDZ19].

Por outro lado, existe também uma grande variedade de propostas de sistemas que privilegiam a coerência. Os sistema *Spanner* [CDE+13] e *CockroachDB* [TSM+20] oferecem *serializability* em múltiplos centros de dados e com **replicação** parcial, no primeiro caso, usando relógios sincronizados e no segundo caso recorrendo a um protocolo de consenso. Existem também alternativas que suportam isolamento mais fraco no sentido de melhorar o desempenho [DEZ13, SPAL11].

2.1.2 Estratégias de replicação de Bases de Dados

Replicação de base de dados é a estratégia referente a replicar bases de dados em vários servidores de modo a atingir um melhor desempenho do sistema e uma melhor tolerância a partições. É possível caracterizar esta **replicação** em relação a quando esta acontece e em relação a onde esta acontece. [GHOS96a].

No que toca ao instante em que a propagação da transação entre os servidores acontece temos dois tipos de **replicação**, **replicação** síncrona (*Eager*) e **replicação** assíncrona (*Lazy*). Na **replicação** síncrona as atualizações são propagadas pelo servidor que recebeu o pedido do utilizador para as réplicas dentro dos limites da transação, ou seja, o utilizador não recebe a notificação de conclusão da transação até um número suficiente de réplicas aplicar a transação. Na **replicação** assíncrona o servidor que recebeu o pedido do utilizador aplica a transação e responde imediatamente ao utilizador, apenas propagando a atualização passado algum tempo [WPS+00]. Na Figura 4 podemos verificar a diferença da resposta síncrona e assíncrona ao utilizador.

A primeira solução é normalmente utilizada por sistemas **CP**, em que é dada prioridade à coerência dos dados, enquanto a segunda solução é utilizada por sistemas **AP**, onde é preferível tempos de resposta mais baixos e maior disponibilidade, em favor da coerência dos dados.

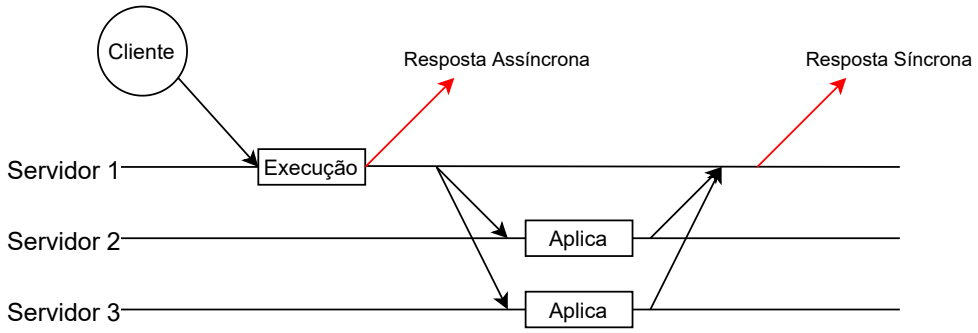


Figura 4: Estratégia síncrona e assíncrona

Além disso, no que toca ao local onde é permitido efetuar transações temos duas abordagens, *primary-copy* e *update-everywhere*. Na estratégia *primary-copy* uma única réplica é responsável por receber e executar todas as transações, simplificando a implementação mas apresentando um **gargalo** e ponto único de falha. Já na estratégia *update-everywhere* qualquer servidor pode executar transações, melhorando o desempenho do sistema mas tornando a implementação mais complexa, devido à coordenação necessária entre réplicas [WPS+00].

Das bases de dados que realizam **replicação** síncrona, recorrendo à estratégia *update-everywhere*,

podemos ainda classificar as bases de dados quanto ao método de execução das transações: execução conservadora e execução otimista. Na execução conservadora as transações são classificadas em classes de conflito e ordenadas (por um algoritmo de ordem total) antes da execução e, posteriormente, as transações conflituosas são executadas sequencialmente. Na execução otimista, as transações são ordenadas após a execução da transação, onde transações que acedem aos mesmo dados poderão ser executadas concorrentemente. Após a execução existe uma fase de certificação, onde a transação é dada como confirmada ou abortada [JPR+10].

2.1.3 Bases de Dados Distribuídas

DBSM

O sistema de base de dados **Database State Machine (Máquina de Estados de Base de Dados)** (**DBSM**) [PGS03] é um sistema **CP**, ou seja, tal como está descrito na secção Subseção 2.1.1, é um sistema que sacrifica a disponibilidade para dar prioridade à coerência de dados.

O objetivo deste sistema é conseguir superar as limitações de desempenho e escalabilidade de algoritmos tradicionais de **replicação** de bases de dados, como **locking** distribuído, preservando mesmo assim a *1-copy serializability*.

O algoritmo de **replicação** apresentado por este sistema é baseado na abordagem de **replicação** de máquinas de estado [Sch90a] (*deferred update*), em que transações são processadas localmente num servidor e posteriormente enviadas a todos os servidores para realizar certificação, onde estas poderão ser confirmadas ou abortadas. O envio destas mensagens é feito utilizando um protocolo de **Atomic Broadcast**, em que as várias réplicas do sistema acordam numa sequencia de mensagens a ser entregues, através de algum algoritmo de acordo distribuído. Este processo de envio de mensagens com ordem total com o objetivo de serem certificadas é chamado de protocolo de terminação e pode ser observado na Figura 5.

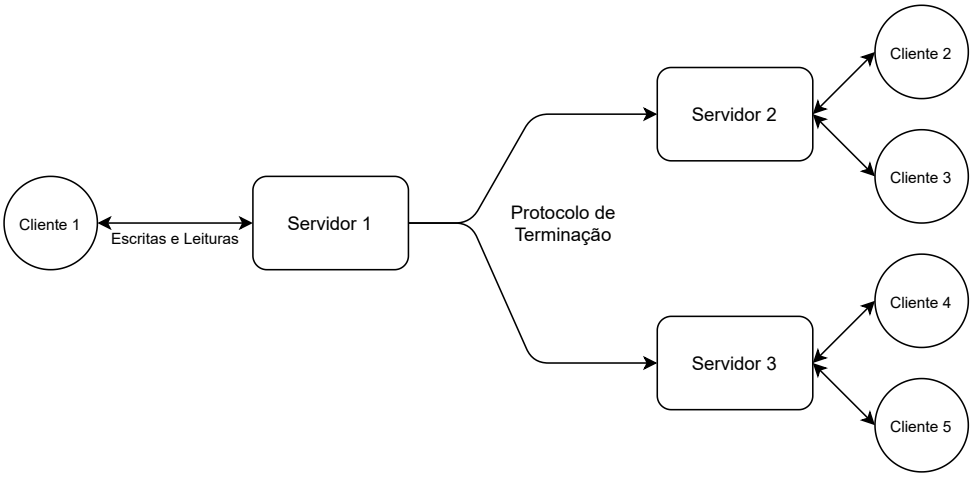


Figura 5: Estratégia *deferred update* [PGS03]

Para garantir a coerência dos dados, o servidor que processa o pedido envia, juntamente com o resultado da execução da transação, o conjunto de operações de leitura e conjunto de operações de escrita da transação. Posteriormente, no momento da certificação, cada servidor vai comparar o conjunto das operações da sua transação com os das transações concorrentes com esta, decidindo assim se esta é confirmada ou abortada. O objetivo da certificação é atingir *one-copy serializability*. Todos os servidores chegam ao mesmo resultado final porque a sequência de entrega das mensagens é garantido pelo **Atomic Broadcast**.

Contudo, apesar das suas vantagens, este sistema de bases de dados não é escalável e não apresenta o desempenho necessário para geo-replicação, devido à falta de **fragmentação**. Contudo, surgiram algoritmos baseados no **DBSM** que adicionam **fragmentação**, como por exemplo o **Partial Database State Machine (Máquina de Estados de Base de Dados Parcial)** (**PDBSM**), com adição de uma fase de coordenação posterior à certificação para garantir a coerência dos dados [SPMO01].

Spanner

Spanner é uma base de dados distribuída desenvolvida pela *Google* com o objetivo de ser altamente escalável e disponível globalmente, bem como disponibilizar localidade geográfica através de geo-replicação. Para isso, recorre a **fragmentação** e **replicação** dos dados por diversos centros de dados, localizados em vários continentes, organizados com máquinas de estado que utilizam uma variação do algoritmo de consenso distribuído de *Paxos* para acordar na ordem de entrega de mensagens [CDE+13]. Como garante coerência de dados, o **SGBDD** *Spanner* é então um sistema **CP**.

Neste sistema de base de dados, os dados são guardados acompanhados de uma versão, numa linguagem de consulta baseada em **SQL**. Cada versão é automaticamente gerada através da etiqueta

temporal da confirmação, podendo versões antigas ser lidas [CDE+13].

Comparativamente com outros sistemas de bases de dados distribuídas, o *Spanner* incluí diversas funcionalidades. Primeiramente, permite utilizar geo-replicação, em que o cliente que utiliza a base de dados tem opção de escolher os centros de dados, quantas réplicas quer manter para a sua aplicação e quão distantes estas réplicas estão entre si. Isto permite ao cliente controlar a latência de leituras e escritas, bem como a durabilidade, disponibilidade e desempenho da sua aplicação. Além disso, apesar de ser um sistema **CP**, o *Spanner* consegue garantir mesmo assim uma disponibilidade bastante elevada (aproximadamente 99.999%).

Estas funcionalidades apenas são possíveis por causa da **API** *TrueTime* desenvolvida pela *Google*. Esta **API** mantém uma incerteza de relógio abaixo dos 10 milissegundos através de *GPS* e relógios atómicos e expõe essa incerteza ao *Spanner*. Se a incerteza for alta, o *Spanner* abranda até acertar com os valores fornecidos por essa **API** [CDE+13].

Como podemos observar com a Figura 6, uma implementação do *Spanner* é chamado de universo, que por sua vez está organizado por zonas (que podem ser adicionadas e removidas). Cada zona tem um conjunto de *spanservers*, um *zonemaster* que distribui os dados por estes *spanservers* e vários *location proxies* que fornecem aos clientes o servidor a utilizar. Cada spanserver é responsável por um fragmento dos dados.

Tal como está demonstrado na Figura 7, a arquitetura de um *spanserver* é composta por um conjunto de *tablets*, que são estruturas de dados que guardam o fragmento de dados pelo qual o *spanserver* é responsável numa estrutura *map*: (chave, etiqueta temporal) -> valor. O estado de cada *tablet* está guardado num sistema de ficheiros distribuído chamado *Colossus* [CDE+13]. O fragmento de dados de um *spanserver* é replicado noutros *spanservers* de outras zonas (outros centros de dados) da base de dados. O conjunto de réplicas de diferentes zonas responsável pelo mesmo fragmento de dados é chamado um grupo, que comunica entre si por uma instância de *Paxos*, onde é eleito um líder do grupo. Caso uma transação envolva mais do que um fragmento de dados, os grupos dos diversos fragmentos elegem um líder participante (*participant leader*) que é responsável por efetuar a coordenação entre os líderes dos grupos e realizar a transação.

Neste sistema de base de dados, transações de escritas têm de ser feitas diretamente ao líder do grupo, mas transações de leitura podem ser feitas a qualquer *spanserver* que tenha a informação.

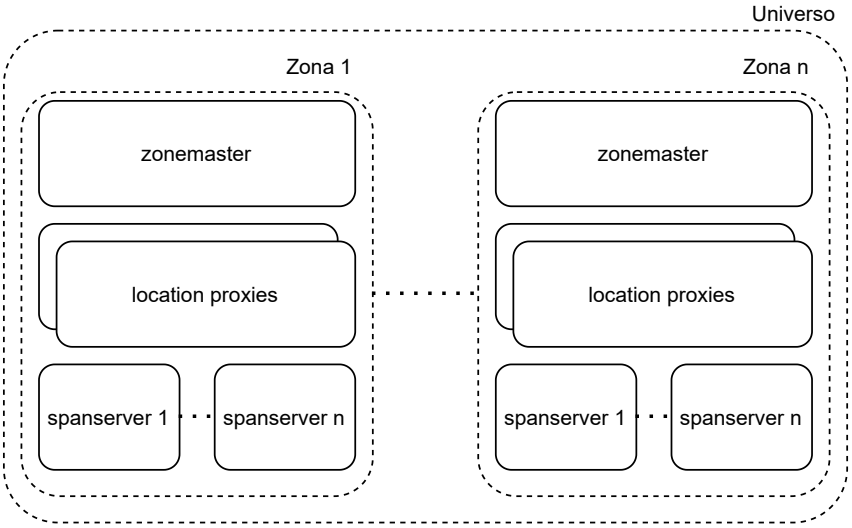


Figura 6: Universo de uma Implementação *Spanner* [CDE+13]

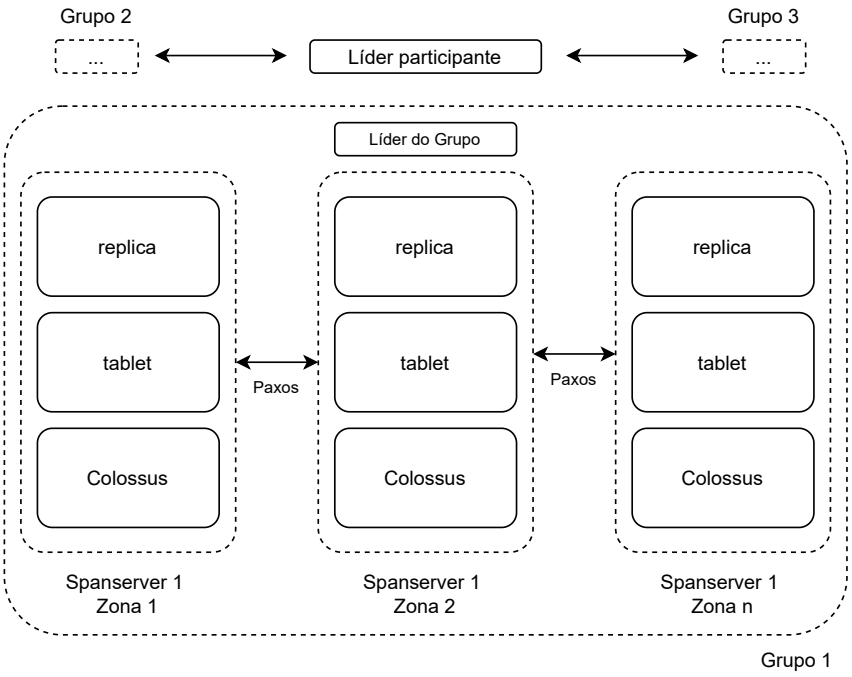


Figura 7: Arquitetura de um *spanserver* [CDE+13]

COPS

O objetivo do desenvolvimento do sistema de base de dados **Clusters of Order-Preserving Servers (Grupo de Servidores Preservantes de Ordem)** (**COPS**) foi implementar um sistema **AP** procurando alcançar a máxima coerência de dados possível: *coerência causal+* (coerência causal com mecanismos de convergência de conflitos) [LFKA11]:

- **Coerência causal -** A componente causal do modelo garante que a ordem de execução das operações respeita a sua ordem causal, ou seja, se A depende de B, então B é executado antes de A.

- **Convergência de conflitos -** A componente de convergência de conflitos garante que duas réplicas nunca divergem permanentemente e que conflitos para a mesma chave são tratados da mesma maneira em todas as réplicas.

Existem duas versões do sistema **COPS**. Uma versão normal, que fornece coerência causal+ entre dados individuais na base de dados distribuída e uma versão **COPS-GT** que permite ao cliente ver uma vista coerente de várias chaves (*get_trans*).

A arquitetura desta base de dados é composta por duas componentes principais, a biblioteca cliente e a estrutura chave-valor:

- **Estrutura chave-valor -** Esta componente é uma estrutura chave-valor simples com algumas alterações. Primeiramente, a cada par chave-valor estão associados meta-dados (no caso do **COPS** estes meta-dados são uma versão e no caso do **COPS-GT** são uma versão e uma lista de dependências (outras chaves e as suas versões)). Além disso, no caso do **COPS-GT**, são mantidas versões antigas das chaves de modo a conseguir realizar a operação *get_trans*. Por último, são exportadas três operações: *put_after*, para satisfazer operações de escrita; *get_by_vers*, para satisfazer operações de leitura; e *dep_check*, para verificar se todas as dependências de uma chave estão satisfeitas antes de executar a operação de escrita (apenas no **COPS-GT**).

- **Biblioteca Cliente -** Esta componente é responsável por exportar para o cliente a operação de leitura (*get* no caso de **COPS** ou *get_trans* no caso de **COPS-GT**) e a operação de escrita (*put*). Além disso, esta estrutura guarda no parâmetro *context* o estado das dependências de um cliente, e exporta as operações *createContext* e *deleteContext* para a sua criação e remoção.

Apesar das várias vantagens deste algoritmo, é necessário guardar uma lista de dependências para cada item, o que pode levar a problemas de escalabilidade devido à quantidade de informação armazenada. Além disso, este algoritmo permite apenas transações de leitura, não permitindo transações de escrita. Surge assim a necessidade para sistemas mais avançados, como por exemplo o algoritmo *Wren* [SDZ18].

Wren

O sistema *Wren* [SDZ18] é um sistema **AP** que surge como evolução de outros sistemas [LFKA11, ATB+16] desta família. Primeiramente, implementa coerência causal transacional, que permite realizar leituras de um instantâneo causal e escritas atómicas de múltiplos itens, ou seja, através de transações o cliente pode ler ou escrever múltiplos itens com apenas um pedido ao servidor.

Além disso, o algoritmo suporta leituras não bloqueantes e, ao mesmo tempo, permite à aplicação escalar horizontalmente através de **fragmentação**, apresentando uma solução com baixa latência e disponível perante partições de rede. Para isso implementa um sistema distribuído de armazenamento chave-valor multi-versão com N partições, em que cada item é atribuído a uma partição através de uma função de *hash*. Os dados são totalmente replicados por todos os centros de dados de forma assíncrona. O algoritmo resolve os conflitos de escrita escolhendo o mais recente, ou seja, com a regra conhecida como *last-writer-wins*.

A ordem causal é mantida através da utilização de duas etiquetas temporais por cada transação mantidas com os protocolos *Binary Dependency Time* e *Binary Stable Time*. A primeira etiqueta temporal, *local stable timestamp* (*lst*), corresponde ao valor de relógio mínimo dentro de todas as partições do centro de dados local. Já a segunda, *remote stable timestamp* (*rst*), corresponde ao *lst* mínimo estável dentro de todos os centros de dados do sistema. O valor de relógio necessário para calcular estas etiquetas é partilhado entre os centros de dados por um protocolo epidémico, de modo a reduzir o número de mensagens transmitidas no sistema. Com estes dois escalares cada servidor consegue controlar as dependências de dados, sem ser necessário guardar meta-dados individualmente para cada dependência, melhorando o desempenho do sistema em grande escala. Na Figura 8 podemos observar o cálculo do *remote stable timestamp* com recurso ao protocolo *Binary Stable Timestamp*, em comparação com outros algoritmos.

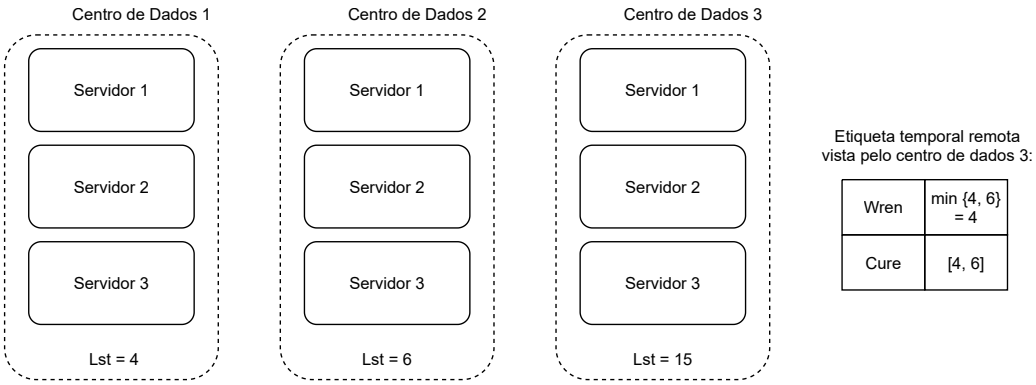


Figura 8: Protocolo *BiST* [SDZ18]

Por último, o *Wren* implementa o protocolo *Client-Assisted Non-Blocking Transaction Reads* que garante que o instantâneo dos dados visíveis por uma transação é dado pela união de um instantâneo estável da base de dados, garantido pelos protocolos descritos em cima, com uma *cache* do cliente que contém todos os dados que foram escritos pelo cliente mas ainda não se encontram aplicados em todos os centros de dados. Contudo, este protocolo expõe ao cliente dados ligeiramente no passado, porque o instantâneo obtido contém apenas os dados que estão aplicados em todos os centros de dados, sendo esta a desvantagem perante a melhoria de desempenho obtida com este algoritmo em comparação com outros que enumeram as dependências entre itens de dados [LFKA11].

2.2 Simulação

2.2.1 Simulação em geral

Simulação é o processo de desenhar sistemas de computadores para imitar vários tipos de processos do mundo real. Este processo é geralmente chamado de sistema, e é o ambiente sobre o qual o modelo de simulação atua.

Um sistema é definido como uma coleção de entidades que atuam e interagem em conjunto para alcançar um objetivo lógico. Já o estado do sistema é a coleção de variáveis necessárias para o modelar [Law15].

Por exemplo, se quisermos estudar o tempo que demora um funcionário de caixa de um supermercado a atender os clientes que estão em espera, o sistema é composto por estas duas entidades. Algumas variáveis de estado poderão ser o número de clientes, o número de caixas registadoras, etc. Esta interação pode ser descrita com a Figura 9. O sistema é composto por duas entidades, servidor e cliente. Quando um cliente chega ao sistema aguarda numa fila de espera até ser a sua vez de ser atendido. O servidor atende os clientes sequencialmente e demora um certo intervalo de tempo a atender cada cliente. Após ser atendido, o cliente sai do sistema.

Figura 9: Sistema Simples de Simulação [Law15]

Sistemas de simulação podem ser de dois tipos: discretos ou contínuos. Simulação contínua envolve sistemas que operem de forma contínua, como por exemplo, simulação de meteorologia onde a temperatura estimada evolui de uma forma contínua. Por outro lado, simulação discreta (**Discrete Event Simulation (Simulação de Eventos Discretos)** (**DES**)) envolve variáveis discretas, que "incrementam como um degrau". Um exemplo claro destes sistemas é o exemplo do supermercado discutido anteriormente [Law15, Mat08].

2.2.2 Simulação de Eventos Discretos

Com o passar dos anos e diversos avanços no mundo de simulação, surgiram diversos paradigmas de simulação de eventos discretos. Alguns desses paradigmas que vamos analisar são o paradigma orientado à atividade, o paradigma orientado ao evento e o paradigma orientado ao processo.

Paradigma orientado à atividade

Neste primeiro paradigma, o tempo de execução da simulação é particionado em incrementos de tempo bastante pequenos. Para simular toda a atividade de um sistema durante um determinado tempo de simulação, o modelo irá iterar por todos os intervalos de simulação, verificando em cada um deles se existe alguma atividade a decorrer nesse instante [Mat08].

Recorrendo ao exemplo de um modelo de simulação de um funcionário de uma caixa de supermercado, com tempo máximo de execução de 10 minutos e incremento de 1 milissegundo. Neste modelo, começando de um instante 0, a cada 1 milissegundo irá-se verificar se alguma atividade se alterou no sistema (chegou um cliente, saiu um cliente, etc.), de modo a simular o seu comportamento, até atingir os 10 minutos de simulação.

Paradigma orientado ao evento

No paradigma orientado ao evento é guardada uma lista de eventos, que são os eventos que irão acontecer na simulação do sistema. Em seguida, em vez de iterar por um determinado incremento como no paradigma anterior, o sistema irá ver qual o tempo até à execução dos eventos, e avança o tempo de simulação para o evento mais próximo.

No exemplo anterior, se houvesse clientes a chegar à caixa de supermercado a cada 4 minutos, em vez de realizar 24000 iterações (como no paradigma anterior), iríamos avançar o tempo de simulação diretamente para este instante.

Paradigma orientado ao processo

Neste último paradigma, são definidos vários processos, em que cada um deles serve como um modelo para uma determinada atividade do sistema. Cada processo pode ter um de três estados: ativo, onde está no momento a ser processado pelo simulador; *holding*, onde está à espera que um intervalo de tempo simulado passe; ou em espera, onde se encontra numa fila à espera que ocorra um evento [Sch86]. Além disso, existe um gestor de processos responsável por gerir vários processos que possam estar ativos num determinado instante [Sch90b].

Assim, no exemplo da caixa de supermercado iríamos ter dois processos, um que iria simular a chegada dos clientes à loja e outro que iria simular o atendimento destes pelo funcionário da caixa.

Comparação entre paradigmas

O paradigma orientado ao evento oferece então muito mais eficiência que o paradigma orientado à atividade, uma vez que não têm de ser simulados todos os instantes temporais, mas apenas os que contem eventos. Deste modo, este tendo sido o paradigma de simulação mais utilizado durante bastantes anos.

Em relação ao paradigma orientado ao processo, o paradigma orientado ao evento oferece uma maior facilidade de implementação, pois não utiliza processos, e uma maior flexibilidade de adicionar novos eventos a meio da execução do algoritmo. Em relação à velocidade de execução, o paradigma orienta ao processo, comparativamente ao paradigma orientado ao evento, apresenta um tempo de arranque mais longo devido à inicialização dos processos mas apresenta tempo de execução mais rápido devido ao paralelismo. Este paradigma apresenta também código mais modular, uma vez que cada atividade está a ser executada por um processo diferente. Deste modo, hoje em dia é o paradigma mais utilizado, havendo diversas ferramentas de simulação baseadas nele.

2.2.3 Simuladores

Simuladores são ferramentas ou programas que nos permitem desenvolver um modelo de simulação de um sistema, de modo a analisar o seu desempenho.

Inicialmente, na década de 1960, houve a necessidade de criar linguagens específicas para ser possível modelar esses sistemas, como a linguagem *SIMULA* [Mat08]. Contudo, com o desenvolvimento das ferramentas de simulação, começaram-se a produzir bibliotecas para linguagens já existentes, tornando a curva de aprendizagem da modelação mais pequena. Destas ferramentas, sobressae no âmbito deste

trabalho a ferramenta para *Python*: *SimPy* [Sim22].

SimPy

SimPy é uma biblioteca de simulação de eventos discretos para *Python*. Esta biblioteca utiliza o paradigma orientado ao processo, em que cada componente é modelado num processo que opera concorrentemente com outros. Porem, em vez de utilizar fios de execução, esta biblioteca faz uso dos geradores de *Python*. Esta ferramenta permite parar a execução de uma função de forma prematura, executando outra e podendo retornar ao ponto de paragem da primeira mais à frente, utilizando a palavra-chave *yield* [Mat08].

Os processos (*Process*) existem num ambiente (*Environment*) e, durante a sua vida, produzem eventos para esperar por eles. Quando produz um evento, o processo fica suspenso e quando o evento ocorre, o *SimPy* continua o processo [Sim22].

Para além disso, a biblioteca *SimPy* inclui recursos partilhados (*Resources*) que nos ajudam a resolver problemas de modelação. Estes recursos têm uma quantidade limitada e têm de ser requisitados por um processo para poderem ser utilizados. Caso o recurso não esteja disponível na altura do pedido, o processo fica em espera até este estar disponível [Sim22].

Na Listagem 1, segue um exemplo que nos demonstra um ambiente com um processo de um carro, onde é produzido um evento para o pôr a fazer uma viagem com duração de *driving_time*, seguido de um pedido para utilizar o carregador (recurso *bcs*) e, por último, um evento para o pôr a carregar com duração *charging_time*.

```
def car(env, name, bcs, driving_time, charge_duration):
    yield env.timeout(driving_time)
    print('%s arriving at %d' % (name, env.now))
    with bcs.request() as req:
        yield req

        print('%s starting to charge at %s' % (name, env.now))
        yield env.timeout(charge_duration)
        print('%s leaving the bcs at %s' % (name, env.now))
```

Listagem 1: Exemplo de um processo em *SimPy* [Sim22].

2.2.4 Simulação Paralela de Eventos Discretos

Simulação paralela de eventos discretos (**PDES**), também conhecida como simulação distribuída, consiste em executar um simulador de eventos discretos num computador paralelo, de modo a obter tempos de simulação mais reduzido comparativamente à execução de um simulador sequencial [Fuj90].

A utilização de modelos de eventos discretos do *SimPy* permite a utilização futura de ferramentas de simulação paralela de eventos discretos (**PDES**) como *Simian* [SEL15], *SimGrid* [CGL+14] ou *Simulus* [Liu20] caso a simulação numa única máquina seja limitativa.

2.2.5 Modelos de Simulação de Bases de Dados Distribuídas

No desenvolvimento do modelo de simulação de base de dados, de modo a replicar o seu comportamento, são utilizadas filas de espera. Para isso é utilizada a classificação de *Kendall* onde se utilizam vários símbolos e barras para representar o processo da fila de espera , com por exemplo *A/B/X/Y/Z*. Neste exemplo, o A representa o distribuição utilizada para as chegadas à fila, o *B* representa a distribuição utilizada para o tempo de serviço do pedido e o *X* representa o numero de canais paralelos. Além disso, o *Y* representa a capacidade máxima do sistema e o *Z* representa a disciplina da fila [GSTH08]. Estes dois últimos campos, em caso de omissão, são assumidos respetivamente como ∞ e *FCFS*.

Através da Tabela 1 podemos verificar os diferentes símbolos utilizados para representar os diversos parâmetros de uma fila de espera.

Característica	Símbolo	Explicação
Distribuição de Chegada e Distribuição do Serviço	*M*	Exponencial
	D	Determinística
	E_k	*Earlang*
	H_k	Hiperexponencial
	PH	*Phase-Type*
	G	Geral
Sistemas em Paralelo	1,2,...,∞	
Capacidade do Sistema	1,2,...,∞	
Disciplina da Fila	*FCFS*	Primeiro a chegar, primeiro servido (*FIFO*)
	LCFS	Último a chegar, primeiro servido (*LIFO*)

RSS	Seleção de serviço aleatório
PR	Fila de Prioridade
RR	*Round Robin* [NJ00]
GD	Geral

Tabela 1: Símbolos da notação de *Kendall* [GSTH08]

Por exemplo, se tivermos um sistema $M/D/m/\infty/LCFS$, iremos ter uma distribuição de chegadas exponencial (*Poisson*), uma distribuição de serviço determinística, com *m* servidores em paralelo, sem capacidade máxima e com uma disciplina de fila *Last Come First Served*.

Nas próximas secções iremos-nos basear no artigo [NJ00], que compara diferentes abordagens de modelação de bases de dados distribuídas, principalmente em relação ao modelo da base de dados, de comunicação, de **replicação**, de acesso de dados e de processo de transações.

Modelos de Bases de Dados

Em relação à modelação da base de dados, existem bastantes abordagens diferentes em relação às filas de espera utilizadas para representar o comportamento da base de dados.

Nos primeiros estudos realizados, utilizava-se um processo de filas de espera $M/M/m/FCFS$, onde m servidores recebiam e atendiam pedidos numa distribuição de *Poisson*, em filas *First Come First Served*. Uma grande desvantagem destes modelos iniciais é a utilização de apenas uma fila para a chegada de pedidos, negligenciando o tempo de comunicação entre diferentes servidores [NJ00].

Posteriormente, alguns estudos modelaram uma rede de filas para representar o sistema. Neste caso, a comunicação entre os servidores é modelada à parte, e o processo de filas de espera para cada servidor é, no caso mais simplificado, $M/M/1$. No entanto, outros estudos assumem um modelo mais complexo, como o modelo $M/G/1$, com uma distribuição geral para o tempo de serviço, ou o modelo $M/H_2/1$, com uma distribuição hiperexponencial. No estudo [HLC96], por exemplo, é utilizado um modelo $M/G/1/\infty/RR$.

Outro abordagem que difere de modelo para modelo, no caso da utilização de um modelo de comunicação, é a utilização de uma rede de filas aberta ou fechada. Numa rede de filas fechada, existe um número fixo de pedidos no sistema, enquanto numa rede de filas aberta não existe.

Algo que é comum a todos os modelos é a utilização de uma distribuição de *Poisson* para a chegada dos pedidos e a utilização de filas sem capacidade máxima em cada servidor. Além disso, algo também

bastante comum é a utilização da disciplina *FCFS*.

Modelos de Comunicação

Em relação à modelação da comunicação entre servidores, a maioria dos estudos abordados pelo artigo [NJ00] utilizam uma rede sem capacidade máxima e com um tempo de transmissão constante ($M/D/\infty$), onde a rede nunca é considerada um **gargalo**.

No entanto, o modelo de comunicação pode ser também modelado com um tempo de transmissão variável, mas na mesma sem capacidade máxima da rede. Neste caso, o tempo de transmissão é modelado através de uma distribuição de *Poisson* ($M/M/\infty$) ou através de uma distribuição Geral ($M/G/\infty$). No estudo [RTH96], podemos verificar um exemplo de um sistema que utiliza um sistema ($M/G/\infty$) para modelar a rede.

Apesar de tudo, também é possível modelar a rede com capacidade limitada.

Modelos de Replicação

Para o desenvolvimento do modelo de **replicação** de dados, existe uma grande divergência na quantidade de dados que são replicados e na quantidade de servidores utilizados para guardar réplicas, devido à utilização de diferentes arquiteturas de bases de dados distribuídas.

De acordo com o artigo [NJ00], apesar de existirem estudos de bases de dados distribuídas que não utilizam **replicação**, existem 5 principais abordagens diferentes:

- **Todos os objetos em todos os servidores -** Estes modelos são chamados de modelos de **replicação** total, pois todos os dados são guardados em todos os servidores da rede. Contudo, este tipo de **replicação** foi várias vezes provado não ser o ótimo.

- **Todos os objetos em alguns servidores -** Nestes modelos, que são modelos de **replicação** parcial unidimensional, todos os objetos são guardados em alguns servidores, onde é assumido que esses objetos estão igualmente distribuídos em toda a rede. O número de réplicas onde os objetos são guardados é representado por $r \in \{1, 2, ..., n\}$, onde n representa o número total de servidores [Ulu94].

- **Alguns objetos em todos os servidores -** Esta abordagem, juntamente com a anterior, representa modelos de **replicação** parcial unidimensional. Esta é considerada não ótima também, pois alguns dados serão replicados totalmente, enquanto outros não serão replicados. A percentagem de dados a ser replicados é dada por $r \in [0; 1]$ [GHOS96b].

- **Alguns objetos em alguns servidores -** Estes modelos são denominados de modelos de **replicação** parcial bidimensional, pois integram as duas abordagens anteriores numa só. Este modelo tem a vantagem de aplicações com dados que são atualizados bastantes vezes poderem ser guardados em poucos servidores e aplicações com dados lidos muitas vezes poderem ser guardados em muitos servidores. O modelo é representado por $(r1, r2) \in [0; 1] \times \{1, 2, ..., n\}$ onde $r1$ e $r2$ são, respetivamente, a percentagem de dados a ser replicados e o número de réplicas onde os objetos são guardados.

- **Replicação por objeto -** Apesar do modelo anterior ser uma evolução dos anteriores, tem uma falha bastante grande: todos os dados têm o mesmo grau de **replicação**. Assim, modelos de **replicação** por objeto assumem que cada objeto tem um número de réplicas para o qual vai ser replicado. Assim, o número de réplicas onde um objeto vai estar guardado é dado por $r(i)$, onde $r : 1, 2, ..., d \rightarrow 1, ..., n$, em que d representa o número de objetos e n o número de servidores. Contudo, este modelo é bastante complexo e não foi integrado em nenhum estudo.

Na Figura 10 podemos ver uma comparação entre diferentes abordagens de **replicação** em simulação, onde o eixo $r1$ corresponde à percentagem de dados replicados, o eixo $r2$ corresponde ao número de réplicas e o eixo vertical à percentagem do sistema replicado. Primeiramente, podemos observar os dois extremos, a inexistência de **replicação**, onde nenhum dos dados é replicado ($r1 = 0$ e $r2 = 0$), e a **replicação** total, onde todos os dados são replicados em todas as réplicas ($r1 = 1$ e $r2 = 1$).

Além disso, quando $r1 = 1$ e $r2 \in [0, 50]$ temos **replicação** de todos os objetos para alguns servidores e quando $r1 \in [0, 1]$ e $r2 = 50$ temos **replicação** de alguns objetos para todos os servidores. Em ambas as abordagens é possível ver que o aumento a percentagem de dados replicados no sistema aumenta linearmente com o aumento do respetivo parâmetro a variar.

Por último, quando $r1$ e $r2$ não assumem nem o seu valor máximo, nem o seu valor mínimo, temos **replicação** de alguns objetos para alguns servidores. Em comparação com as outras abordagens esta torna-se a mais versátil, pois conseguimos manipular as duas variáveis e encontrar um ponto de ótimo para a execução do modelo.

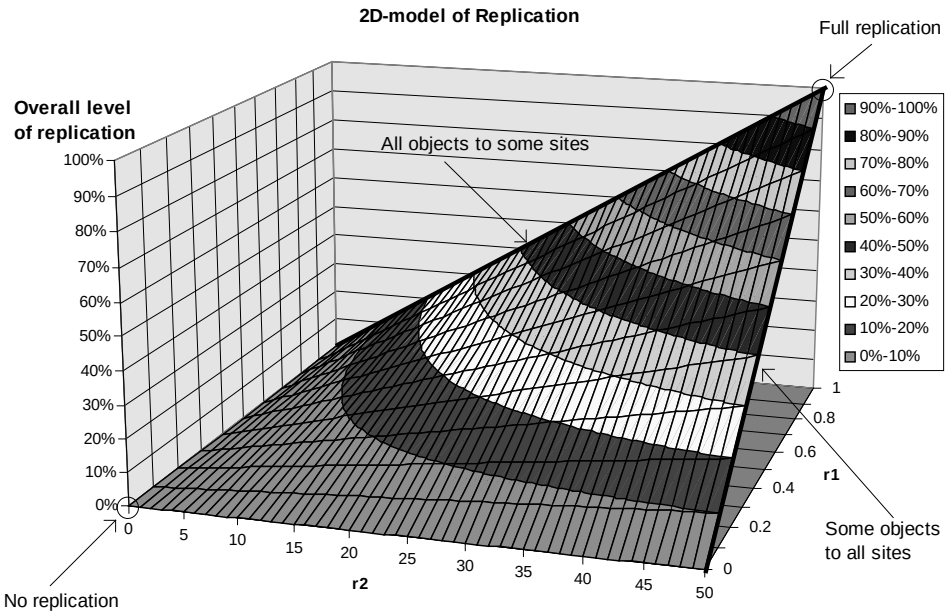


Figura 10: Comparação entre percentagem de replicação de diferentes modelos [NJ00]

Modelos de Acesso de Dados

Em relação ao modelo de acesso de dados, este pode assumir um acesso aos dados de forma uniforme, onde todos os dados têm a mesma probabilidade de ser acedidos, ou não uniforme. Neste caso, dados acedidos de um modo não uniforme podem ser classificados como:

- **Modelos hot-spot -** Neste modelo, certos grupos de dados têm uma probabilidade maior de ser acedidos que outros (acesso *b-c*, em que uma percentagem *b* dos pedidos é feita sobre uma percentagem *c* dos dados).

- **Modelos de localidade -** Neste modelo, dados locais têm maior probabilidade ser acedidos que dados remotos (acesso *b-l*, em que uma percentagem *b* dos dados pode ser acedida localmente) [Ulu94] [GHOS96b].

Modelos de Transações e Modelos de Controlo de Concorrência

Em relação ao modelo das transações, apesar de alguns modelos assumirem apenas transações de escrita ou de leituras, é importante fazer uma distinção entre ambas, visto que as primeiras requerem menos processamento para a base de dados [Ulu94] [GHOS96b]. Contudo, apenas a distinção de leituras e escritas não é suficiente, sendo necessária uma avaliação da aplicação no mundo real para estudar as cargas de ambas no seu âmbito.

Em relação ao modelo de controlo de concorrência, na maioria dos casos é utilizado **locking** dinâmico, visto que em **locking** estático todos os trincos são adquiridos para realizar uma transação, e não apenas os trincos necessários. Em certos casos, a probabilidade de um conflito é tão baixo que o controlo de concorrência é negligenciado, tendo apenas em consideração o tempo necessário para adquirir os trincos.

A partir dos tópicos acima podemos observar as diferentes escolhas possíveis para desenvolver um modelo de simulação de bases de dados.

Na Tabela 2 podemos observar de forma resumida as várias alternativas disponíveis para desenvolver o modelo.

Componentes do Modelo	**Opções de Modelação**			
Base de Dados	Sistema *M/M/m* para *m* servidores	Fila de *m* sistemas (*M/M/1*)	Fila de *m* sistemas (*M/G/1*)	Uma rede por servidor
Comunicação	Atraso constante Capacidade ilimitada (*M/D/*∞)	Atraso exponencial Capacidade limitada (*M/M/*∞)	Atraso exponencial Capacidade limitada (*M/M/1*)	Atraso geral Capacidade limitada (*M/G/1*)
Replicação	Sem replicação	Replicação total	Replicação parcial de 1 dimensão	Replicação parcial de 2 dimensões
Qualidade da Replicação	Ignorada	Colocação das réplicas	Seleção das réplicas	Colocação e seleção das réplicas
Acesso de Dados	Uniforme	Localidade de dados	*Hot-Spot*	Localidade e *Hot-Spot*
Transação	Só leituras	Só escritas	Escritas e leituras	Mais de 2 tipos de transação
Controlo de Concorrência	Ignorado	*2PL*, só trincos exclusivos	*2PL*, trincos de escrita e leitura	Outros protocolos

Tabela 2: Opções para Modelo de Simulação de Bases de Dados Distribuídas [NJ00]

2.2.6 Simulação de Bases de Dados Distribuídas

Tal como acontece com a generalidade dos sistemas de computação, existem várias propostas de utilização de simulação na avaliação de sistemas de bases de dados, mas são focadas na concorrência interna do servidor não na sua distribuição [HLC96, NJ00]. Uma abordagem particularmente interessante é a concretização de um sistema real mas simplificado com variantes para diferentes métodos de controlo de concorrência [YBP+14] que depois é executado num simulador de sistema completo [MKK+10] para analisar em detalhe o comportamento em plataformas com um grande número de núcleos.

A implementação e avaliação de algoritmos distribuídos em grande escala, em especial os que abordam desafios relacionados com a tolerância a faltas e a resiliência, é uma tarefa complexa, e o desem-

penho e o comportamento destas aplicações são altamente influenciados pelas propriedades da rede. Desta forma ferramentas como *Babel* [FCPL22] e *Kollaps* [GNS+20] ajudam na rápida e eficiente prototipagem de especificações ou algoritmos assim como a emular/simular rede e sistemas distribuídos complexos. No entanto, estas ferramentas são genéricas e os algoritmos de bases de dados geo-replicadas são complexos, mas existe uma base comum para a simulação das propriedades dos mesmos que tirámos vantagem neste artigo ao propor um modelo configurável de simulação.

O *FoundationDB* [ZXS+21] é uma base de dados geo-replicada e integra um simulador determinístico onde todas as fontes de não-determinismo e de comunicação são abstraídas. Apesar, desta abordagem ser eficaz para testar novas funcionalidades do sistema, esta não é adequada para o propósito de comparar e avaliar os compromissos de diferentes algoritmos de bases de dados geo-replicadas, pois implica implementações em detalhe dos algoritmos.

Em qualquer dos casos, é importante a utilização de modelos de carga que reproduzam a localidade de acessos para obter uma avaliação de desempenho realista. A abordagem seguida aqui é semelhante ao *YCSB* [CST+10], adequando-se por isso a sistemas chave-valor. Tradicionalmente, a carga de trabalho preferida para avaliar sistemas transacionais e com processamento de interrogações **SQL** é o *TPC-C* [TPC10], mas é de mais difícil aplicação em sistemas chave-valor. É também interessante a possibilidade de integrar cargas de trabalho pensadas explicitamente para o teste de sistemas geo-replicados com alta disponibilidade [TZB+17]. Ao contrário destas alternativas, focadas no desempenho, alguns geradores de carga como o *Elle* [KA20] permitem avaliar a correção do sistema através da análise dos resultados obtidos, o que no nosso caso não é útil uma vez que os dados são efetivamente abstraídos.

Capítulo 3

Modelo SAGeo

3.1 Modelo Genérico

Nesta secção descrevemos e analisamos o modelo de simulação de bases de dados geo-replicadas, *SAGeo*, desenvolvido na ferramenta *SimPy*, apresentada na Seção 2.2. Este modelo-base pode ser configurado de modo a corresponder com as especificações do sistema de base de dados em estudo. A implementação de uma nova configuração é um processo bastante simples, sendo apenas necessário alterar pequenas parte do código no que toca a troca de mensagens e, caso necessário, adicionar componentes modulares, para implementar um novo algoritmo.

De modo a modelar este sistema, seguimos a abordagem descrita no capítulo anterior em que primeiramente se modela os componentes do sistema em separado. Primeiramente, em relação ao modelo de bases de dados, utilizamos para cada servidor uma fila de espera M/M/k/∞/FCFS, ou seja, a distribuição de chegada e a distribuição de serviço seguem uma distribuição de *Poisson*, com *k* fios de execução a atender os pedidos, não existe capacidade máxima de pedidos e a disciplina da fila é primeiro a chegar, primeiro a ser servido. Contudo, ao configurar o modelo é possível alterar a disciplina da fila para fila de prioridade, por exemplo. Em relação ao modelo de comunicação, é utilizada uma fila de espera com tempo de transmissão que segue uma distribuição de *Poisson* e é possível configurar a capacidade da rede para ser ou não limitada, sendo então a fila M/M/1/j/FCFS ou M/M/1/∞/FCFS. No que toca ao modelo de **replicação**, o modelo está inicialmente configurado para utilizar **replicação** parcial bidimensional, ou seja, alguns objetos estão guardados em alguns servidores, mas é possível configurar um sistema com **replicação** total. Além disso, em relação ao modelo de acesso a dados, é possível configurar o acesso a dados através de uma distribuição uniforme ou por *hot-spot*, através de uma distribuição *Zipfian*. Por último, em relação ao modelo de transações, é possível configurar a carga de transações de escritas e leituras que entram no sistema e, em relação ao modelo de controlo de concorrência, é utilizado **locking** dinâmico, se o sistema em questão o utilizar.

Em relação aos componentes do modelo, este é composto por um conjunto de centros de dados e pela rede, que é um recurso partilhado por todos os centros de dados, como podemos observar na Figura 11. Cada centro de dados é modelado por um processo e é composto por um gerador de carga, que gera transações a serem executadas na base de dados, e um número finito de servidores, sendo cada um responsável por uma partição dos dados.

O gerador de carga é também modelado por um processo e gera transações com intervalo que segue uma distribuição de *Poisson*, como está descrito em cima, com tamanho médio e carga de escritas e leituras configurável.

Cada servidor é modelado por um processo e é composto por um **CPU** (com um número de núcleos configurável) e um conjunto de discos (também configurável). Algo que pode ser adicionado ao modelo, caso o algoritmo em questão necessite, é uma tabela de trincos, modelada por uma lista de recursos. Estes recursos são partilhados por um conjunto de fios de execução responsáveis por atender pedidos de clientes e um segundo conjunto de fios de execução responsáveis por atender pedidos de outros servidores. O conjunto de fios de execução para clientes e servidores é separado de modo a introduzir prioridade nos pedidos vindos de servidores. Contudo, esta funcionalidade pode ser removida na configuração do modelo.

Além desta arquitetura base é possível adicionar componentes ao modelo de modo a corresponder com as funcionalidades que estamos a simular. Para além disso, o código executado para atender cada pedido e a geração dos pedidos também são editáveis, de modo a poder ser ajustado ao comportamento do sistema em questão.

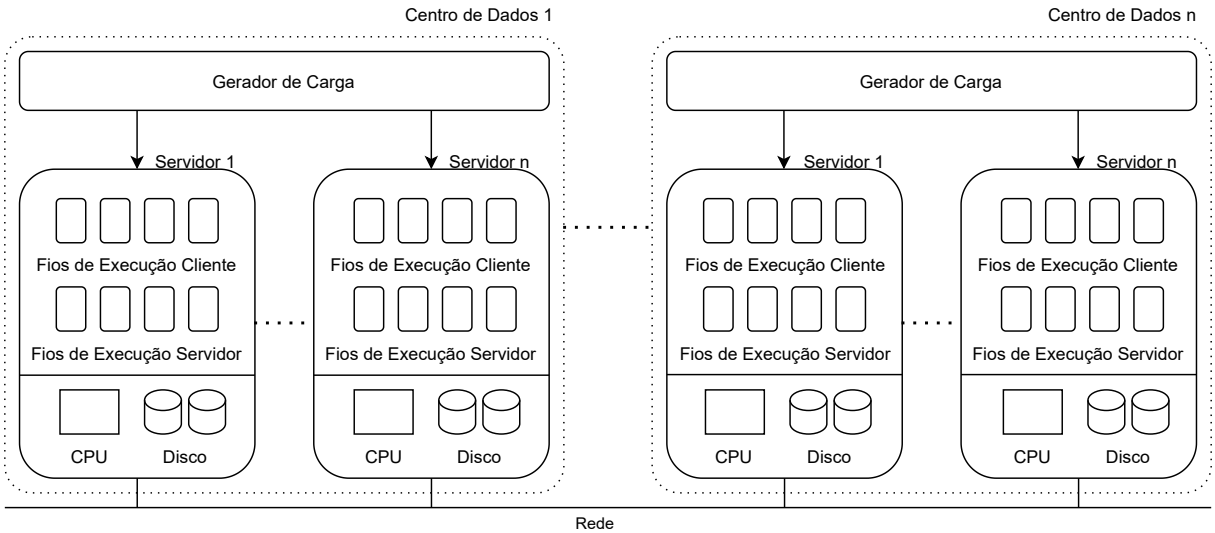


Figura 11: Arquitetura do modelo de simulação.

3.1.1 Exemplos de código de simulação

Apesar do nosso modelo de simulação de sistemas de bases de dados estar destinado a vários tipos de bases de dados, muitos componentes do nosso modelo são reutilizáveis, sendo possível utilizá-los quando construimos um novo modelo.

Por exemplo, uma prática muito comum em vários sistemas de bases de dados é a utilização de trincos. Este processo é muito fácil de simular recorrendo aos recursos partilhados do *SimPy*. Para adquirir o trinco apenas é necessário fazer um pedido do recurso e utilizar a palavra chave do *Python* *yield*, como podemos observar na Listagem 2. Este código pode ainda ser modificado para adicionar funcionalidade aos trincos, como por exemplo, utilizar filas de prioridade e desafetação forçada.

```
def aquire_locks(self, variables):
    for var in variables:
        lock = self.shard.locks[var].request()
        yield lock
```

Listagem 2: Código para adquirir um trinco.

Além disso, algo também essencial a todos os modelos de sistemas bases de dados é a simulação do tempo de execução de **CPU** de um item de uma transação. Para isso, primeiro temos de adquirir o recurso **CPU**, que tem capacidade igual ao número de cores do **CPU**, como podemos observar na Listagem 3. Posteriormente, utilizamos a função *generate_timeout* do gerador de carga para gerar um tempo de execução que segue uma distribuição de *Poisson* e, através do método *timeout* e da palavra chave *yield*, simulamos a execução. Por último, libertamos o recurso do **CPU** para poder ser utilizado por outro processo.

```
def use_cpu_execute(self):
    cpu_req = self.shard.cpu.request()
    yield cpu_req

    cpu_time = self.shard.zone.generator
                .generate_timeout(self.vars["CPU_EXECUTE"])
    yield self.env.timeout(cpu_time)

    self.shard.cpu.release(cpu_req)
```

Listagem 3: Código para simular o tempo de execução.

3.1.2 Introdução de parâmetros

No nosso modelo de simulação os parâmetros são introduzidos através de um ficheiro de configuração em formato *JSON*, onde são indicados todos os valores de simulação. Alguns destes parâmetros são o número de servidores, o número de centros de dados, o tempo de simulação, etc. Além destes parâmetros é também possível adicionar novas variáveis, de modo a aproximar o modelo de simulação da base de dados em questão.

Além dos parâmetros estáticos na simulação, é também introduzido um parâmetro que é variado na simulação, de modo a poder comparar o efeito da sua variação no comportamento do sistema. Isto é feito através da utilização do campo *VARIATION_VARIABLE* do ficheiro *JSON*. Por exemplo, como podemos observar na Listagem 4, nesta simulação estamos a utilizar 3 centros de dados com 3 servidores em cada um, com um tempo máximo de simulação 10000000. A simulação vai ser executada 4 vezes variando a taxa média de chegada de pedidos ao sistema pelos 4 valores descritos.

```
{
    "VARIATION_VARIABLE": "ARRIVAL_MEAN",
    "VARIATION_VALUES": [200, 500, 1000, 2000],

    "NUM_DATA_CENTERS": 3,
    "NUM_SHARDS": 3,
    "MAX_SIM_TIME": 10000000,

    ...
}
```

Listagem 4: Excerto de ficheiro de configuração.

Os parâmetros de simulação genéricos para o modelo podem ser encontrados na Tabela 3, junto com a sua explicação.

Identificador	Explicação
DATABASE_SIZE	Tamanho da base de dados
DATA_DISTRIBUITION	Distribuição dos dados
NUM_DATA_CENTERS	Número de centros de dados
NUM_SHARDS	Número de servidores por centro de dados
POOL_SIZE	Número de fios de execução
CPU_CORES	Número de cores **CPU**
IO_DISKS	Número de discos *IO*
TRANSACTION_SIZE	Tamanho médio da transação
TRANSACTION_DISTRIBUTION	Distribuição de escritas e leituras
ARRIVAL_MEAN	Taxa de chegada de pedidos
NETWORK_LATENCY_CLIENT	Latência entre servidor e cliente
NETWORK_LATENCY_INTRA_ZONE	Latência entre servidores no mesmo centro de dados
NETWORK_LATENCY_INTER_ZONE	Latência entre servidores em centros de dados diferentes
CPU_EXECUTE	Tempo de execução de uma transação
CPU_READ	Tempo de leitura de uma transação
IO_WRITE	Tempo de escrita em disco
NETWORK_CAPACITY	Capacidade máxima da rede

Identificador	Explicação
MAX_SIM_TIME	Tempo de simulação
SEED	Semente de aleatoriedade

Tabela 3: Parâmetros do modelo de simulação

3.1.3 Recolha de resultados

Após a execução do modelo, é extraído de cada execução o tempo de resposta médio de pedidos, o débito e o número de transações executadas. Estas estatísticas são depois apresentadas em 4 gráficos: o parâmetro a variar em função do tempo de resposta, o parâmetro a variar em função do débito, o débito em função do tempo de resposta e o parâmetro a variar em função do número de transações, como podemos observar pela Figura 12. Contudo, o módulo de apresentação dos resultados pode facilmente ser alterado para melhor apresentar os mesmos para um caso de estudo em particular.

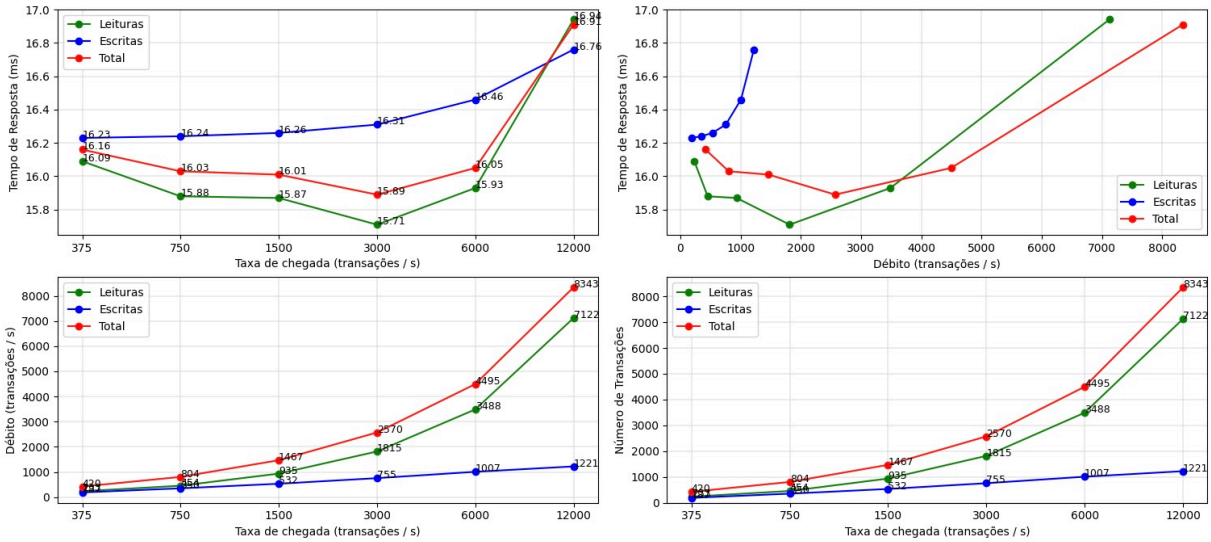


Figura 12: Exemplo de resultados.

3.2 Implementação de Algoritmos

Nesta secção iremos apresentar a configuração do modelo de simulação para os sistemas de bases de dados **DBSM**, *Spanner* e *Wren*, explicando que módulos foram alterados. Além disso, iremos apresentar os parâmetros necessários para a execução de cada modelo.

3.2.1 DBSM

Nesta secção apresentamos a configuração do modelo de simulação para o sistema de bases de dados **DBSM**. Este sistema tem a particularidade de não ter **fragmentação**, sendo o seu modelo mais simples que o modelo de simulação definido. Contudo, é na mesma possível configurar o modelo de modo a simular este algoritmo.

Primeiramente, como podemos observar na Figura 13, é removido o módulo centro de dados da arquitetura geral, sendo cada localização representada por um servidor. Cada servidor contém um gerador de carga responsável por gerar as transações que este irá executar.

Em vez de um conjunto de fios de execução destinados a pedidos de clientes e outro destinados a pedidos de servidores, o modelo contém fios de execução destinados a executar transações e um único fio de execução destinado a realizar a certificação das transações (fio de execução para terminação). O código a ser executado por estes fios de execução é também alterado de modo a corresponder ao comportamento do algoritmo.

Por último, é adicionado um recurso que corresponde ao disco de *Log*, que é o disco onde as transações são guardadas de forma ordenada após a troca de mensagens entre servidores com o algoritmo de ordenação total, e uma tabela de trincos, que é modelada por uma lista de recursos, cada um correspondente a um item da base de dados.

Na configuração deste modelo o consenso distribuído entre réplicas necessário para modelar o **Atomic Broadcast** foi simulado através de uma latência constante, de modo a garantir que todos os processos recebem as mensagens na mesma ordem, para que a certificação seja realizada corretamente em todos os servidores. Deste modo, na configuração deste algoritmo não é possível simular partições do sistema. Além disso, a latência constante gerada para simular o consenso não tem em conta o número de réplicas do sistema.

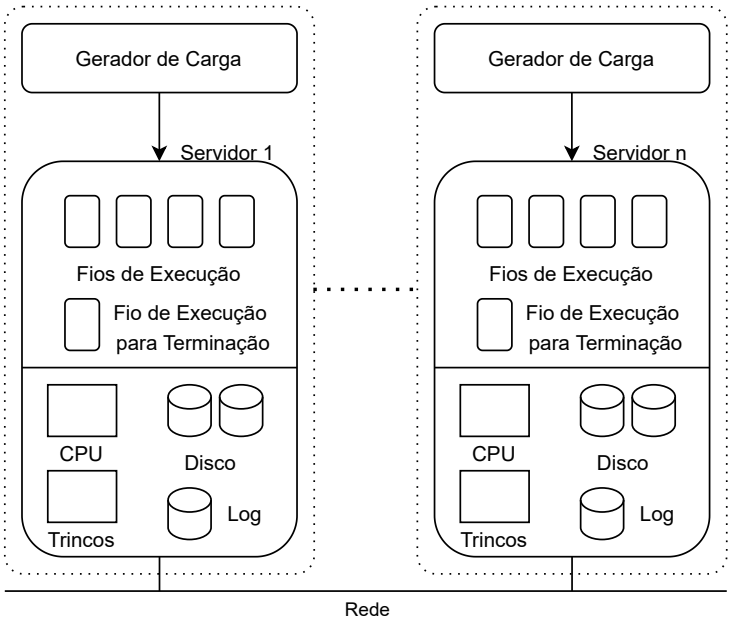


Figura 13: Arquitetura do modelo de simulação para **DBSM**

Troca de mensagens

Como podemos observar na Figura 14, para a realização de uma transação de escrita na nossa implementação do algoritmo **DBSM** utilizamos os seguintes passos:

1. O gerador de carga gera uma transação de escrita e seleciona, aleatoriamente, um servidor para executar essa transação.

2. O servidor atribui um fio de execução à transação, que irá executar a mesma, adquirindo os recursos dos trincos e consumindo o recurso **CPU**. Posteriormente, após a simulação da execução por parte do **CPU**, são libertados os trincos.

3. A transação é propagada para todos os servidores através de uma latência constante, de modo a simular o **Atomic Broadcast**.

4. A certificação da transação é realizada em cada servidor de forma sequencial, pelo fio de execução de terminação, onde é consumido mais uma vez o recurso **CPU** e o recurso *Log*. No fim do processo de certificação, a transação é dada como confirmada ou como abortada e passada novamente para uma fio de execução para finalizar a transação.

5. Caso a transação tenha sido dado como abortada não é feito nada, caso contrário é simulado o tempo de escrita da transação, consumindo o recurso Disco.

6. Por último, no momento da resposta ao cliente, o simulador guarda o estado final da transação e o tempo que esta demorou a executar.

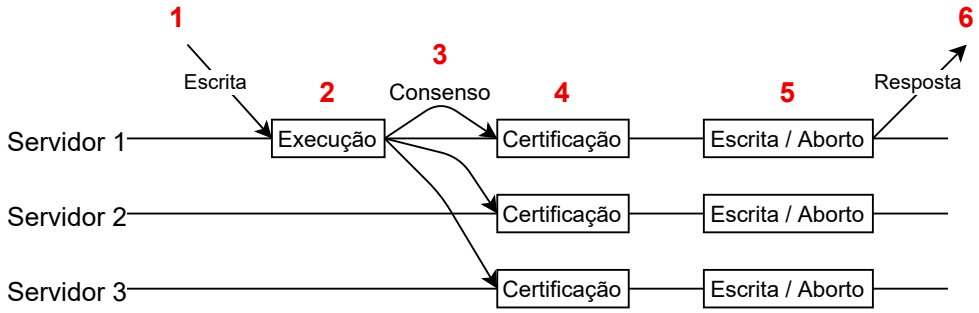


Figura 14: Troca de mensagens para realização de uma escrita em **DBSM**

Já para a realização de uma transação de leitura, o algoritmo varia conforme a certificação escolhida como argumento. Caso seja pretendido *serializability*, o processo é igual a uma escrita. Contudo, caso seja pretendido *snapshot isolation*, o processo pode ser observado na Figura 15 e é constituído pelos seguintes passos:

1. O gerador de carga gera uma transação de leitura e seleciona, aleatoriamente, um servidor para executar essa transação.

2. O servidor executa a leitura da transação, adquirindo os recursos correspondentes aos trincos e consumindo o recurso **CPU**. Posteriormente, após a simulação da execução por parte do **CPU**, são libertados os trincos.

3. No momento da resposta ao cliente, o simulador guarda o estado final da transação e o tempo que esta demorou a executar.

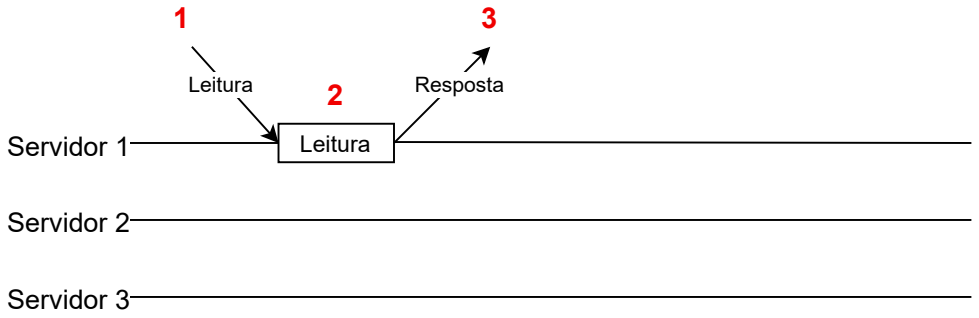


Figura 15: Troca de mensagens para realização de uma leitura em **DBSM**

Exemplo de código adaptado

De modo a demonstrar a implementação deste algoritmo no modelo de simulação, temos o exemplo da certificação realizada sequencialmente em cada fio de execução de terminação. Como podemos observar na Listagem 5, quando uma transação está a ser certificada, o seu conjuntos de operações de escrita ou de leitura (*write_set* ou *read_set*) será comparado com o conjunto de operações de escrita (*write_set*) de todas as transações concorrentes. A transação não é abortada se nenhum dos conjuntos de operações se intercetar.

```
aborted = False
curr_commit_size = self.server.len_commit_log()
for i in range(commit_size, curr_commit_size):
    ws = self.server.get_commit_log(i)

    if self.vars["CERTIFICATION"] == "SNAPSHOT_ISOLATION":
        if write_set.intersection(ws):
            aborted = True
            break

    if self.vars["CERTIFICATION"] == "SERIALIZABILITY":
        if read_set.intersection(ws):
            aborted = True
            break
```

Listagem 5: Processo de certificação.

Introdução de parâmetros

Os parâmetros de simulação para o modelo deste algoritmo podem ser encontrados na Tabela 4, junto com a sua explicação.

Comparativamente ao modelo de simulação genérico, podemos verificar que apenas temos uma variável para o número de servidores (*NUM_SERVERS*), em vez de número de centros de dados e número de servidores responsáveis por um fragmento dos dados, sendo que neste modelo não existe **fragmentação** nem geo-replicação. Devido à mesma razão, apenas existe uma variável referente à latência entre servidores (*NETWORK_LATENCY_SERVER*). Além disso, existem duas variáveis referentes à componente de certificação do algoritmo. A primeira, *CPU_CERTIFICATE*, corresponde ao tempo médio que o processador demora a certificar a transação e a segunda, *CERTIFICATION*, corresponde ao tipo de certificação

utilizada: *Snapshot Isolation* ou *Serializability*. Por último, existe uma variável referente ao número de discos de *Log* existentes, *LOG_DISKS*.

Identificador	Explicação
DATABASE_SIZE	Tamanho da base de dados
DATA_DISTRIBUITION	Distribuição dos dados
NUM_SERVERS	Número de servidores
CLIENT_POOL_SIZE	Número de fios de execução para atender clientes
SERVER_POOL_SIZE	Número de fios de execução para atender servidores
CPU_CORES	Número de cores **CPU**
IO_DISKS	Número de discos *IO*
LOG_DISKS	Número de discos *Log*
TRANSACTION_SIZE	Tamanho médio da transação
UPDATE_TRANSACTIONS	Distribuição de escritas e leituras
WRITE_IN_UPDATE	Percentagem de escritas em transações
ARRIVAL_MEAN	Taxa de chegada de pedidos
NETWORK_LATENCY_CLIENT	Latência entre servidor e cliente
NETWORK_LATENCY_SERVER	Latência entre servidores
CPU_EXECUTE	Tempo de execução de uma transação
CPU_READ	Tempo de leitura de uma transação
CPU_CERTIFICATE	Tempo de certificação de uma transação
IO_WRITE	Tempo de escrita em disco
NETWORK_CAPACITY	Capacidade máxima da rede
MAX_SIM_TIME	Tempo de simulação
CERTIFICATION	Certificação utilizada
SEED	Semente de aleatoriedade

Tabela 4: Parâmetros do modelo de simulação para **DBSM**

3.2.2 Spanner

Nesta secção apresentamos a configuração do modelo de simulação ao algoritmo *Spanner*. Primeiramente, o código executado por cada fio de execução é modificado de modo a simular o comportamento de execução de transações e troca de mensagens do algoritmo.

De modo a atribuir transações a servidores específicos como é possível no algoritmo *Spanner*, o código do gerador de carga é alterado para simular este comportamento.

Em cada servidor é adicionada uma tabela de trincos, como podemos ver na Figura 16. No simulador esta tabela é modelado por uma lista de recursos, cada um correspondente a um item da base de dados. Este recurso contém prioridade e desafetação forçada, de modo a simular o comportamento *wound wait* do sistema, onde transações com etiqueta temporal mais antigo interrompem os trincos de transações com etiqueta temporal mais recente [Spa23].

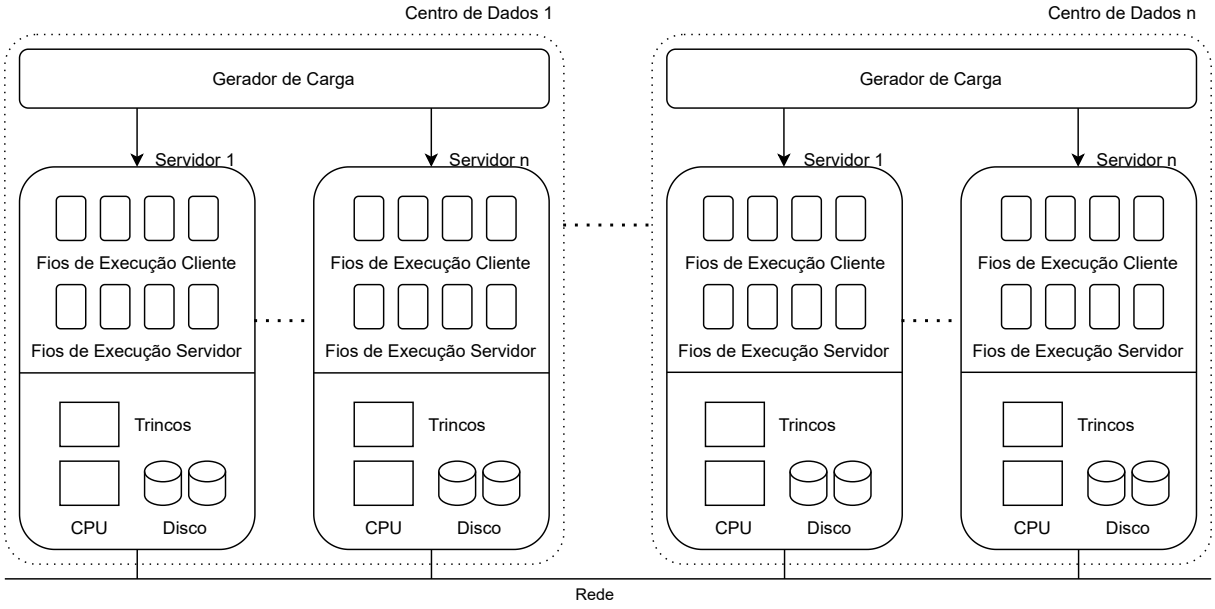


Figura 16: Arquitetura do modelo de simulação para *Spanner*

Troca de mensagens

Na Figura 17 podemos verificar a troca de mensagens entre os diversos servidores do sistema para o processo de uma transação de escrita na implementação do algoritmo *Spanner*, seguindo os seguintes passos:

1. O gerador de carga gera uma transação de escrita que poderá envolver um ou mais fragmentos de dados. Caso a transação apenas envolva um fragmento, esta é enviada ao líder do grupo

responsável por esse fragmento, passando o algoritmo para o passo 6. Caso contrário, é selecionado de todos os lideres dos grupos com os fragmentos envolvidos na transação um coordenador responsável por coordenar a operação, que irá então receber a transação.

2. O servidor coordenador irá atribuir um fio de execução para coordenar a transação, que irá enviar a todos os outros servidores líderes do grupo um pedido para o grupo adquirir os trincos relativos aos itens da transação.

3. Cada líder de grupo irá enviar aos servidores do mesmo grupo um pedido para adquirir então os trincos, adquirindo então cada elemento os recursos correspondentes ao trinco de cada item da transação.

4. Quando uma maioria dos servidores do grupo responderem ao líder a confirmar que adquiriram os trincos, este informa o coordenador que está preparado para executar a transação.

5. Quando o coordenador receber mensagem de todos os líderes a confirmar que estão prontos, este envia de volta uma mensagem a pedir para aplicar a transação.

6. O líder de cada grupo informa os servidores do grupo que podem aplicar os dados.

7. Cada servidor aplica então os dados localmente, consumindo os recursos **CPU** e Disco. Além disso, após aplicar os dados, são libertados os trincos.

8. Por último, o coordenador guarda o tempo de execução da transação.

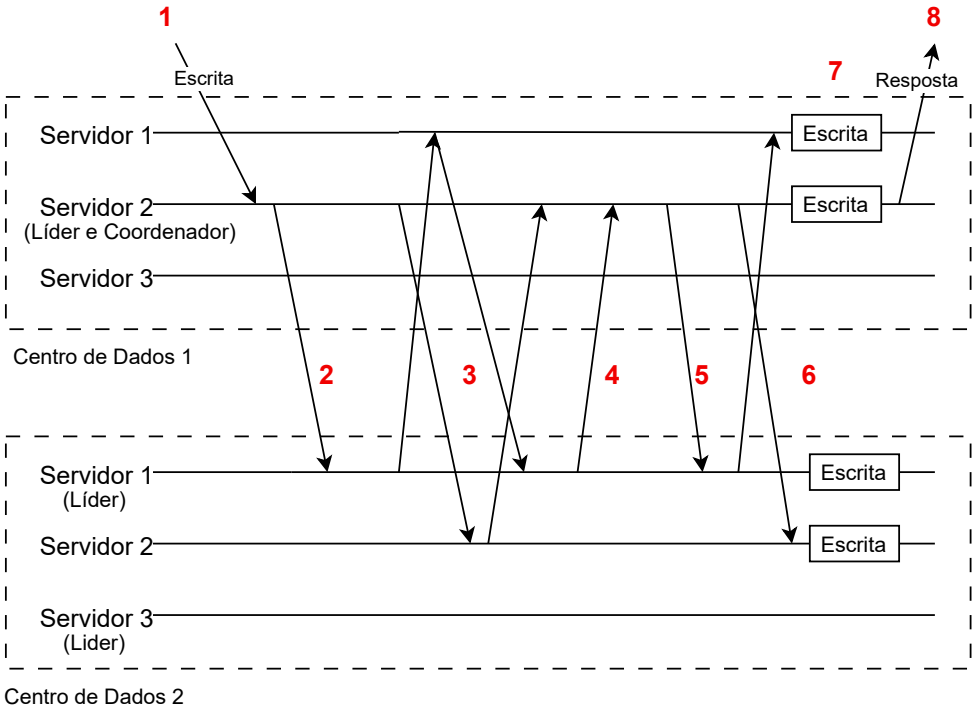


Figura 17: Troca de mensagens para realização de uma escrita em *Spanner*

Na Figura 18 podemos observar a troca de mensagens envolvida numa transação de leituras na implementação do algoritmo *Spanner*. É de notar que no algoritmo *Spanner* as transações de leitura apenas abordam dados de um fragmento. O processo é composto pelos seguintes passos:

1. Primeiramente, o gerador de carga gera uma transação e envia-a qualquer um dos servidores responsável pelo fragmento dos dados abordada por esta. Caso o servidor seja o líder do grupo, é avançado para o passo 4, realizando as leituras diretamente.

2. O servidor irá atribuir um fio de execução à transação que irá pedir ao líder o etiqueta temporal mais recente dos dados que pretende ler.

3. O líder responde com os etiquetas temporais de todos os itens da transação.

4. O servidor irá atribuir um novo fio de execução que irá comparar os etiquetas temporais enviados pelo líder com os etiquetas temporais que tem localmente, para todos os dados da transação. Caso estes estejam atuais, efetua a leitura de todos os itens, consumindo tempo do recurso **CPU**. Caso contrário espera receber a atualização do líder para o poder fazer.

5. Por último, guarda o tempo de execução da transação no momento da resposta ao cliente.

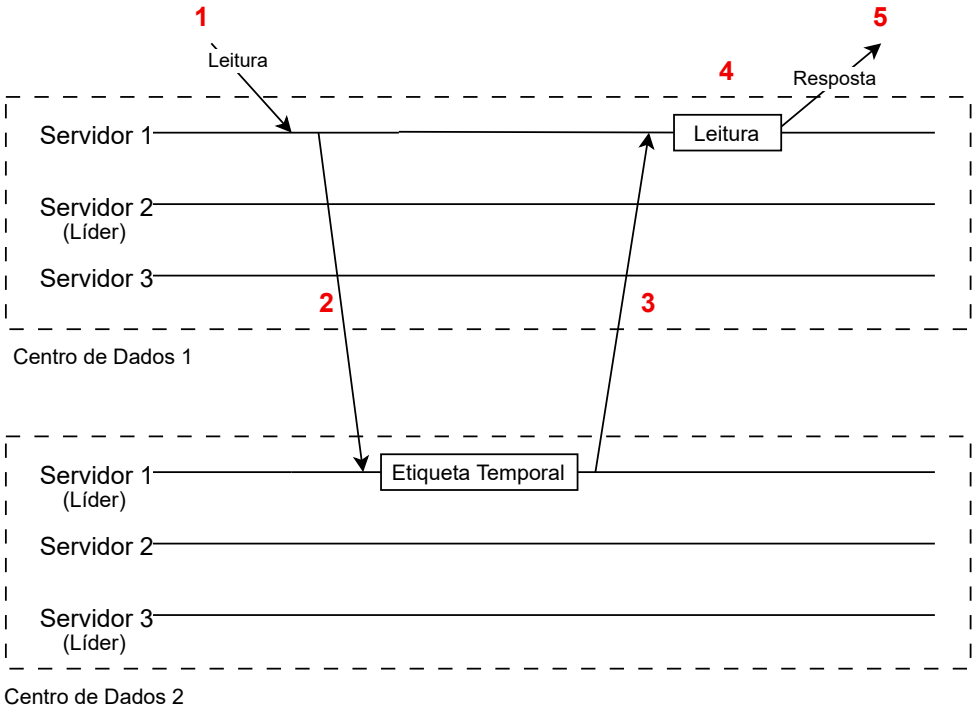


Figura 18: Troca de mensagens para realização de uma leitura em *Spanner*

Exemplo de código adaptado

Para demonstrar um exemplo de configuração do modelo a este algoritmo, temos o exemplo de adquirir os trincos de uma transação. Como podemos observar na Listagem 6, primeiramente definimos a prioridade da transação a adquirir os trincos, que é definida pelo etiqueta temporal da transação e pelo identificador do fio de execução, em caso de empate. Posteriormente, é pedido o trinco com preemptividade, caso este não esteja bloqueado e a prioridade definida em cima. Por último, após a aquisição do trinco, é bloqueada a preemptividade, caso seja definido como argumento.

```
def aquire_locks(self, transaction, msg_id, timestamp, block=False):
    variables = self.order_locks(transaction)
    priority = timestamp * self.vars["MAX_SIM_TIME"] + self.tid

    for var in variables:
        lock = self.split.variable_locks[var][0].request(
            preempt=self.split.variable_locks[var][1], priority=priority
        )
        if block:
            self.split.variable_locks[var][1] = False
        self.split.locks[msg_id][0][var] = lock
        yield lock
```

Listagem 6: Aquisição de trincos com desafeção forçada.

Introdução de parâmetros

Os parâmetros da simulação, bem como a sua descrição, podem ser encontrados na Tabela 5.

Em comparação com o modelo genérico, é adicionado o parâmetro *PARTITIONS*, que corresponde aos intervalos de partição de serviço para cada servidor.

Identificador	Explicação
DATABASE_SIZE	Tamanho da base de dados
DATA_DISTRIBUITION	Distribuição dos dados
NUM_ZONES	Número de centros de dados
NUM_SPLITS	Número de servidores por centro de dados
PARTITIONS	Partições de serviço
CPU_CORES	Número de cores **CPU**
IO_DISKS	Número de discos *IO*
TRANSACTION_SIZE	Tamanho médio da transação
TRANSACTION_DISTRIBUTION	Distribuição de escritas e leituras
ARRIVAL_MEAN	Taxa de chegada de pedidos
NETWORK_LATENCY_CLIENT	Latência entre servidor e cliente
NETWORK_LATENCY_INTRA_ZONE	Latência entre servidores no mesmo centro de dados
NETWORK_LATENCY_INTER_ZONE	Latência entre servidores em centros de dados diferentes

Identificador	Explicação
CPU_EXECUTE	Tempo de execução de uma transação
CPU_READ	Tempo de leitura de uma transação
IO_WRITE	Tempo de escrita em disco
NETWORK_CAPACITY	Capacidade máxima da rede
MAX_SIM_TIME	Tempo de simulação
SEED	Semente de aleatoriedade

Tabela 5: Parâmetros do modelo de simulação para *Spanner*

3.2.3 Wren

Nesta secção apresentamos a adaptação do o modelo de simulação do sistema de bases de dados *Wren*. Primeiramente, o código executado por cada fio de execução no servidor é modificado, de modo a garantir que os passos do algoritmo são simulados corretamente.

Como podemos observar na Figura 19, é adicionado um componente cliente, que é composto por um registo de transações efetuadas que tem o propósito de funcionar como *cache*. A *cache* do sistema *Wren* contém todas as escritas com etiqueta temporal superior à etiqueta temporal da última confirmação. Quando o cliente recebe um pedido de leitura do gerador de carga verifica na sua *cache* se contém o pedido e, caso contrário, envia o pedido ao fragmento coordenador. O gerador de carga é também alterado, de modo a que cada transação tem associado um cliente e um fragmento coordenador que irá receber a transação.

Por último, existem mais três processos a correr no servidor. Os dois primeiros são responsáveis por manter atualizados os relógios usados para gerar as etiquetas lst e rst. O terceiro é responsável por propagar as transações para os outros centros de dados.

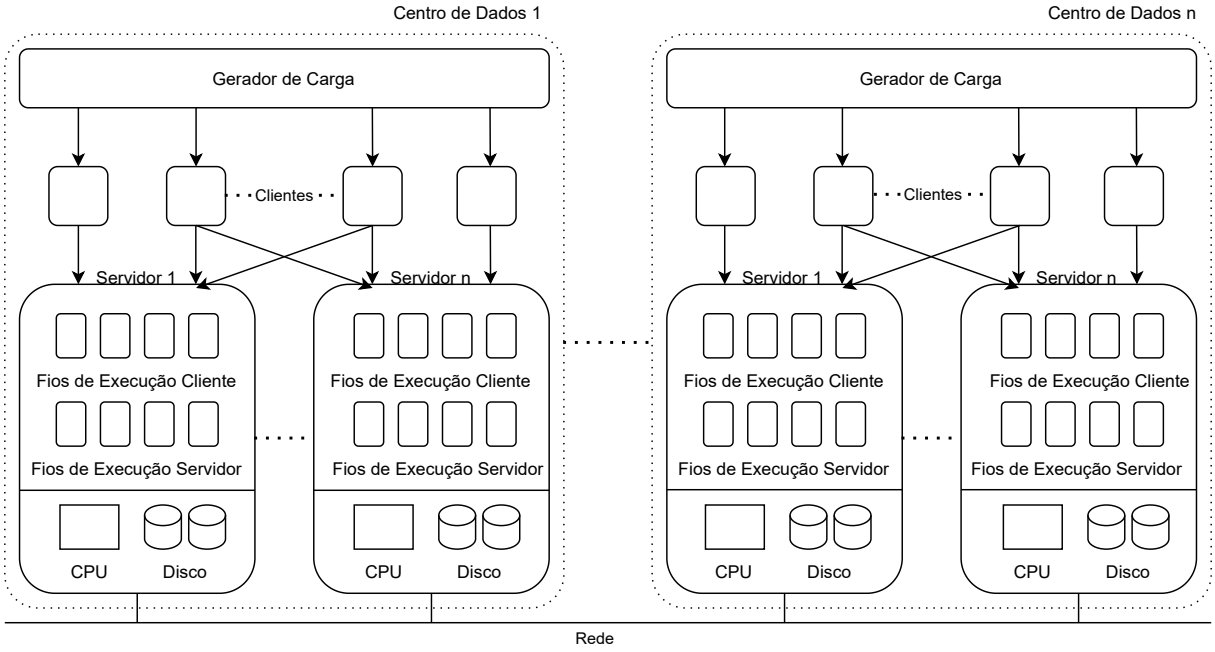


Figura 19: Arquitetura do modelo de simulação para *Wren*

Troca de mensagens

Na Figura 20 podemos verificar o processo de troca de mensagens relativo a uma transação de escrita na implementação do algoritmo *Wren*. O processo da transação de escrita está composto pelos seguintes passos:

1. Primeiramente, o gerador de carga gera uma transação de escrita e envia a transação a um servidor aleatório do centro de dados local que contenha um fragmento dos itens dessa transação.

2. Ao receber a transação, o servidor irá atribuir um fio de execução para pedidos do cliente e este vai enviar cada parte da transação ao servidor responsável por esse fragmento no centro de dados local.

3. Cada servidor irá atribuir um fio de execução para pedidos de servidor para executar a transação recebida, consumindo o recurso **CPU** e o recurso Disco.

4. Após o servidor que recebeu a transação inicialmente receber resposta de todos os servidores envolvidos na transação, este guarda o tempo de execução. da transação. Além disso, o etiqueta temporal da *cache* do cliente é alterada, truncando assim futuramente os itens da *cache* para o novo etiqueta temporal.

5. Posteriormente, após um determinado intervalo de tempo, os dados escritos são partilhados com os fragmentos correspondentes dos outros centros de dados.

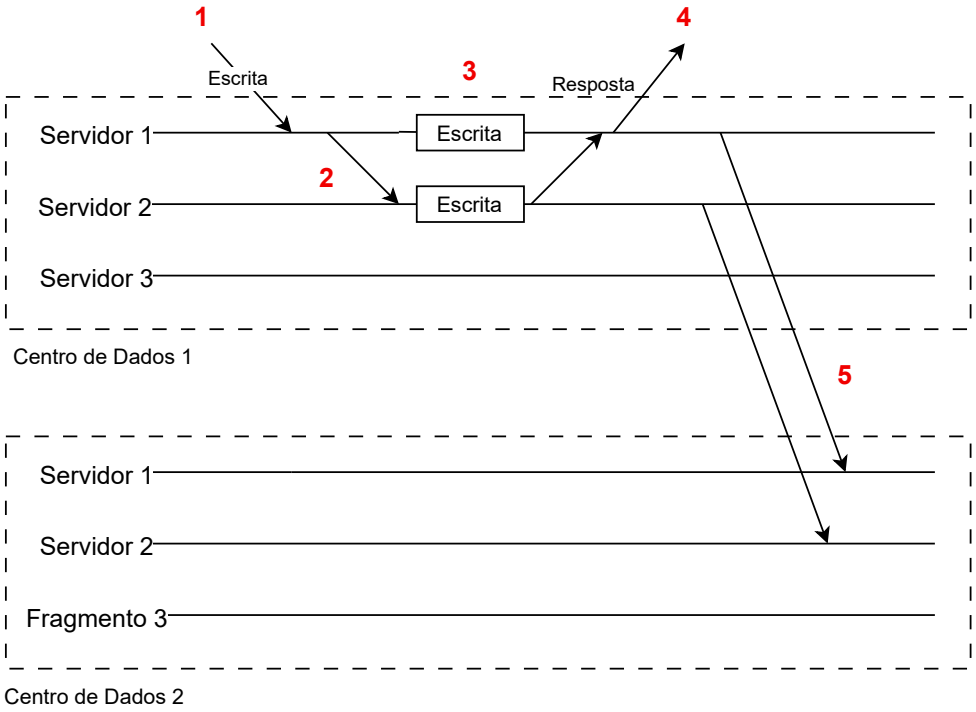


Figura 20: Troca de mensagens para realização de uma escrita em *Wren*

Na Figura 21 podemos observar o processo de troca de mensagens relativo a uma transação de leitura neste mesmo algoritmo, composto pelos seguintes passos:

1. Primeiramente, o gerador de carga gera uma transação de leitura e verifica na *cache* do cliente quais os itens que se intercetam. Caso todos os itens da transação estejam presentes em *cache*, o tempo total de execução é adicionado às estatísticas, não procedendo com o envio da mensagem ao servidor. Caso contrário, é enviada a transação com os itens não presentes na *cache* a um fragmento do centro de dados que tem guardados parte dos itens da transação.

2. Ao receber a transação, o fragmento irá atribuir um fio de execução para pedidos do cliente e este vai enviar cada parte da transação ao fragmento correspondente.

3. Cada fragmento irá atribuir um fio de execução para pedidos do servidor para realizar a leitura da transação recebida, consumindo assim o recurso **CPU**.

4. Após o fragmento coordenador receber resposta de todos os fragmentos envolvidos na transação é registado o tempo de resposta da mesma.

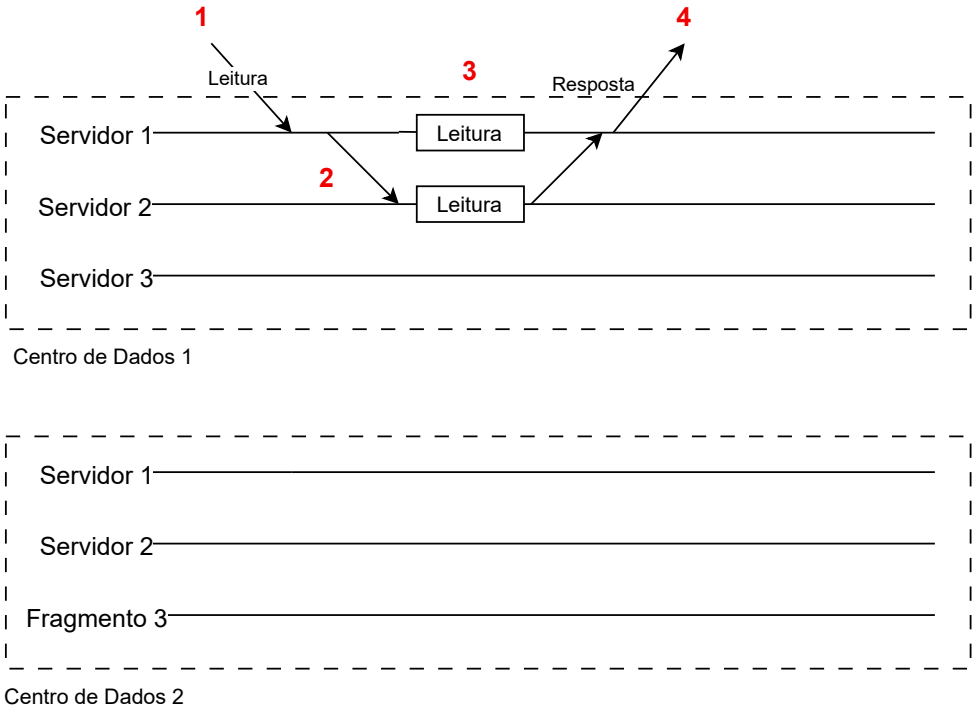


Figura 21: Troca de mensagens para realização de uma leitura em *Wren*

Exemplo de código adaptado

Para demonstrar a adaptação do modelo de simulação à base de dados *Wren*, temos como exemplo o processo que atualiza o *local stable timestamp*, como podemos observar na Listagem 7. Este processo irá esperar o tempo correspondente à propagação de uma mensagem dentro de um centro de dados, gerado através de uma distribuição de *Poisson*. Posteriormente, irá recolher os valores de relógio de todos os centros de dados e calcular o mínimo. Deste modo, simulamos o processo de atualização do *lst* sem ter de introduzir mensagens no sistema.

```
def lst_update(self):
    while True:
        rtt = self.datacenter.generator.generate_timeout(
            self.vars["NETWORK_LATENCY_INTRA_DC"]
        )
        yield self.env.timeout(rtt)

        clocks = list()
        for s in self.datacenter.shards.values():
            clocks.append(s.clock)

        self.lst = min(clocks)
```

Listagem 7: Processo de atualização do lst.

Introdução de parâmetros

Os parâmetros da simulação, bem como a sua descrição, podem ser encontrados na Tabela 6.

Em comparação com o modelo genérico, é adicionado o parâmetro *PARTITIONS*, que corresponde aos intervalos de partição de serviço para cada servidor. Além disso, é adicionado o parâmetro *COMMIT_INTERVAL_SERVER*, que é referente ao intervalo em que cada servidor propaga as alterações com os outros centros de dados. Por último, são adicionados os parâmetros *NUM_CLIENTS_DC* e *CLIENT_CACHE*, que correspondem ao número de clientes por centros de dados e a existência ou não de *cache* no sistema, respetivamente.

Identificador	Explicação
DATABASE_SIZE	Tamanho da base de dados
DATA_DISTRIBUITION	Distribuição dos dados
NUM_DATA_CENTERS	Número de centros de dados
NUM_SHARDS	Número de fragmentos
PARTITIONS	Partições de serviço
CLIENT_POOL_SIZE	Número de fios de execução para atender clientes
SERVER_POOL_SIZE	Número de fios de execução para atender servidores
COMMIT_INTERVAL_SERVER	Intervalo entre commits
CPU_CORES	Número de cores **CPU**

Identificador	Explicação
IO_DISKS	Número de discos *IO*
TRANSACTION_SIZE	Tamanho médio da transação
TRANSACTION_DISTRIBUTION	Distribuição de escritas e leituras
TRANSACTION_PARTITIONS	Número de partições por transação
ARRIVAL_MEAN	Taxa de chegada de pedidos
NUM_CLIENTS_DC	Número de clientes por centro de dados
CLIENT_CACHE	Existencia de *cache* para os clientes
NETWORK_LATENCY_CLIENT	Latência entre servidor e cliente
NETWORK_LATENCY_INTRA_DC	Latência entre servidores no mesmo centro de dados
NETWORK_LATENCY_INTER_DC	Latência entre servidores em centros de dados diferentes
CPU_EXECUTE	Tempo de execução de uma transação
CPU_READ	Tempo de leitura de uma transação
IO_WRITE	Tempo de escrita em disco
NETWORK_CAPACITY	Capacidade máxima da rede
MAX_SIM_TIME	Tempo de simulação
SEED	Semente de aleatoriedade

Tabela 6: Parâmetros do modelo de simulação para *Wren*

Capítulo 4

Avaliação experimental

Neste capítulo iremos apresentar as diversas experiências efetuadas com o modelo de simulação *SAGeo*, com o objetivo de validar o modelo de simulação e comparar o desempenho de diversas implementações do modelo.

Os testes foram realizados numa máquina com 1 servidor com um processador *Intel Xeon E5-2698 v4* com 20 núcleos e uma frequência de 2.20GHz. Quanto à memória, o servidor tem 32GB de **RAM** a uma frequência de 3600MHz e um disco com 360GB. Esta máquina tem como sistema operativo o *Rocky Linux 8* e utiliza a versão 3.11 do *Python*.

4.1 Validação do modelo

Nesta secção analisamos a avaliação experimental do modelo de simulação de bases de dados configurado para o algoritmo *Wren*, de modo a validar os seus resultados. Este teste foi realizado comparando o desempenho do algoritmo com e sem a utilização de *cache* e comparando o comportamento da nossa implementação com os resultados no artigo que introduz o algoritmo.

4.1.1 Ambiente simulado

Para esta simulação consideramos 3 centros de dados com uma latência média de 100 ms entre eles. Cada centro de dados está dividido em 8 servidores, sendo a latência considerada entre servidores do mesmo centro de dados 2 ms. Cada servidor tem um cliente associado, sendo este servidor responsável por coordenar os pedidos do cliente. Em relação ao desempenho dos servidores, consideramos um tempo médio de execução de **CPU** de 80 μs, um tempo médio de leitura de 50μs e um tempo médio de escrita em disco de 90 μs.

Nestes testes, consideramos um tamanho médio de transação de 12 itens, espalhados por 4 servidores. Em relação à distribuição das leituras e escritas, variamos a carga de trabalho entre 90:10 (90%

leituras e 10% escritas), correspondentes a uma utilização de leituras pesadas, e 50:50 (50% leituras e 50% escritas), correspondente a uma utilização de escritas pesadas.

Ambos os testes são executados variando a taxa de chegada de pedidos por centro de dados entre 1250 transações\s e 40000 transações\s.

A primeira simulação demorou no total 14 minutos e a segunda simulação demorou no total 20 minutos.

4.1.2 Resultados

A nossa experiência tem por objetivo comparar o desempenho de um algoritmo com coerência causal transacional que utiliza *cache*, com um que não a utiliza, perante diferentes distribuições de carga de trabalho.

Como podemos observar pela Figura 22, quando estamos perante um sistema com uma utilização pesada de leituras, o algoritmo com utilização de *cache* garante tempos de resposta inferiores e débito superior ao algoritmo que não utiliza, devido ao facto de muitas destes valores estarem presentes na *cache*, não sendo necessário fazer o pedido ao servidor. Além disso, podemos verificar que inicialmente ao aumentar o número de pedidos à base de dados o tempo de resposta diminui. Este caso deve-se ao facto de quantos mais pedidos estiverem a ser feitos à base de dados, maior será a *cache*, aumentando a probabilidade do item se encontrar lá. Apenas quando os recursos da base de dados se esgotam é que o tempo de resposta aumenta.

Na Figura 23 podemos observar que num sistema mais pesado em escritas o beneficio da *cache* não é tão notável. Contudo, o *Wren* com *cache* continua com um maior débito e com menor tempo de resposta do que a versão sem *cache*.

Os resultados destes dois testes com o *SAGeo* podem ser comparados aos resultados incluídos no artigo que apresenta o *Wren* com um protótipo num sistema real [SDZ18], em que o sistema com coerência causal transacional que não utiliza *cache* é representado pelo algoritmo Cure e o que utiliza é representado pelo próprio *Wren*. Comparando os nossos resultados com esses, verificamos que a simulação reproduz a vantagem relativa da *cache*.

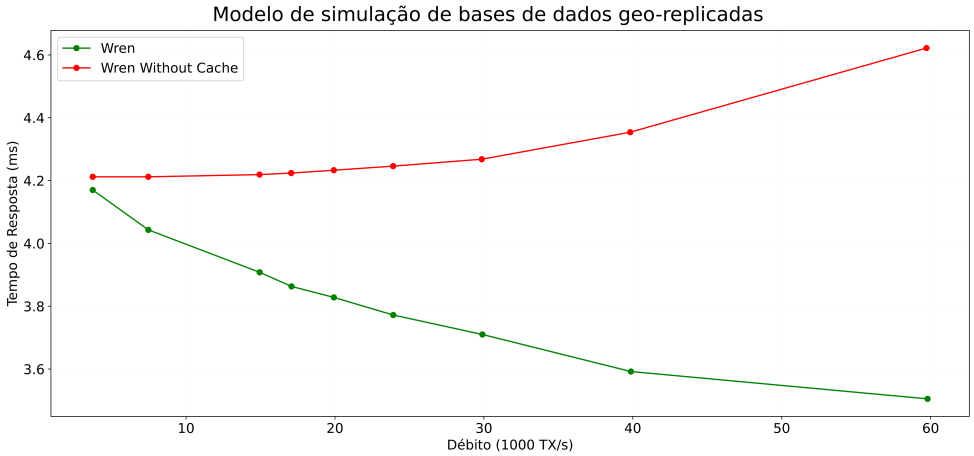


Figura 22: Comparação entre *Wren* com e sem *cache* com carga de trabalho 90:10

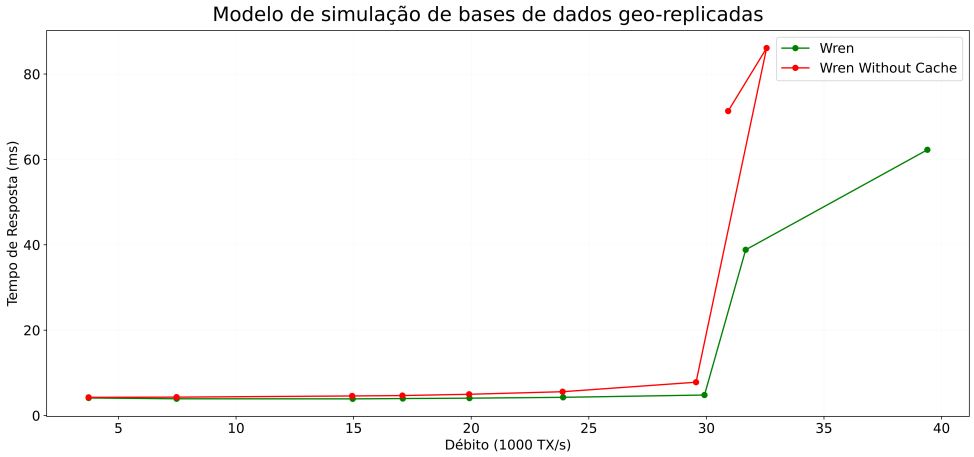


Figura 23: Comparação entre *Wren* com e sem *cache* com carga de trabalho 50:50

4.2 Comparação de Bases de Dados CP

Nesta secção analisamos a avaliação experimental do modelo de simulação de bases de dados configurado para o algoritmo **DBSM** e para o algoritmo *Spanner*. Este teste foi realizado de modo a comparar os benefícios e as desvantagens da utilização de geo-replicação de dados, em comparação com **replicação** dentro de apenas um centro de dados.

4.2.1 Ambiente simulado

Nesta simulação, para o algoritmo de bases de dados **DBSM**, consideramos que no centro de dados estariam presentes 3 servidores, com uma latência média entre eles de 2 ms. Consideramos também uma latência média entre o servidor e o cliente de 3 ms. Em relação ao desempenho dos servidores,

consideramos um tempo médio de execução de **CPU** de 30 μs, um tempo médio de leitura de 15 μs, um tempo de certificação da transação de 50 μs e um tempo médio de escrita em disco de 50 μs. Consideramos que a base de dados teria 3000 itens e uma carga de trabalho fixa 60:40 (60% leituras e 40% escritas), em que cada transação tem em média 10 itens, que seguem uma distribuição *Zipfian*. Em relação ao tipo de certificação utilizamos *Snapshot Isolation*.

Já em relação ao algoritmo de bases de dados *Spanner*, utilizamos duas configurações diferentes. Nas duas configurações consideramos, em relação ao desempenho dos servidores, ao número de itens da base de dados, à latência entre servidor e cliente e ao tamanho da transação utilizamos as mesmas configurações que o **DBSM**. Consideramos que as transações seguiriam também uma carga de trabalho fixa 60:40 (60% leituras, 30% escritas num servidor e 10% escritas em vários servidores). Para a primeira configuração consideramos 3 centros de dados, cada um com 3 servidores, em que a latência entre centros de dados é de 100 ms e a latência entre servidores do mesmo centro de dados é de 2ms. Já na segunda configuração consideramos na mesma 3 centros de dados, cada um com 3 servidores, mas em que tanto a latência entre centros de dados e a latência entre servidores é de 2ms.

As duas configurações do *Spanner* foram executadas em comparação com a configuração do **DBSM**, variando a taxa de chegada de pedidos entre 500 transações\s e 10000 transações\s em cada centro de dados.

A primeira simulação demorou no total 22 minutos e a segunda simulação demorou no total 21 minutos.

4.2.2 Resultados

A nossa experiência tem por objetivo comparar o desempenho de um algoritmo **CP** que não utiliza geo-replicação com um que utiliza, perante diferentes taxas de chegada de pedidos. Além disso, pretende-se comparar o efeito que a execução otimista e pessimista destes algoritmos tem no tempo de execução e débito.

Como podemos observar na Figura 24, onde são apresentados os resultados da execução do modelo de simulação para o **DBSM** e o *Spanner* com latências de geo-replicação, o algoritmo **DBSM** apresenta, inicialmente, tempos de resposta bastante inferiores ao *Spanner*, devido ao facto de no primeiro a latência entre servidores ser de apenas de 2ms. Além disso, como a execução do **DBSM** é otimista, ou seja, as transações são executadas no servidor que recebe a transação e posteriormente certificadas nos outros, sem necessidade de recorrer a **locking** distribuído, o débito deste algoritmo é inicialmente superior ao *Spanner*. Contudo, como as transações têm de ser certificadas sequencialmente por cada servidor,

existe um **gargalo** deste fio de execução, comprometendo a escalabilidade do sistema **DBSM**. Já o algoritmo *Spanner*, apesar de pior desempenho perante baixas taxas de chegada de pedidos, apresenta uma escalabilidade muito superior.

Na Figura 25 podemos observar os resultados da execução destes dois algoritmos sem latências de geo-replicação, ou seja, o algoritmo *Spanner* comporta-se como se todos os seus servidores estivessem no mesmo centro de dados. Neste caso em particular, podemos verificar que o desempenho superior do **DBSM** em relação ao tempo de resposta já não é significativo, comparativamente aos resultados da Figura 24. Contudo, é possível verificar na mesma que a execução otimista do algoritmo **DBSM** permite débitos superiores perante taxas de chegadas de pedidos mais pequenas, mas perde em escalabilidade comparativamente ao algoritmo *Spanner*.

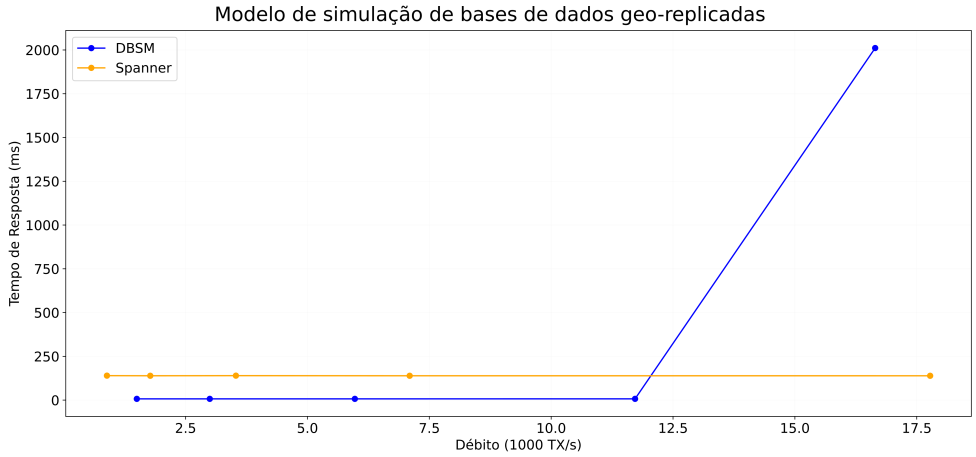


Figura 24: Comparação entre **DBSM** e *Spanner* com latências de geo-replicação

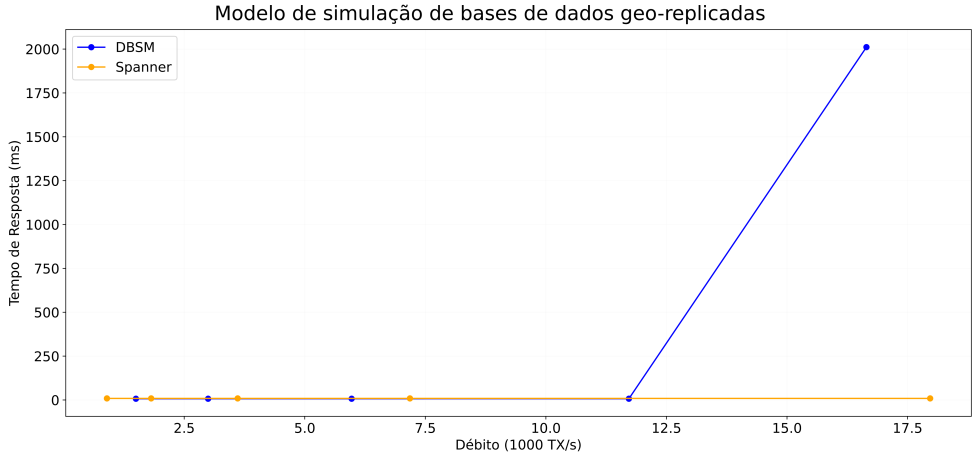


Figura 25: Comparação entre **DBSM** e *Spanner* sem latências de geo-replicação

4.3 Comparação de Bases de Dados Geo-Replicadas

Nesta secção analisamos a avaliação experimental do modelo de simulação de bases de dados configurado para o algoritmo *Spanner* e para o algoritmo *Wren*. Este teste foi realizado de modo a comparar as vantagens e as desvantagens dos sistemas **AP** e **CP**, perante diferentes latências entre centros de dados e perante partições.

4.3.1 Ambiente simulado

Nesta simulação, numa primeira configuração consideramos, tanto para o algoritmo de bases de dados *Spanner* como para o *Wren*, 3 centros de dados, cada um com 3 servidores, em que a latência dentro do centro de dados é de 2ms. Consideramos também uma latência média entre o servidor e o cliente de 3ms. Em relação ao desempenho dos servidores consideramos um tempo médio de execução de **CPU** de 30 μs, um tempo médio de leitura de 15 μs e um tempo médio de escrita em disco de 50 μs. Consideramos 3000 itens na base de dados, e que as transações seguiriam uma carga de trabalho fixa 60:40 (60% leituras, 30% escritas num servidor e 10% escritas em vários servidores). Cada transação teria em média 10 itens, que seguem uma distribuição *Zipfian*.

Numa segunda configuração consideramos todos os parâmetros descritos em cima, com o acréscimo de partições no intervalo de simulação 2000000 a 3000000 para o servidor 1 do centro de dados 1, 1000000 a 2000000 para o servidor 1 do centro de dados 2 e 8000000 a 9000000 para o servidor 1 do centro de dados 3.

Foram então executados dois testes, comparando ambas as configurações do algoritmo de *Spanner* e do algoritmo de *Wren* (com e sem partições), variando a latência entre centros de dados entre 10ms e 200ms.

A primeira simulação demorou no total 8 minutos e a segunda simulação demorou no máximo 17 minutos.

4.3.2 Resultados

Esta experiência tem como objetivo comparar o desempenho de um algoritmo **CP** e de um algoritmo **AP** perante diferentes latências entre centros de dados e perante partições.

Como podemos observar na Figura 26, onde estão presentes os resultados da execução do modelo de simulação variando as latências entre centro de dados e sem presença de partições, o algoritmo de *Wren* apresenta sempre os mesmos resultados a nível de tempo de resposta e débito, devido ao facto

de se tratar de um sistema **AP** que responde ao cliente antes de propagar as alterações da transação para os outros centros de dados. Já o algoritmo *Spanner*, um sistema **CP**, realiza **locking** distribuído entre centros de dados para aplicar uma transação e só responde ao cliente após os servidores de todos os centros de dados aplicarem a transação. Devido a este comportamento, podemos verificar que, para este algoritmo, o aumento da latência entre servidores contribui para o aumento do tempo de resposta e uma diminuição do débito. Além disso, comparando diretamente o débito do algoritmo *Spanner* com o algoritmo *Wren*, podemos verificar que o primeiro nunca consegue atingir valores tão altos como o segundo, mesmo perante latências entre centros de dados muito baixas. Este resultamos mostram as principais vantagens dos sistemas **AP** comparativamente aos sistemas **CP**.

Na Figura 27, podemos observar o mesmo teste realizado anteriormente mas com presença de partições. Nesta execução, podemos verificar que o tempo de resposta do algoritmo *Spanner* aumenta linearmente comparativamente à execução sem partições, devido ao facto das transações ficarem em espera que a partição acabe para este ser concluída. No algoritmo *Wren* o comportamento é diferente, devido ao facto de este algoritmo permitir utilização de *cache*, sendo o tempo de execução mais baixo quanto maior for a latência entre centros de dados. A combinação de uma partição de rede com uma longa latência entre servidores significa que os servidores não irão aumentar o seu *rst*. Assim, o cliente quando fizer um pedido ao servidor irá verificara que tem o *remote timestamp* mais recente e irá realizar a leitura dos itens da sua *cache* em vez de fazer o pedido ao servidor. Este comportamento é, em parte, uma limitação do modelo de simulação porque o tamanho da *cache* não é limitado, garantindo que enquanto houver uma partição o cliente vai estar a adicionar mais itens à sua *cache* reduzindo assim o tempo de execução da transação. Contudo, mesmo com uma *cache* limitada, apesar do tempo de resposta ser mais baixo este comportamento não é desejável, porque as réplicas irão estar constantemente a divergir, sendo esta uma das maiores desvantagens dos sistemas **AP** comparativamente aos sistemas **CP**.

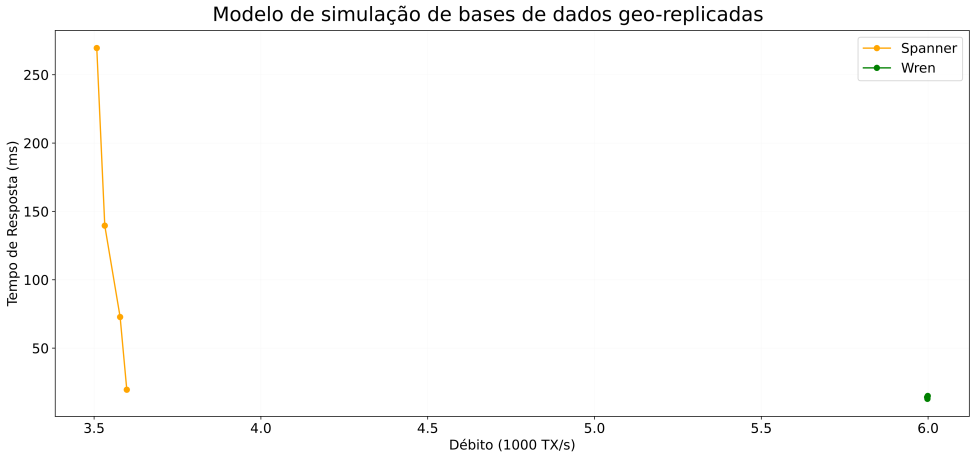


Figura 26: Comparação entre *Spanner* e *Wren* sem partições

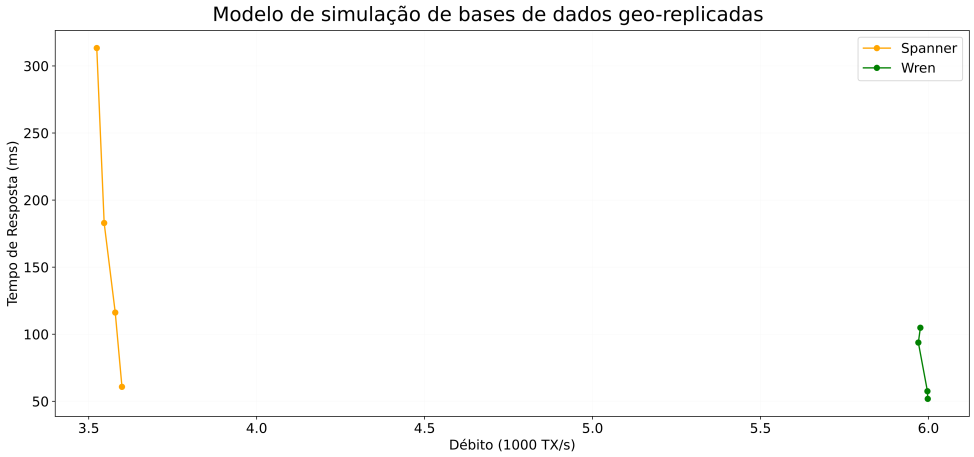


Figura 27: Comparação entre *Spanner* e *Wren* com partições

Capítulo 5

Conclusão

Com o objetivo de avaliar e comparar o desempenho de diversos algoritmos de bases de dados, bem como validar os algoritmos e ajudar a encontrar problemas que estes possam ter associados, esta tese desenvolve o *SAGeo*, um modelo de simulação configurável de bases de dados distribuídas geo-replicadas.

Deste modo, o simulador foi desenvolvido de uma forma modular, ou seja, certos componentes do modelo podem ser alterados de modo a poder ser adaptado a diferentes algoritmos de bases de dados. Este modelo de simulação recorre ao simulador de eventos discretos que utiliza o paradigma orientado ao processo para *Python*: *SimPy* [Sim22].

A utilidade deste modelo é inicialmente avaliada com uma configuração para o sistema de bases de dados *Wren* [SDZ18]. Nesta execução conseguimos reproduzir os resultados originais relativos à vantagem de utilização de *cache* no cliente, uma das características principais do algoritmo, validando então a sua funcionalidade.

A configuração do modelo de simulação para os sistemas de bases de dados **DBSM** [PGS03] e *Spanner* [CDE+13] permite comparar o desempenho dos dois sistemas. Com estas execuções conseguimos verificar as vantagens da execução otimista do primeiro algoritmo, bem como os seus problemas de escalabilidade, comparativamente com o *Spanner*. Além disso, foi possível verificar as diferenças em tempo de resposta quando é utilizada geo-replicação em vez de apenas um centro de dados.

Por último, utilizamos as configurações do modelo de simulação para os sistemas *Wren* [SDZ18] e *Spanner* [CDE+13] para avaliar as diferenças entre sistemas que dão prioridade à consistência dos dados (sistemas **CP**) e sistemas que dão prioridade à disponibilidade (sistemas **AP**). Com estes testes conseguimos verificar que os algoritmos que dão prioridade à consistência dos dados têm uma escalabilidade muito inferior perante latências crescentes entre centros de dados. Conseguimos verificar que estes sistemas também manifestam pior desempenho perante partições, comparativamente com sistemas que dão prioridade à disponibilidade.

Futuramente, o modelo de simulação *SAGeo* será configurado para mais sistemas, de modo a pode-

rem ser comparadas diferentes abordagens utilizadas por diferentes sistemas e avaliar os compromissos de cada um, podendo fazer uma melhor avaliação de cada sistema para a implementação em que este será utilizado. Além disso, o modelo de simulação será utilizado para executar mais simulações sobre os sistemas já configurados, de modo a continuar a avaliar o seu desempenho perante diferentes ambientes e parâmetros.

Bibliografia

[ATB+16] Deepthi Devaki Akkoorath, Alejandro Z Tomsic, Manuel Bravo, Zhongmiao Li, Tyler Crain, Annette Bieniusa, Nuno Preguiça, and Marc Shapiro. Cure: Strong semantics meets high availability and low latency. In *2016 IEEE 36th International Conference on Distributed Computing Systems (ICDCS)*, pages 405–414, June 2016.

[Bre12] Eric Brewer. CAP twelve years later: How the "rules"have changed. *Computer*, 45(2):23–29, 2012.

[BRVR17] Manuel Bravo, Luís Rodrigues, and Peter Van Roy. Saturn: a distributed metadata service for causal consistency. In *Proceedings of the Twelfth European Conference on Computer Systems*, EuroSys '17, pages 111–126, New York, NY, USA, April 2017. Association for Computing Machinery.

[CDE+13] James C Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, Jeffrey John Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, et al. Spanner: Google's globally distributed database. *ACM Transactions on Computer Systems (TOCS)*, 31(3):1–22, 2013.

[CGL+14] Henri Casanova, Arnaud Giersch, Arnaud Legrand, Martin Quinson, and Frédéric Suter. Versatile, Scalable, and Accurate Simulation of Distributed Applications and Platforms. *Journal of Parallel and Distributed Computing*, 74(10):2899–2917, June 2014.

[CST+10] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, page 143–154, New York, NY, USA, 2010. Association for Computing Machinery.

[DEZ13] Jiaqing Du, Sameh Elnikety, and Willy Zwaenepoel. Clock-SI: Snapshot isolation for partitioned data stores using loosely synchronized clocks. In *2013 IEEE 32nd International Symposium on Reliable Distributed Systems*, pages 173–184, September 2013.

[DHJ+07] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: amazon's highly available key-value store. *Oper. Syst. Rev.*, 41(6):205–220, October 2007.

[Edu23] IBM Cloud Education. What is the cap theorem?, 2023.

[FCPL22] P. Fouto, P. Costa, N. Preguica, and J. Leitao. Babel: A framework for developing performant and dependable distributed protocols. In *2022 41st International Symposium on Reliable Distributed Systems (SRDS)*, pages 146–155, Los Alamitos, CA, USA, sep 2022. IEEE Computer Society.

[Fuj90] Richard M. Fujimoto. Parallel discrete event simulation. *Commun. ACM*, 33(10):30–53, oct 1990.

[GHOS96a] Jim Gray, Pat Helland, Patrick O'Neil, and Dennis Shasha. The dangers of replication and a solution. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, SIGMOD '96, page 173–182, New York, NY, USA, 1996. Association for Computing Machinery.

[GHOS96b] Jim Gray, Pat Helland, Patrick O'Neil, and Dennis Shasha. The dangers of replication and a solution. *SIGMOD Rec.*, 25(2):173–182, jun 1996.

[GL02] Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2):51–59, June 2002.

[GL12] Seth Gilbert and Nancy Lynch. Perspectives on the cap theorem. *Computer*, 45(2):30–36, 2012.

[GNS+20] Paulo Gouveia, João Neves, Carlos Segarra, Luca Liechti, Shady Issa, Valerio Schiavoni, and Miguel Matos. Kollaps: decentralized and dynamic topology emulation. In Angelos Bilas, Kostas Magoutis, Evangelos P. Markatos, Dejan Kostic, and Margo I. Seltzer, editors, *EuroSys '20: Fifteenth EuroSys Conference 2020, Heraklion, Greece, April 27-30, 2020*, pages 23:1–23:16. ACM, 2020.

[GSTH08] Donald Gross, John F. Shortle, James M. Thompson, and Carl M. Harris. *Fundamentals of Queueing Theory*. John Wiley & Sons, 4 edition, 2008.

[HLC96] San-Yih Hwang, Keith KS Lee, and YH Chin. Data replication in a distributed system: A performance study. In *International Conference on Database and Expert Systems Applications*, pages 708–717. Springer, 1996.

[JPR+10] Afrâneo Correia Jr., José Pereira, Luís Eduardo Teixeira Rodrigues, Nuno Carvalho, and Rui Oliveira. *Practical Database Replication*, volume 5959. Springer, 2010.

[KA20] Kyle Kingsbury and Peter Alvaro. Elle: Inferring isolation anomalies from experimental observations. *Proc. VLDB Endow.*, 14(3):268–280, nov 2020.

[Kle16] Martin Kleppmann. How to do distributed locking, 2016.

[LAdS+15] Ignacio Laguna, Dong H. Ahn, Bronis R. de Supinski, Todd Gamblin, Gregory L. Lee, Martin Schulz, Saurabh Bagchi, Milind Kulkarni, Bowen Zhou, Zhezhe Chen, and Feng Qin. Debugging high-performance computing applications at massive scales. *Commun. ACM*, 58(9):72–81, August 2015.

[Law15] Averill M Law. *Simulation Modeling and Analysis*. McGraw-Hill Education, 2015.

[LFKA11] Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. Don't settle for eventual: Scalable causal consistency for wide-area storage with cops. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, page 401–416, New York, NY, USA, 2011. Association for Computing Machinery.

[Liu20] Jason Liu. Simulus: Easy breezy simulation in python. In *2020 Winter Simulation Conference (WSC)*, pages 2329–2340, 2020.

[LSS+17] Tanakorn Leesatapornwongsa, Cesar A. Stuardo, Riza O. Suminto, Huan Ke, Jeffrey F. Lukman, and Haryadi S. Gunawi. Scalability bugs: When 100-node testing is not enough. In *HotOS '17*, pages 24–29. ACM, 2017.

[Mat08] Norm Matloff. Introduction to discrete-event simulation and the simpy language. *Davis, CA. Dept of Computer Science. University of California at Davis. Retrieved on August*, 2(2009):1–33, 2008.

[MKK+10] Jason E. Miller, Harshad Kasture, George Kurian, Charles Gruenwald, Nathan Beckmann, Christopher Celio, Jonathan Eastep, and Anant Agarwal. Graphite: A distributed parallel

simulator for multicores. In *HPCA - 16 2010 The Sixteenth International Symposium on High-Performance Computer Architecture*, pages 1–12, 2010.

[NJ00] M. Nicola and M. Jarke. Performance modeling of distributed and replicated databases. *IEEE Transactions on Knowledge and Data Engineering*, 12(4):645–672, 2000.

[PGS03] Fernando Pedone, Rachid Guerraoui, and André Schiper. The database state machine approach. *Distributed and Parallel Databases*, 14, 07 2003.

[RGG03] Raghu Ramakrishnan, Johannes Gehrke, and Johannes Gehrke. *Database management systems*, volume 3. McGraw-Hill New York, 2003.

[RTH96] Jing Fei Ren, Yutaka Takahashi, and Toshiharu Hasegawa. Analysis of impact of network delay on multiversion conservative timestamp algorithms in DDBS. *Performance Evaluation*, 26(1):21–50, 1996.

[Sch86] Herb Schwetman. CSIM: A C-based, process-oriented simulation language. *Winter Simulation Conference*, 1986.

[Sch90a] Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Comput. Surv.*, 22(4):299–319, dec 1990.

[Sch90b] Herbert D. Schwetman. Introduction to process-oriented simulation and CSIM. *Winter Simulation Conference*, 1990.

[SDQR10] Jörg Schad, Jens Dittrich, and Jorge-Arnulfo Quiané-Ruiz. Runtime measurements in the cloud: Observing, analyzing, and reducing variance. *Proc. VLDB Endow.*, 3(1-2):460–471, September 2010.

[SDZ18] Kristina Spirovska, Diego Didona, and Willy Zwaenepoel. Wren: Nonblocking reads in a partitioned transactional causally consistent data store. In *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 1–12, June 2018.

[SDZ19] Kristina Spirovska, Diego Didona, and Willy Zwaenepoel. PaRiS: Causally consistent transactions with non-blocking reads and partial replication. In *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*, pages 304–316, July 2019.

[SEL15] Nandakishore Santhi, Stephan Eidenbenz, and Jason Liu. The simian concept: Parallel discrete event simulation with interpreted languages and just-in-time compilation. In *2015 Winter Simulation Conference (WSC)*, pages 3013–3024, 2015.

[Sim22] Team SimPy. Simpy documentation, 2022.

[Spa23] 2023.

[SPAL11] Yair Sovran, Russell Power, Marcos K Aguilera, and Jinyang Li. Transactional storage for geo-replicated systems. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 385–400, New York, NY, USA, October 2011. Association for Computing Machinery.

[SPMO01] Antonio Sousa, Fernando Pedone, Francisco Moura, and Rui Oliveira. Partial replication in the database state machine. In *n Proc. of the IEEE International Symposium on Network Computing and Applications (NCA 2001)*, pages 298–309. IEEE CS, October 2001.

[TPC10] February 2010.

[TSM+20] Rebecca Taft, Irfan Sharif, Andrei Matei, Nathan VanBenschoten, Jordan Lewis, Tobias Grieger, Kai Niemi, Andy Woods, Anne Birzin, Raphael Poss, Paul Bardea, Amruta Ranade, Ben Darnell, Bram Gruneir, Justin Jaffray, Lucy Zhang, and Peter Mattis. CockroachDB: The resilient Geo-Distributed SQL database. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, SIGMOD '20, pages 1493–1509, New York, NY, USA, May 2020. Association for Computing Machinery.

[TZB+17] Gonçalo Tomás, Peter Zeller, Valter Balegas, Deepthi Akkoorath, Annette Bieniusa, João Leitão, and Nuno Preguiça. Fmke: A real-world benchmark for key-value data stores. In *Proceedings of the 3rd International Workshop on Principles and Practice of Consistency for Distributed Data*, PaPoC '17, New York, NY, USA, 2017. Association for Computing Machinery.

[Ulu94] Özgür Ulusoy. Processing real-time transactions in a replicated database system. *Distributed and Parallel Databases*, 2(4):405–436, 1994.

[WPS+00] M. Wiesmann, F. Pedone, A. Schiper, B. Kemme, and G. Alonso. Understanding replication in databases and distributed systems. In *Proceedings 20th IEEE International Conference on Distributed Computing Systems*, pages 464–474, 2000.

[YBP+14] X. Yu, G. Bezerra, A. Pavlo, S. Devadas, and M. Stonebraker. Staring into the abyss: An evaluation of concurrency control with one thousand cores. *Proc. VLDB Endowment*, 8(3):209–220, nov 2014.

[Zha20] Danny Zhang. System design topics: CAP theorem, 2020.

[ZXS+21] Jingyu Zhou, Meng Xu, Alexander Shraer, Bala Namasivayam, Alex Miller, Evan Tschannen, Steve Atherton, Andrew J. Beamon, Rusty Sears, John Leach, Dave Rosenthal, Xin Dong, Will Wilson, Ben Collins, David Scherer, Alec Grieser, Young Liu, Alvin Moore, Bhaskar Muppana, Xiaoge Su, and Vishesh Yadav. Foundationdb: A distributed unbundled transactional key value store. In *Proceedings of the 2021 International Conference on Management of Data*, SIGMOD '21, page 2653–2666, New York, NY, USA, 2021. Association for Computing Machinery.

[ÖV96] M Tamer Özsu and Patrick Valduriez. Distributed and parallel database systems. *ACM Computing Surveys*, 28, 1996.

[ÖV11] M Tamer Özsu and Patrick Valduriez. *Principles of Distributed Database Systems, Third Edition*. Pearson Education, Inc, 2011.