Universidade Federal Tecnológica do Paraná – UTFPR
Programa de Pós-Graduação em Computação Aplicada - PPGCA

Wagner Rezende Muniz Barretto

Cache Transparente: Uma Solução para Armazenamento de Dados Distribuídos voltada para Cargas de Trabalho Intensivas em Leituras

Curitiba

2018

Wagner Rezende Muniz Barretto

Cache Transparente: Uma Solução para Armazenamento de Dados Distribuídos voltada para Cargas de Trabalho Intensivas em Leituras

Dissertação de Mestrado apresentada ao Programa de Pós-graduação em Computação Aplicada da Universidade Tecnológica Federal do Paraná como requisito parcial para obtenção do grau de "Mestre em Computação Aplicada". Área de concentração: Redes e Sistemas Distribuídos.

Universidade Federal Tecnológica do Paraná – UTFPR

Programa de Pós-Graduação em Computação Aplicada - PPGCA

Orientadora: Prof.ª Dr.ª Ana Cristina B. Kochem Vendramin

Coorientador: Prof. Dr. Mauro Sérigo Pereira Fonseca

Curitiba

2018

Dados Internacionais de Catalogação na Publicação

Barretto, Wagner Rezende Muniz
B274c Cache transparente : uma solução para armazenamento
2018 de dados distribuídos voltada para cargas de trabalho intensivas em leituras / Wagner Rezende Muniz Barretto.—2018.
82 p. : il. ; 30 cm

Disponível também via World Wide Web
Texto em português com resumo em inglês
Dissertação (Mestrado) - Universidade Tecnológica Federal do Paraná. Programa de Pós-graduação em Computação Aplicada, Curitiba, 2018
Bibliografia: p. 79-82

1. Memória cache. 2. Sistemas de recuperação da informação. 3. Armazenamento de dados. 4. Banco de dados - Gerência. 5. Mineração de dados (Computação). 6. Computação - Dissertações. I. Vendramin, Ana Cristina Barreiras Kochem. II. Fonseca, Mauro Sergio Pereira. III. Universidade Tecnológica Federal do Paraná. Programa de Pós-graduação em Computação Aplicada. IV. Título.

CDD: Ed. 23 – 621.39

Biblioteca Central da UTFPR, Câmpus Curitiba
Bibliotecário: Adriano Lopes CRB9/1429

Ministério da Educação
Universidade Tecnológica Federal do Paraná
Diretoria de Pesquisa e Pós-Graduação

ATA DA DEFESA DE DISSERTAÇÃO DE MESTRADO Nº 62

DISSERTAÇÃO PARA OBTENÇÃO DO TÍTULO DE MESTRE EM COMPUTAÇÃO APLICADA
PROGRAMA DE PÓS-GRADUAÇÃO EM: COMPUTAÇÃO APLICADA
ÁREA DE CONCENTRAÇÃO: ENGENHARIA DE SISTEMAS COMPUTACIONAIS
LINHA DE PESQUISA: REDES E SISTEMAS DISTRIBUÍDOS

No dia 28 de junho de 2018 às 10h reuniu-se na Sala B204 da Sede Centro a banca examinadora composta pelos pesquisadores indicados a seguir, para examinar a dissertação de mestrado do(a) candidato(a) Wagner Rezende Muniz Barretto**,** intitulada: Cache Transparente: Uma Solução Para Armazenamento de Dados Distribuídos Voltada para Cargas de Trabalho Intensivas.

Orientador(a): Prof(a). Dr(a). Ana Cristina B. Kochem Vendramin

Co-orientador(a): Prof(a). Dr. Mauro Sergio P. Fonseca

Após a apresentação, o(a) candidato(a) foi arguido(a) pelos examinadores que, em seguida à manifestação dos presentes, consideraram o trabalho de pesquisa: ( ) Aprovado. ( ) Aprovado com restrições. Revisor indicado para verificação: _____. ( ) Reprovado.

Observações:

_____
_____
_____

Nada mais havendo a tratar, a sessão foi encerrada às __h__, dela sendo lavrado a presente ata, que segue assinada pela Banca Examinadora e pelo Candidato.

O candidato está ciente que a concessão do referido título está condicionada à: (a) satisfação dos requisitos solicitados pela Banca Examinadora; (b) entrega da dissertação em conformidade com as normas exigidas pela UTFPR; (c) atendimento ao requisito de publicação estabelecido nas normas do Programa; e (d) entrega da documentação necessária para elaboração do Diploma. A Banca Examinadora determina um **prazo máximo de _____ dias**, considerando os prazos máximos definidos no Regulamento Geral do Programa, para o cumprimento dos requisitos (desconsiderar caso reprovado), sob pena de, não o fazendo, ser desvinculado do Programa sem o Título de Mestre.

Prof(a). Dr(a). Ana Cristina B. Kochem Vendramin - Presidente – UTFPR _____

Prof(a). Dr(a). Luiz Celso Gomes Junior – UTFPR _____

Prof(a). Dr(a). Luiz Carlos E. de Bona – UFPR _____

Assinatura do Candidato: _____

Reservado à Coordenação

DECLARAÇÃO PARA A OBTENÇÃO DO TÍTULO DE MESTRE

A Coordenação do Programa declara que foram cumpridos todos os requisitos exigidos pelo Programa de Pós-Graduação para a obtenção do título de Mestre.

Curitiba, _____de _____de 20____.

Carimbo e Assinatura do(a) Coordenador(a) do Programa

Resumo

Em aplicações modernas de larga escala, usuários consomem mais dados do que produzem. Este comportamento resulta em sistemas de armazenamento de dados com cargas de trabalho dominadas por leituras. Deparados com este cenário, projetistas tem adotado modelos de replicação com cópia primária e mecanismos de cache como soluções. Estas soluções empregam sistemas de armazenamento de dados que enfrentam grandes desafios para garantir a consistência dos dados. Atualmente, a literatura carece de uma análise formal da consistência provida por estas soluções e de quais problemas decorrem do uso das mesmas. Este trabalho apresenta uma análise formal que identifica os problemas de consistência exibidos pelas soluções em uso na atualidade e introduz o Cache Transparente, uma nova solução baseada em mecanismos de cache capaz de endereçar tais problemas. A análise das soluções demonstra que as mesmas permitem diversas anomalias de consistência. O projeto do Cache Transparente é apresentado em detalhes e sua consistência é formalmente demonstrada. A comparação entre as soluções mostra que o Cache Transparente é mais consistente e mais eficiente no uso de rede do que as soluções atuais. As contribuições feitas por esse trabalho possibilitam que projetistas identifiquem de forma mais precisa a consistência dos seus sistemas e os fornece uma nova solução a ser considerada.

Palavras-chave: Armazenamento de Dados Distribuídos, Consistência, Replicação de Dados

Abstract

In large-scale modern applications, users consume much more data than they create. This behavior results in data storage systems with workloads dominated by reads. Faced with this scenario, designers have adopted solutions such as primary copy replication and cache mechanisms. These solutions employ distributed data storage systems, which face major challenges to ensure data consistency. Currently, the literature lacks a formal and in-depth analysis of the consistency provided by these solutions and which problems arise from their use. This work presents a formal analysis that identifies the consistency problems presented by current solutions and introduces Cache-Through, a new solution capable of addressing such problems. The analysis of current solutions demonstrates that various consistency anomalies can happen. The design of Cache-Through is presented in detail and its consistency is formally demonstrated. The comparison between the solutions shows that Cache-Through is more consistent and more efficient in network usage than current solutions. The contributions made by this work allow designers to identify more precisely the consistency of their systems and provide them with a new solution to consider.

Keywords: Distributed Data Storage Systems, Consistency, Data Replication

Lista de ilustrações

Lista de tabelas

Lista de Siglas

2PC	Two Phase Commit
CDN	Content Delivery Network
FIFO	First In First Out
HTTP	Hyper Text Transfer Protocol
JSON	Javascript Object Notation
RDBMS	Relational Database Management System
RPC	Remote Procedure Call
WAN	Wide Area Network

Lista de símbolos

$\mathbb{I}$	Conjunto de instâncias de processos *Cache*
$\mathbb{N}$	Conjunto dos números naturais

Sumário

1 Introdução

Com a disseminação e popularização do acesso à internet e a dispositivos móveis conectados, uma demanda por aplicações baseadas em compartilhamento de informações como redes sociais, lojas virtuais, aplicativos baseados em economia de compartilhamento e *marketplaces* nasceu. Usuários destes serviços consomem muito mais dados do que produzem (NISHTALA et al., 2013). Por exemplo, em um comércio eletrônico mais produtos são visualizados do que comprados, já em redes sociais mais comentários são lidos do que publicados.

Outro fator a ser observado é uma tendência de alta popularidade para um conjunto muito pequeno dos dados. Pesquisas realizadas sobre padrões de acesso ao Twitter mostram que o 99.9° percentil dos usuários possuem aproximadamente 24 mil seguidores, já o 99° percentil possui aproximadamente 2 mil seguidores, enquanto que a mediana é de apenas 61 seguidores (BRUNER, 2013). Já pesquisas sobre padrões de acesso a servidores web (BRESLAU et al., 1999) mostram que o acesso a páginas segue uma distribuição semelhante a lei de Zipf (ZIPF, 1949) [1]. Os dados demonstram que existe uma tendência para cargas de trabalho intensivas em leituras e estas leituras são inclinadas para um pequeno subconjunto dos dados totais.

As características de uso previamente mencionadas resultam em sistemas de armazenamento de dados submetidos a cargas de trabalho dominadas por buscas e leituras. Deparados com esta situação, projetistas de aplicações tem adotado soluções de armazenamento de dados distribuídos desenvolvidas para priorizar a leitura dos dados. Tais soluções sofrem de problemas relacionados à consistência dos dados e eficiência na utilização de recursos de rede. Este trabalho descreve tais soluções e demonstra formalmente os problemas causados pelas mesmas. Adicionalmente, este trabalho propõe o projeto de uma nova solução que resolve grande parte dos problemas identificados e ao mesmo tempo atende aos requisitos de cargas de trabalho intensivas em leituras.

1.1 Sistemas de Armazenamento de Dados

Sistemas de armazenamento de dados são sistemas que gerenciam um conjunto de objetos potencialmente mutáveis. Como exemplos de tais sistemas podem-se citar sistemas de arquivos e bases de dados. Estes sistemas oferecem funcionalidades de recuperação e

[1] A Lei de Zipf diz que a frequência de um elemento em um determinado conjunto é inversamente proporcional a sua posição no rank de frequências. Por exemplo, a palavra mais frequente em um determinado livro ocorre duas vezes mais do que a segunda palavra mais frequente, três vezes mais do que a terceira mais frequente e assim por diante.

armazenamento de objetos ao seus clientes. A interação dos clientes com o sistema é feita através de operações de leitura ou escrita expostas pelo sistema. Quanto à arquitetura, os sistemas podem operar com um único processo ou vários processos distribuídos que se comunicam através de uma rede de computadores.

Sistemas de armazenamento de dados distribuídos tem como objetivo principal serem versões mais escaláveis e tolerante a falhas se comparado com sistemas de armazenamento centralizados (VIOTTI; VUKOLIć, 2016). Em arquiteturas distribuídas, vários processos do sistema de armazenamento de dados são distribuídos entre vários nós de uma rede. Neste trabalho, considera-se que cada nó da rede hospeda apenas um processo do sistema proposto.

1.2 Soluções para Cargas de Trabalho Intensivas em Leituras em Uso na Atualidade

Quando deparados com cargas de trabalho intensivas em leituras, projetistas tem adotado diferentes soluções para sistemas de armazenamento de dados. As soluções normalmente empregam sistemas distribuídos para dividir a carga de trabalho entre vários processos. Duas soluções em particular são amplamente adotadas na indústria, a *Replicação Assíncrona com Cópia Primária* e o *Cache-aside*.

A replicação assíncrona com cópia primária é uma técnica de replicação de dados explorada na literatura de bancos de dados (GRAY et al., 1996; PACITTI; MINET; SIMON, 1999; CECCHET; CANDEA; AILAMAKI, 2008) que permanece em uso nos dias de hoje. Na técnica, um processo (réplica primária) é responsável por processar operações de escrita e leitura enquanto um conjunto separado de processos (réplicas secundárias) processam apenas operações de leitura. A réplica primária se encarrega de enviar as modificações causadas pelas operações de escrita para cada uma das réplicas secundárias. Este processo de replicação ocorre após as operações de escrita terem sido confirmadas aos clientes.

O *Cache-aside* é uma técnica muito utilizada na indústria, particularmente em aplicações web (ATIKOGLU et al., 2012). Na técnica, um sistema de armazenamento de dados otimizado para leituras (*cache*) é utilizado "ao lado" de um sistema de armazenamento de dados principal. Uma aplicação cliente faz uso dos dois sistemas para realizar as operações de leituras e escritas.

Ambas as soluções enfrentam desafios relacionados ao fato de trabalharem com uma arquitetura distribuída. Um sistema de armazenamento de dados distribuído, como as duas soluções previamente mencionadas, mantém cópias dos dados em diferentes processos. Manter o estado destas cópias consistentes entre todos os processos do sistema não é uma

tarefa trivial.

1.3 Consistência em Sistemas Distribuídos

A noção de consistência em sistemas distribuídos teve diferentes significados nos últimos anos (VIOTTI; VUKOLIć, 2016). Nos anos 80 consistência era normalmente atribuída ao que hoje chamamos de "consistência forte", conceito este que posteriormente passou a ser conhecido como *Linearização* (HERLIHY; WING, 1990). A linearização estipula que um sistema deve expor seu estado como se houvesse uma cópia única de tal estado, mesmo que este seja internamente replicado entre diferentes processos. Nos dias de hoje, consistência é discutida sob a forma de *modelos de consistência*. A linearização é um destes modelos, porém muitos outros surgiram oferecendo garantias menos restritas que a linearização.

A principal motivação da adoção de modelos de consistência que ofereçam garantias menos restritas se comparados à linearização é explicada pelo *Teorema CAP* (BREWER, 2000; GILBERT; LYNCH, 2002). O conceito principal lançado pelo teorema é que um sistema distribuído não é capaz de prover linearização e alta disponibilidade[2] sob presença de particionamentos de rede. Esta impossibilidade associada à necessidade de minimizar indisponibilidades levou projetistas a considerarem alternativas à linearização (VOGELS, 2008; BAILIS; GHODSI, 2013).

Os primeiros trabalhos a popularizarem modelos menos restritos de consistência apresentavam sistemas de armazenamento distribuídos que operavam com um modelo de consistência chamado pelos autores de "consistência eventual" (DECANDIA et al., 2007; COOPER et al., 2008; LAKSHMAN; MALIK, 2010). Trabalhos posteriores passaram a explorar um modelo mais restrito do que o eventual chamado de "consistência causal" (LLOYD et al., 2011; LLOYD et al., 2013; ALMEIDA; LEITÃO; RODRIGUES, 2013). As definições de consistência apresentadas por estes trabalhos não seguem um padrão que permita uma comparação objetiva entre os modelos, nem usam métodos formais para especificar consistência. Motivado por estes fatos um trabalho recente (BURCKHARDT, 2014) apresentou um *framework* matemático capaz de modelar e comparar não apenas um, mas todo o espectro de possíveis modelos de consistência.

1.4 Objetivos do Trabalho

Este trabalho aborda soluções de sistemas de armazenamento de dados distribuídos projetados para trabalhar com cargas de trabalho intensivas em leituras. O objetivo geral

[2] Alta disponibilidade significa que todas requisições enviadas para um processo não falho do sistema devem resultar em uma resposta

deste trabalho é propor uma nova solução para estes sistemas que ofereça vantagens se comparado as soluções atuais.

Adicionalmente, o trabalho visa os seguintes objetivos específicos:

- Descrever o funcionamento, prós e contras das soluções atuais para sistemas de armazenamento de dados distribuídos projetados para cargas de trabalho intensivas em leitura.
- Analisar formalmente os modelos de consistência providos por tais soluções.
- Propor uma nova solução capaz de atender aos requisitos providos pelas soluções atuais ao mesmo tempo que provê um modelo de consistência mais forte.

1.5 Estrutura do Trabalho

Os primeiros capítulos deste trabalho introduzem os principais temas abordados e resumem alguns dos conceitos importantes para o entendimento do trabalho. O presente capítulo introduz o problema de cargas de trabalho intensivas em leituras, suas soluções e o tópico de consistência em sistemas distribuídos. O capítulo ainda apresenta os objetivos do trabalho. O Capítulo 2 descreve os conceitos básicos sobre replicação de dados. O Capítulo 3 descreve o estado da arte em consistência de sistemas distribuídos. Os conceitos apresentados nestes capítulos servem como base para as discussões apresentadas no decorrer do trabalho.

Os próximos capítulos apresentam as principais contribuições deste trabalho. O Capítulo 4 aborda as soluções em uso na atualidade para sistemas com carga de trabalho intensivas em leituras. As principais soluções adotadas pela indústria são descritas e analisadas com foco na consistência provida. O Capítulo 5 apresenta uma nova solução que mantém os benefícios das soluções existentes e garante um modelo de consistência mais forte. O funcionamento da solução é apresentado sob a forma de pseudocódigos e seu modelo de consistência é demonstrado formalmente. Por fim, o Capítulo 6 apresenta observações finais sobre o trabalho, bem como trabalhos relacionados e futuros.

2 Replicação de Dados

Este capítulo apresenta os conceitos básicos sobre o tema replicação de dados. Estes conceitos são fundamentais para as discussões apresentadas no decorrer deste trabalho. As principais estratégias de replicação são apresentadas e descritas juntamente com suas aplicabilidades.

Replicação de dados é o processo pelo qual um sistema de armazenamento de dados distribuído copia dados entre seus participantes. A replicação objetiva manter os dados atualizados em todos os participantes do sistema. Existem dois motivos principais para se empregar replicação de dados: aumento de desempenho e aumento de disponibilidade (CECCHET; CANDEA; AILAMAKI, 2008).

Para efeitos das definições apresentadas neste capítulo, considera-se como um sistema de armazenamento de dados distribuído um conjunto de processos interconectados capazes de se comunicar através de troca de mensagens. O objetivo do sistema é gerenciar um conjunto de *objetos* que representa o *estado* do sistema. Cada processo mantém uma cópia do estado do sistema e por tal motivo são chamados de *réplicas*. Os *clientes* do sistema são processos que se comunicam individualmente com as réplicas. Os clientes submetem operações de leitura ou escrita sobre o estado do sistema mantido pelas réplicas.

2.1 Estratégias de Replicação

Gray et. al. (1996) categoriza replicação de dados utilizando-se de dois parâmetros: *Propagação* e *Posse* (do inglês: *ownership*). Propagação determina quando os dados são enviados da réplica que recebeu uma atualização para as demais réplicas. Posse determina quais réplicas podem realizar escritas nos objetos e quais podem realizar apenas leituras.

A propagação pode ser de dois tipos:

- *Eager*: as escritas são aplicadas em todas as réplicas antes da operação ser confirmada ao cliente. Este trabalho se refere a replicação *eager* como replicação *síncrona*.
- *Lazy*: as escritas são enviadas às demais réplicas após a operação ter sido confirmada ao cliente. Este trabalho se refere a replicação *lazy* como replicação *assíncrona*.

A posse também pode ser de dois tipos:

- *Cópia Primária*: cada objeto possui uma única réplica (*Primária*) que pode realizar escritas no mesmo. Todas as demais (*Secundárias*) podem apenas realizar leituras no objeto.

	Síncrona	**Assíncrona**
Cópia Primária	Síncrona com Cópia Primária	Assíncrona com Cópia Primária
Atualzação em todos os Lugares	Síncrona com Atualização em Todos os Lugares	Assíncrona com Atualização em Todos os Lugares

Tabela 1 – Possíveis estratégias de replicação de dados

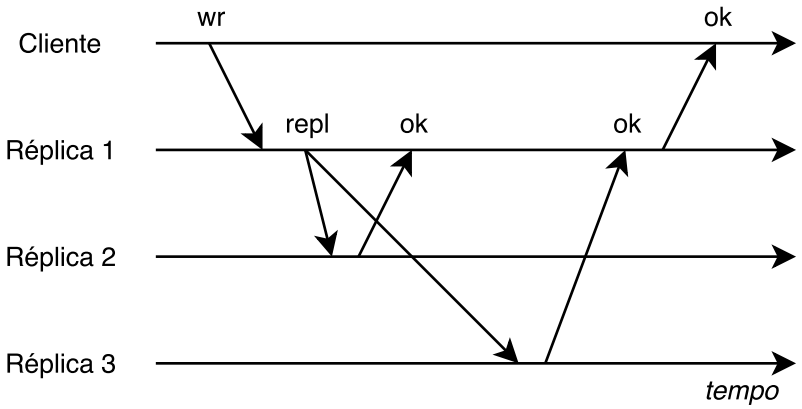


Figura 1 – Representação de uma operação de escrita (*wr*) em estratégias de replicação síncrona

- *Atualização em Todos os Lugares* (do inglês: *Update Anywhere*): todos os objetos podem ser escritos e lidos por todas as réplicas.

A combinação dos possíveis tipos para as propriedades propagação e posse gera quatro estratégias de replicação distintas descritas na Tabela 1.

2.1.1 Estratégias Síncronas

As estratégias síncronas executam a replicação dos objetos no escopo de cada operação. A Figura 1 ilustra um fluxo básico de uma operação de escrita nas estratégias síncronas. Isto faz com que todas as réplicas se mantenham atualizadas durante todos os momentos (GRAY et al., 1996). Em contrapartida, o tempo de resposta das operações é sobrecarregado por conta das rotinas de replicação. A propagação síncrona não é adequada para sistemas que podem sofrer particionamentos de rede. Um particionamento acarreta na impossibilidade de se contatar todas as réplicas, o que é uma necessidade para cada operação executada nestas estratégias.

Na estratégia *Replicação Síncrona com Cópia Primária* as escritas são sempre realizadas na réplica primária. A réplica primária é responsável por enviar uma atualização para as demais réplicas contendo a modificação feita pela escrita. A operação só é confirmada para o cliente após todas as demais réplicas confirmarem a atualização. Algoritmos como

Two Phase Commit (2PC) (GRAY, 1978) são utilizados para coordenar o processo de atualização das réplicas (WIESMANN et al., 2000).

A replicação síncrona com cópia primária se beneficia da ausência de conflitos de escritas entre réplicas. Isto se dá pelo fato de todas as escritas serem processadas por um único processo. Em contrapartida, os tempos de resposta observados pelos clientes nas operações de escrita são prejudicados. Cada escrita submetida deve levar em consideração o tempo de propagação entre todas as réplicas.

Na *Replicação Síncrona com Atualização em Todos os Lugares* qualquer réplica pode receber operações de escrita para todos os objetos. Isto torna a estratégia sujeita a conflitos causados por escritas concorrentes em um mesmo objeto submetidas a réplicas diferentes. Por tal motivo a estratégia requer um mecanismo de trava distribuída ao nível de objeto para controlar o acesso de escrita. A trava é utilizada com o propósito de impedir que mais de uma réplica inicie uma operação de escrita no mesmo objeto, evitando assim conflitos de escrita. Uma vez a trava estabelecida, o mesmo mecanismo de coordenação com 2PC utilizado na estratégia de cópia primária é empregado para atualizar o objeto em todas as réplicas.

Com o uso das travas, assim como a estratégia de cópia primária, a replicação síncrona com atualização em todos os lugares não sofre de problemas de conflitos de escritas. Em contrapartida, a possibilidade de *deadlocks* na estratégia cresce proporcionalmente a medida em que se adiciona réplicas no sistema (GRAY, 1978). Outro problema está nos tempos de resposta, o mecanismo de trava distribuída sobrecarrega ainda mais as operações de escrita.

2.1.2 Estratégias Assíncronas

Nas estratégias assíncronas as operações são confirmadas aos clientes logo que a réplica que recebe a operação termina de atualizar o seu estado local. Este fluxo de execução é ilustrado pela Figura 2. O tempo de resposta observado pelos clientes do sistema é baixo, pois o único processamento necessário é o da réplica que recebeu a requisição. A replicação assíncrona é ideal para cenários onde a rede que conecta as réplicas possa sofrer particionamentos ou sistemas que permitem que réplicas possam trabalhar de forma desconectada. A principal desvantagem destas estratégias está nos problemas de consistência gerados pelo tipo de propagação.

A estratégia *Replicação Assíncrona com Cópia Primária* é a estratégia com implementação mais simples dentre as quatro apresentadas na Tabela 1. Escritas são aplicadas na réplica primária e imediatamente confirmadas aos clientes. Somente após o envio da confirmação ao cliente as demais réplicas são atualizadas. A atualização ocorre através do envio de mensagens para todas as réplicas via *unicast* ou *broadcast*, ambos ordenados.

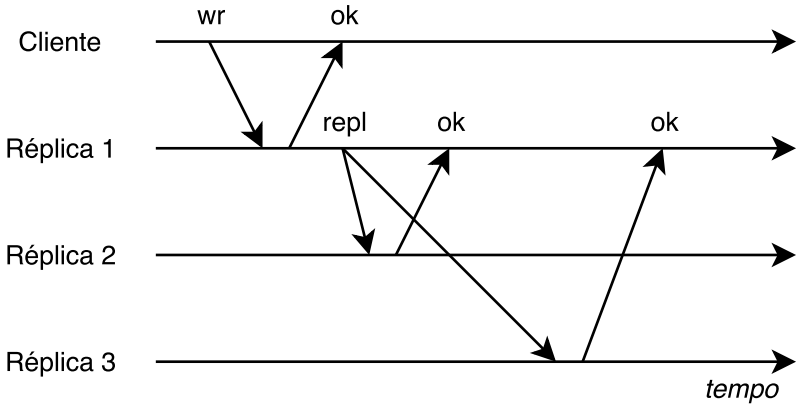


Figura 2 – Representação de uma operação de escrita (*wr*) em estratégias de replicação assíncrona

O uso da réplica primária que age como centralizadora das escritas elimina os conflitos de escrita. A única condição obrigatória para que a replicação ocorra corretamente é que as réplicas secundárias recebam as atualizações na ordem em que as mesmas foram executadas pela réplica principal. Esta estratégia é abordada com detalhes no Capítulo 4, onde uma análise formal do modelo de consistência provido pela estratégia é apresentado bem como as diversas anomalias de consistência geradas pela estratégia.

Na estratégia *Replicação Assíncrona com Atualização em Todos os Lugares* os clientes podem submeter operações de escrita para todas as réplicas. As operações são retornadas assim que a réplica em questão atualiza o seu estado local. Esta estratégia acarreta em grandes desafios relacionados à sincronização do estado das diferentes réplicas. Se particionamentos de rede forem suportados a consistência forte é impossível de ser atingida conforme demonstrado pelo teorema CAP.

As abordagens de projeto para replicação assíncrona com atualização em todos os lugares são várias. Uma breve revisão dos principais trabalhos na área disponíveis na literatura é apresentada no Capítulo 6. O principal foco dos trabalhos realizados até então está em projetar sistemas que oferecem modelos de consistência relaxados e que suportam particionamentos de rede.

2.2 Desempenho e Disponibilidade

Estabelecidos conceitos sobre estratégias de replicação, pode-se explorar como os mesmos são aplicados para solucionar problemas de desempenho e disponibilidade. Primeiramente consideremos um sistema de instância única não replicado. Em uma eventual falha de *hardware*, o sistema se torna indisponível. No caso de danos irreversíveis em componentes de armazenamento como discos, os dados do sistema podem não ser recuperáveis. Este é um exemplo de problema de disponibilidade. No caso de desempenho,

a instância única pode não ser suficiente para processar determinadas cargas de trabalho. O que torna necessário a distribuição da carga entre várias instâncias do sistema, o que implica em replicação.

Um das soluções mais comuns para problemas de disponibilidade é através de mecanismos de *Failover*, isto é, a capacidade de se direcionar clientes de uma réplica falha para uma não falha (CECCHET; CANDEA; AILAMAKI, 2008). O processo inverso, onde os clientes retornam para uma réplica que se recuperou de uma falha é chamado de *Failback*. Esta capacidade pode ser atendida com a estratégia de replicação síncrona com cópia primária. Neste cenário as réplicas secundárias trabalham num modo chamado de *Hot standby*. Neste modo as réplicas não recebem invocações de operações pelos clientes até que uma falha seja detectada na réplica primária, a partir daí uma das secundárias assume o papel de primária.

Para tolerar falhas ao nível de centro de dados como catástrofes naturais e ainda assim permanecer disponível, a replicação deve compreender réplicas conectadas por redes de longa distância (WANs). Estratégias de replicação síncrona não são adequadas para WANs devido a alta chance de atrasos e particionamentos. Logo, somente as estratégias assíncronas são disponíveis como opção para estes cenários.

Do ponto de vista dos clientes de um sistema de armazenamento de dados, desempenho significa tempo de resposta das operações submetidas. Portanto, estratégias assíncronas são adequadas para solucionar problemas de desempenho. Já o tipo de posse a ser utilizado é dependente da carga de trabalho do sistema. Em cargas de trabalho intensivas em leituras a replicação assíncrona com cópia primária é mais adequada. Já em cargas de trabalho balanceadas ou intensivas em escrita, a replicação assíncrona com atualização em todos os lugares é mais adequada.

O uso de replicação assíncrona, apesar de prover baixos tempos de resposta possui a desvantagem de não possibilitar a consistência forte. Por tal motivo, compreender os conceitos de consistência em sistemas distribuídos é fundamental. O próximo capítulo aborda este tema em detalhes. Os conceitos a serem apresentados possibilitam analisar a exatidão de projetos sistemas de armazenamento de dados distribuídos e suas implicações nas diversas aplicações que o utilizam.

3 Consistência em Sistemas Distribuídos

Motivados pelas dificuldades impostas pelo teorema CAP e pela necessidade de prover aplicações em larga escala, membros da indústria passaram a adotar soluções de armazenamento de dados com garantias de consistência menos restritas. Tais garantias menos restritas de consistência passaram a ser conhecidas como *Consistência Eventual* (VOGELS, 2008). A noção de consistência eventual não é nova, a mesma foi introduzida na década de 90 (TERRY et al., 1994), porém ganhou popularidade com trabalhos como PNUTS do Yahoo! (COOPER et al., 2008), o Dynamo da Amazon (DECANDIA et al., 2007) e o Cassandra do Facebook (LAKSHMAN; MALIK, 2010). Esses trabalhos e outros que vieram a surgir posteriormente apresentam projetos de sistemas de banco de dados distribuídos que sacrificam consistência em prol de alta disponibilidade ao mesmo tempo em que abandonam o modelo de dados relacional (por tal motivo passaram a ser chamados de *NoSQL*).

O conceito de consistência eventual não era precisamente definido por tais trabalhos, o que motivou alguns trabalhos a tentar prover descrições mais formais do conceito (SHAPIRO et al., 2011b). Ao mesmo tempo, trabalhos como COPS (LLOYD et al., 2011) e Eiger (LLOYD et al., 2013) passaram a explorar um modelo mais restrito de consistência que ainda é capaz de manter alta disponibilidade. Este novo modelo, baseado nas noções de causa em sistemas distribuídos (LAMPORT, 1978), passou a ser chamado de *Consistência Causal*.

Cada trabalho que explorava um novo modelo de consistência o especificava de forma diferente. Isto acabou por gerar muitas noções de modelos de consistência que não eram padronizadas ou diretamente comparáveis. Esta falta de padronização motivou a criação de métodos formais para descrever e verificar consistência em sistemas distribuídos (BURCKHARDT, 2014; VIOTTI; VUKOLIć, 2016). Estes métodos são capazes de expressar formalmente o espectro total de consistência em sistemas distribuídos e não apenas um modelo de forma isolada. Este Capítulo revisa o estado da arte sobre modelos de consistência com base em tais métodos. Inicialmente, os conceitos que servem como base para o entendimento do tópico são apresentados. Após, os modelos de consistência são gradualmente apresentados, partindo do menos para o mais restrito. Por fim, uma comparação entre os modelos é apresentada. Salvo quando explicitamente mencionado, todas as definições apresentadas neste capítulo são baseadas nos trabalhos de Burkhardt (2014) e Viotti (2016).

3.1 Notações e Nomenclaturas Utilizadas no Trabalho

As especificações e verificações de modelos de consistência fazem uso extensivo de teoria de conjuntos e suas notações. Um breve resumo das notações menos elementares utilizadas neste capítulo e ao decorrer do trabalho é apresentado a seguir.

Conjuntos definidos por um predicado (*Pred*) são representados utilizando a notação de construção de conjuntos $\{x \mid Pred(x)\}$, ou $\{x \in \mathbf{dom} \mid Pred(x)\}$ quando existe um domínio (**dom**) predefinido. Por exemplo, o conjunto dos números naturais positivos é representado por $\{x \in \mathbb{N} \mid x > 0\}$. A cardinalidade ou tamanho de um conjunto C é representada por $|C|$. O complemento relativo entre dois conjuntos A e B, ou todos os elementos de B não presentes em A é representado por $B \setminus A$. O produto cartesiano de dois conjuntos A e B é representado por $A \times B$ cujo resultado é um conjunto de pares $\{(a, b) \mid a \in A, b \in B\}$.

3.2 Operações, Históricos e Relações

Para efeitos deste trabalho, considera-se um *Sistema de Armazenamento de Dados Distribuído* como um conjunto finito de processos interagindo uns com os outros sobre uma rede assíncrona. A interação clientes do sistema com os processos que o compõe é feita através de *Operações*. Operações podem ser definidas pela tupla $\langle proc, type, ival, oval, stime, rtime \rangle$, onde:

- *proc* é o identificador do processo invocando a operação.
- *type* é o tipo da operação.
- *ival* é o valor de entrada da operação.
- *oval* é o valor de saída da operação.
- *stime* é o tempo inicial da invocação da operação.
- *rtime* é o tempo de retorno da operação.

O conjunto de invocações de operações em uma execução de um dado sistema é representada por um *Histórico*. Históricos demonstram o comportamento observável de um sistema através da interação entre clientes e sistema (BURCKHARDT, 2014). Históricos são ilustrados através de diagramas de linha do tempo como o ilustrado na Figura 3. Na Figura, dois processos (A e B) interagem com um registrador através de operações de escrita (wr) e leitura (rd).

Uma *Relação* entre as operações de um histórico é um subconjunto do produto cartesiano de um histórico (BURCKHARDT, 2014). Por exemplo, um histórico com duas

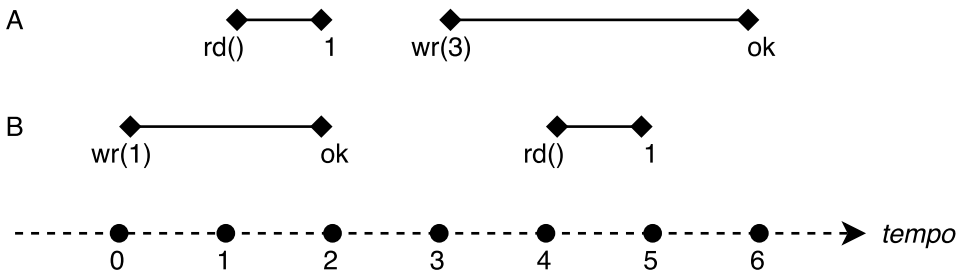


Figura 3 – Histórico de um registrador

operações $H = \{a, b\}$ tem seu produto cartesiano definido como:

$$H \times H = \{(a, a), (a, b), (b, b), (b, a)\} \tag{3.1}$$

Uma relação arbitrária $\mathsf{rel} \subseteq H \times H$ pode ter a seguinte forma:

$$\mathsf{rel} = \{(a, a), (b, a)\} \tag{3.2}$$

No decorrer do trabalho a notação $x \xrightarrow{\mathsf{rel}} y$ é usada para indicar que $(x, y) \in \mathsf{rel}$.

Para efeitos de demonstrações de modelos de consistência as seguintes relações entre operações de um histórico são utilizadas (BURCKHARDT, 2014; VIOTTI; VUKOLIć, 2016):

- rb (retorna-antes, do inglês *returns-before*) é uma relação de ordem baseada em precedência, indica que uma operação retornou antes que outra. Formalmente:

$$\mathsf{rb} \stackrel{\text{def}}{=} \{(a, b) \mid a, b \in H \wedge a.rtime < b.stime\} \tag{3.3}$$

- ss (mesma-sessão, do inglês *same-session*) é uma relação de equivalência entre duas operações invocadas pelo mesmo processo ou sessão. Formalmente:

$$\mathsf{ss} \stackrel{\text{def}}{=} \{(a, b) \mid a, b \in H \wedge a.proc = b.proc\} \tag{3.4}$$

- so (ordem-de-sessão, do inglês *session-order*) é uma relação de ordem das operações de uma mesma sessão. Formalmente:

$$\mathsf{so} \stackrel{\text{def}}{=} \mathsf{rb} \cap \mathsf{ss} \tag{3.5}$$

A Tabela 2 descreve as operações do histórico ilustrado pela Figura 3 e as relações rb, ss e so.

Além das três relações previamente descritas, outras duas relações são utilizadas para definições de modelos de consistência. Estas relações modelam comportamentos não determinísticos de comunicações assíncronas e, especialmente, sistemas que não possuem uma noção global de tempo. São estas relações que permitem a definição de modelos de consistência relaxados. As relações são (BURCKHARDT, 2014; VIOTTI; VUKOLIć, 2016):

Operações $\langle proc, type, ival, oval, stime, rtime \rangle$	Relações
$a = \langle A, rd, \varnothing, 1, 1, 2 \rangle$ $b = \langle A, wr, 3, ok, 3, 6 \rangle$ $c = \langle B, wr, 1, ok, 0, 2 \rangle$ $d = \langle B, rd, \varnothing, 1, 4, 5 \rangle$	$\mathsf{rb} = \{(a,b),(a,d),(c,b),(c,d)\}$ $\mathsf{ss} = \{(a,a),(a,b),(b,a),(b,b),(c,c),$ $(c,d),(d,c),(d,d)\}$ $\mathsf{so} = \{(a,b),(c,d)\}$

Tabela 2 – Operações e relações do histórico representado pela Figura 3

- vis (visibilidade): uma operação a é visível à operação b ($a \xrightarrow{\mathsf{vis}} b$) se os efeitos de a são visíveis ao processo que invoca b. Por exemplo, b lê um valor escrito por a.
- ar (arbitragem): é utilizado para indicar como um sistema resolve conflitos. Dada duas operações não relacionadas a e b, $a \xrightarrow{\mathsf{ar}} b$ significa que o sistema considera que a ocorreu antes que b. Em implementações de sistemas distribuídos assíncronos essa relação normalmente é resolvida por tempos lógicos, processos centralizadores ou protocolos de consenso.

Ao contrário de rb, ss e so, visibilidade e arbitragem não são imediatamente inferidas com base em um histórico. Estas relações são estabelecidas com base no algoritmo que um determinado sistema implementa.

3.3 Modelos de Consistência

O comportamento de um sistema de armazenamento de dados distribuído é determinado aplicando-se *Garantias de Ordenação* sobre as execuções do sistema. Uma garantia de ordenação é um predicado lógico que deve ser válido para todos os possíveis históricos de um sistema. Formalmente, as garantias de ordenação são expressas sobre a forma de expressões algébricas utilizando-se das relações entre operações.

Um determinado sistema pode atender múltiplas garantias de ordenação. Determinados conjuntos de garantias de ordenação formam *Modelos de Consistência*. Um sistema que atende todas as garantias de ordenação de um modelo de consistência X é por vezes chamado de "X-consistente". O conjunto de garantias de ordenação atendidas por um sistema não são restritas aos conjuntos dos modelos de consistência. Por exemplo, se o modelo de consistência X requer as garantias de ordenação (a, b), e o modelo de consistência Y requer as garantias (a, b, c, d), um sistema pode atender as garantias (a, b, c), porém ainda será um sistema X-consistente.

A seguir os modelos de consistência e as garantias de ordenação que os compõem são descritos. Todas as garantias de ordenação são apresentadas juntamente com suas

respectivas definições formais. Por fim, um método de comparação e hierarquia entre os modelos é descrita.

3.3.1 Consistência Eventual

Informalmente, consistência eventual pode ser definida como garantia de que se operações adicionais pararem de ser invocadas em um determinado objeto, o estado do objeto em todas as réplicas eventualmente será o mesmo (SHAPIRO et al., 2011b; BAILIS; GHODSI, 2013). Esta definição descreve a propriedade de "convergência" de um sistema eventualmente consistente.

Formalmente, o modelo de consistência eventual é expresso pela junção de duas garantias de ordenação: VisibilidadeEventual e AusênciaDeCausalidadeCircular. A garantia VisibilidadeEventual determina que uma operação a que tenha completado deve eventualmente se tornar visível para todas as sessões. Esta garantia formaliza a propriedade de convergência de sistemas eventualmente consistentes. A garantia é expressa requerendo que em cada sessão, quase todas as operações que iniciaram após a ter retornado vejam a. A definição formal da garantia não é trivial, para tal define-se "cada sessão" do histórico como $[f] \in H$, e o conjunto de "quase" todas as operações de uma sessão como $H/\approx_{ss}$. Assim, a definição formal da garantia é expressa como:

$$\text{VisibilidadeEventual} \stackrel{\text{def}}{=} \forall a \in H, \forall [f] \in H/\approx_{ss}: \\ |\{b' \in [f] | (a \xrightarrow{\mathsf{rb}} b) \wedge (a \not\xrightarrow{\mathsf{vis}} b)\}| < \infty \quad (3.6)$$

Ausência de causalidade circular é uma propriedade que afirma que um grafo formado pelas dependências das operações realizadas em um mesmo objeto compartilhado é acíclico. Esta garantia não se aplica a sistemas de armazenamentos de dados e por isso não será abordada em detalhes. A garantia existe para tratar situações existentes em sistemas de memória compartilhada (BURCKHARDT, 2014).

Com as duas garantias, pode-se definir o modelo de consistência eventual como:

$$\text{ConsistênciaEventualBásica} \stackrel{\text{def}}{=} \\ \text{VisibilidadeEventual} \wedge \text{AusênciaDeCausalidadeCircular} \quad (3.7)$$

3.3.2 Garantias de Sessão

O modelo de consistência ConsistênciaEventualBásica oferece poucas restrições de ordenação, e por tal motivo permite que uma série de anomalias sejam observadas. As anomalias que geram impactos mais facilmente observados por usuários são as que afetam as sessões. Se usuários submetem uma sequência de operações como parte de uma

sessão, os mesmos esperam que o sistema preserve a ordem em que as operações foram submetidas.

Para especificar os comportamentos relacionados a sessões, duas garantias de ordenação são definidas. Estas garantias não são necessárias para nenhum modelo de consistência, porém podem estar presentes em sistemas que ofereçam ConsistênciaEventualBásica. As duas garantias são descritas a seguir.

Um usuário que insere qualquer tipo de informação em uma aplicação espera conseguir ler ou visualizar a informação que acabou de inserir. Consistência eventual sozinha não garante essa condição, a informação pode se tornar disponível para leitura apenas num momento futuro. Para garantir a condição, um sistema deve ser capaz de prover a garantia LerMinhasEscritas. A formalização da garantia diz que a ordem de sessão deve ser um sub-conjunto da visibilidade, formalmente:

$$\text{LerMinhasEscritas} \stackrel{\text{def}}{=} \mathsf{so} \subseteq \mathsf{vis} \tag{3.8}$$

Outra anomalia possível sob consistência eventual são leituras não incrementais ou monotônicas. A anomalia ocorre quando uma sessão visualiza uma operação de escrita e em um momento futuro da ordem de sessão deixa de visualizar a mesma operação. Novamente, consistência eventual sozinha não garante que esta condição não ocorra, a convergência sempre pode ocorrer num momento futuro. A garantia LeiturasMonotônicas pode ser expressa formalmente da seguinte forma:

$$\text{LeiturasMonotônicas} \stackrel{\text{def}}{=} \forall a \in H, \forall b, c \in H|_{rd} : a \xrightarrow{\mathsf{vis}} b \wedge b \xrightarrow{\mathsf{so}} c \Rightarrow a \xrightarrow{\mathsf{vis}} c \tag{3.9}$$

3.3.3 Consistência Causal

O modelo de consistência causal tem como base a noção de *acontece-antes* (do inglês, *happened before*) (LAMPORT, 1978). Tal noção foi originalmente definida no contexto de sistemas de troca de mensagens e posteriormente traduzida para um modelo de consistência (AHAMAD et al., 1995). De acordo com essa noção, uma operação a acontece antes de b se ao menos uma das três seguintes condições for satisfeita:

1. As duas operações fazem parte de uma mesma *thread* de execução.
2. b lê um valor escrito por a.
3. As operações são transitivas (se a acontece antes de b e b acontece antes de c então a acontece antes de c)

A definição de *acontece antes* pode ser traduzida para uma relação entre operações. A *thread* de execução mencionada na condição 1 é representada pela ordem de sessão so.

A propriedade de leitura da condição 2 pode ser representada pela relação de visibilidade vis. Desta forma a relação hb pode ser definida como a união dos conjuntos de operações de mesma sessão e de operações visíveis umas as outras. Formalmente:

$$\mathsf{hb} \stackrel{\text{def}}{=} \mathsf{so} \cup \mathsf{vis} \tag{3.10}$$

A definição de hb representa *causalidade potencial*, isto é, toda operação que ocorreu até um determinado ponto é uma potencial causa para todas as futuras operações. Na prática isto nem sempre é verdade, por exemplo, a resposta que um usuário escreve para outro em uma aplicação social não tem causa em todas as outras mensagens que este mesmo usuário visualizou anteriormente. Por este motivo a noção de *causalidade explícita* (BAILIS et al., 2012; BAILIS et al., 2013) foi introduzida delegando à aplicação a responsabilidade de definir as causas de cada operação. Até o momento, a causalidade explícita não foi definida com o modelo formal de operações e relações.

A relação hb é aplicada na formulação do modelo de consistência ConsistênciaCausal. A definição do modelo envolve duas garantias de ordenação: VisibilidadeCausal e ArbitragemCausal. A primeira garante que se $a \xrightarrow{\mathsf{hb}} b$, então $a \xrightarrow{\mathsf{vis}} b$. A segunda garante que se $a \xrightarrow{\mathsf{hb}} b$, então $a \xrightarrow{\mathsf{ar}} b$. As duas garantias de ordenação podem ser expressas formalmente como:

$$\begin{aligned} \textsc{VisibilidadeCausal} &\stackrel{\text{def}}{=} \mathsf{hb} \subseteq \mathsf{vis} \\ \textsc{ArbitragemCausal} &\stackrel{\text{def}}{=} \mathsf{hb} \subseteq \mathsf{ar} \end{aligned} \tag{3.11}$$

A junção de VisibilidadeCausal e ArbitragemCausal implica em todas as garantias de ordenação apresentadas até aqui. Logo, ConsistênciaCausal pode ser formalmente definida como:

$$\textsc{ConsistênciaCausal} \stackrel{\text{def}}{=} \textsc{VisibilidadeCausal} \wedge \textsc{ArbitragemCausal} \tag{3.12}$$

3.3.4 Consistência Forte

Consistência causal, apesar de possuir garantias bastante restritas, ainda permite anomalias. Especificamente, o modelo de consistência causal não garante uma *ordem única global* das operações. O *Teste de Dekker* (BURCKHARDT, 2014) é formulado para demonstrar justamente isso. A Tabela 3 descreve o pseudocódigo do teste. O funcionamento do teste demonstra que se um sistema permite que "A venceu" e "B venceu" seja escrito, o sistema não garante uma ordem única global das operações.

Os modelos de consistência tidos como "fortes" forçam uma ordenação única global das operações. Os modelos de consistência forte são conhecidos e estudados há mais tempo do que os modelos de consistência "fraca". O principal deles é a *Linearização* (HERLIHY; WING, 1990), que engloba as semânticas de cópia única dos dados. O outro modelo

Programa A	Programa B
1 $x \leftarrow$ "ativo"; 2 **se** $y = \varnothing$ **então** 3 $\quad$ **escreve** "A venceu"; 4 **fim**	1 $y \leftarrow$ "ativo"; 2 **se** $x = \varnothing$ **então** 3 $\quad$ **escreve** "B venceu"; 4 **fim**

Tabela 3 – Pseudocódigo do teste de Dekker (BURCKHARDT, 2014)

apresentado, *Consistência Sequencial*, é um pouco menos restrito mas ainda preserva a ordem única global das operações. Sistemas que garantam qualquer um dos modelos fortes passam no teste de Dekker.

Tal ordem única global de operações é formulada através da garantia de ordenação OrdemÚnica. A garantia é expressa requerendo que a visibilidade seja igual a arbitragem para todas as operações que tenham completado. Por exemplo, se uma arbitragem corresponder a tempos lógicos entre operações, então a garantia implica que uma operação somente pode visualizar operações com tempos lógicos menores, e deve visualizar todas as operações com tempos lógicos menores (BURCKHARDT, 2014).

Para formalizar a garantia, primeiramente define-se operações incompletas como as operações com $oval = \nabla$. Após, subtrai-se o conjunto de tais operações incompletas H' de arbitragem ($\mathsf{ar} \setminus (H' \times H)$), e iguala este sub-conjunto à visibilidade. Assim OrdemÚnica pode ser definido formalmente como:

$$\text{OrdemÚnica} \stackrel{\text{def}}{=} \exists H' \subseteq \{op \in H \mid op.oval = \nabla\} : \mathsf{vis} = \mathsf{ar} \setminus (H' \times H) \tag{3.13}$$

A garantia OrdemÚnica associada a garantia LerMinhasEscritas implica nas garantias VisibilidadeCausal e ArbitragemCausal. Assim, o modelo de consistência ConsistênciaSequencial pode ser definido formalmente como:

$$\text{ConsistênciaSequencial} \stackrel{\text{def}}{=} \text{LerMinhasEscritas} \wedge \text{OrdemÚnica} \tag{3.14}$$

O modelo de consistência sequencial apesar de muito restrito ainda não captura o comportamento de cópia única dos dados. Para tal a ordem de retorno global das operações deve ser respeitada. Esta garantia é expressa requerendo que a relação rb (retorna antes) seja usada como resolução de conflitos, ou arbitragem. Portanto, define-se a garantia TempoReal como:

$$\text{TempoReal} \stackrel{\text{def}}{=} \mathsf{rb} \subseteq \mathsf{ar} \tag{3.15}$$

A implementação de TempoReal requer relógios físicos perfeitamente sincronizados, o que implica em grandes desafios a nível de projeto de sistemas distribuídos. A junção de TempoReal e OrdemÚnica implica em LerMinhasEscritas. Assim pode-se definir o modelo de consistência Linearização como:

$$\text{Linearização} \stackrel{\text{def}}{=} \text{OrdemÚnica} \wedge \text{TempoReal} \tag{3.16}$$

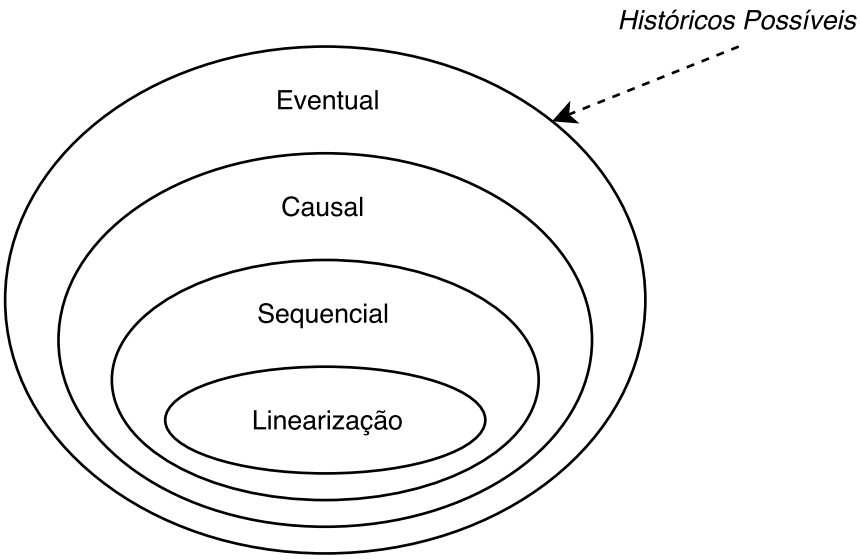


Figura 4 – Conjuntos de históricos possíveis sob cada modelo de consistência

3.3.5 Hierarquia dos Modelos de Consistência

Com diferentes modelos de consistência provendo diferentes semânticas, é natural a necessidade de uma comparação objetiva entre os modelos. Uma hierarquia de modelos de consistência pode ser criada utilizando uma ordem de "força" entre os modelos. Um modelo de consistência A é dito "mais forte" que B se todos os históricos válidos sob A também sejam válidos sob B (BURCKHARDT, 2014). Com as implicações entre as garantias de ordenação descritas na Tabela 4, a seguinte hierarquia pode ser observada:

Linearização
> ConsistênciaSequencial
> ConsistênciaCausal
> ConsistênciaEventualBásica

Sob esta hierarquia, todos os possíveis históricos válidos sob um determinado modelo também são válidos para todos os modelos mais fracos. Por exemplo, todos os históricos válidos sob ConsistênciaCausal também são válidos sob ConsistênciaEventualBásica. Já os históricos válidos sob Linearização são válidos em todos os demais modelos. A Figura 4 ilustra esta propriedade.

Por fim, a Tabela 4 resume as implicações entre as garantias de ordenação e as formulações dos modelos de consistência. As especificações das garantias de ordenação e dos modelos de consistência descritas neste capítulo habilitam a análise precisa da consistência provida por sistemas de armazenamento de dados. O próximo capítulo faz uso dessas especificações para analisar projetos de sistemas de armazenamento de dados amplamente utilizados.

Implicações entre as Garantias de Ordenação
$(\textsc{OrdemÚnica} \wedge \textsc{TempoReal}) \Rightarrow \textsc{LerMinhasEscritas}$
$(\textsc{OrdemÚnica} \wedge \textsc{LerMinhasEscritas}) \Rightarrow \textsc{ArbitragemCausal}$
$(\textsc{OrdemÚnica} \wedge \textsc{LerMinhasEscritas}) \Rightarrow \textsc{VisibilidadeCausal}$
$(\textsc{VisibilidadeCausal} \wedge \textsc{ArbitragemCausal}) \Rightarrow \textsc{LerMinhasEscritas}$
$(\textsc{VisibilidadeCausal} \wedge \textsc{ArbitragemCausal}) \Rightarrow \textsc{LeiturasMonotônicas}$
$(\textsc{VisibilidadeCausal} \wedge \textsc{ArbitragemCausal}) \Rightarrow \textsc{VisibilidadeEventual}$
$(\textsc{VisibilidadeCausal} \wedge \textsc{ArbitragemCausal}) \Rightarrow \textsc{AusênciaDeCausalidadeCircular}$

Modelos de Consistência
$\textsc{Linearização} \stackrel{\text{def}}{=} \textsc{TempoReal} \wedge \textsc{OrdemÚnica}$
$\textsc{ConsistênciaSequencial} \stackrel{\text{def}}{=} \textsc{LerMinhasEscritas} \wedge \textsc{OrdemÚnica}$
$\textsc{ConsistênciaCausal} \stackrel{\text{def}}{=} \textsc{VisibilidadeCausal} \wedge \textsc{ArbitragemCausal}$
$\textsc{ConsistênciaEventualBásica} \stackrel{\text{def}}{=} \textsc{VisibilidadeEventual} \wedge \textsc{AusênciaDeCausalidadeCircular}$

Tabela 4 – Resumo das garantias de ordenação e modelos de consistência

4 Soluções para Sistemas de Armazenamento de Dados com Alta Carga de Leitura

Este capítulo aborda as duas principais soluções encontradas na literatura e em uso na indústria para sistemas de armazenamento de dados com cargas de trabalho intensivas em leituras: replicação assíncrona com cópia primária e *cache-aside*. Afim de normalizar as comparações entre as soluções, as mesmas são descritas sob a forma de tipos de dados replicados. Estes simplificam a descrição das soluções e possibilitam comparações mais objetivas. Primeiramente, o conceito de tipo de dados replicado é descrito. Após, as duas soluções são descritas sob a forma de tipos de dados replicados. Cada solução tem seu modelo de consistência e utilização de rede analisados. Anomalias de consistência são identificadas com base em possíveis execuções condizentes com os algoritmos das soluções. As relações de arbitragem e visibilidade são derivadas e a partir delas as garantias de ordenação atendidas são identificadas. Por fim, o modelo de consistência provido por cada solução é identificado.

Para fins das soluções e algoritmos apresentados neste capítulo assume-se que processos executam operações de forma serializada, ou seja, caso duas operações sejam submetidas de forma concorrente para um mesmo processo uma das operações deve aguardar o término da outra para ser processada. Falhas internas nos processos ou no canal de comunicação entre os processos não são consideradas.

Os pseudocódigos apresentados fazem uso de estruturas de dados de propósito geral como *Mapas* e *Filas FIFO* (*First In First Out*). *Mapas* são conjuntos de objetos representados pela tupla $\langle Chave, Valor \rangle$. Dado um mapa M, $M[Chave]$ retorna o *Valor* associado a *Chave*. $M[k] \leftarrow v$ associa o valor v à chave k. Uma chave não presente no mapa possui valor $\varnothing$. Remoções de objetos no *Mapa* são feitas atribuindo $\varnothing$ a uma chave, ou $m[k] \leftarrow \varnothing$. Modificações no mapa são representadas pela tupla $\langle k, v, p \rangle$, onde k é uma chave, v um valor e p um inteiro que representa o tempo lógico em que a operação foi realizada. As filas FIFO são listas que suportam operações de enfileiramento e desenfileiramento. O enfileiramento insere um elemento na estrutura. O desenfileiramento remove e retorna o elemento mais antigo inserido na estrutura.

4.1 Tipos de Dados Replicados

Projetos de sistemas de armazenamento de dados são complexos e compreendem muitas funcionalidades. Afim de estudar aspectos específicos destes sistemas, abstrações são utilizadas para isolar estes aspectos. No caso do estudo de replicação e consistência,

que é o foco deste trabalho, *tipos de dados replicados* são utilizados como forma de abstração. Os tipos de dados replicados tem sido o foco de diferentes trabalhos que abordam replicação e consistência em sistemas distribuídos (SHAPIRO et al., 2011b; ROH et al., 2011; BURCKHARDT et al., 2014; BURCKHARDT et al., 2015). Eles permitem que o projeto destes sistemas sejam analisados e verificados com o mínimo de funcionalidades necessárias. Projetos de tipos de dados replicados podem posteriormente servir como base para a implementação de sistemas de armazenamento de dados que miram o uso em produção [1]

Um tipo de dados replicado retém as propriedade básicas de um sistema de armazenamento de dados distribuído: um conjunto de processos que se comunicam através de troca de mensagens e recebem invocações de operações por processos clientes. O projeto de um tipo de dados replicado possui um *modelo de dados* e um *protocolo de replicação*. O modelo de dados é a semântica de manipulação de dados oferecida aos clientes. Já o protocolo de replicação é algoritmo pelo qual os processos que compõem o tipo de dados trocam e manipulam mensagens.

Os modelos de dados são representações de tipos de dados comuns de instância única como registradores, filas ou mapas. O modelo mais comum para análise e especificação de sistemas de armazenamento de dados distribuídos é o *Armazenamento Chave-Valor* (do inglês *Key-Value Store*). Este é um tipo de dados de propósito geral que tem a mesma semântica de um *Mapa*. O armazenamento chave-valor é amplamente utilizado em sistemas de armazenamento de objetos em ambientes *cloud* (BURCKHARDT et al., 2015) e pode ser utilizado como base para o projeto de modelos de dados que oferecem semânticas mais complexas (BAILIS et al., 2013). Neste trabalho, afim de normalizar as comparações, todos os tipos de dados replicados apresentados serão armazenamentos chave-valor.

A forma como os diferentes processos que compõem um tipo de dados replicado trocam mensagens afim de replicar seus estados determina um *protocolo de replicação*. Um protocolo de replicação é definido por um conjunto de *papéis* (do inglês, *role*). Cada processo participante do tipo de dados pode assumir um único papel durante a sua execução. Os papéis determinam o comportamento dos processos quando os mesmos recebem uma mensagem de outro processo ou uma requisição de um cliente. Neste trabalho os papéis são descritos na forma de pseudocódigos.

Além do pseudocódigo dos diferentes tipos de papéis envolvidos num protocolo de replicação, as garantias de transporte requeridas são de grande importância na análise destes protocolos. Burckhardt (2014) define um conjunto de garantias de transporte que podem ser requeridos por protocolos de replicação. Tal conjunto inclui garantias de segurança (do inglês, *safety*), que previnem o forjamento, perda, reordenação e duplicação das

[1] O Riak (http://basho.com/products/riak-kv/) é um projeto comercial que utiliza os tipos de dados replicados CRDTs (ver subseção 6.1.3) como base na sua implementação

mensagens; e garantias de vivacidade (do inglês, *liveness*), que garantem que as mensagens sejam eventualmente entregues direta ou indiretamente. Os protocolos apresentados neste trabalho, salvo quando explicitamente mencionado, requerem as seguintes garantias de transporte:

- *sem-forja*: todas as mensagens devem ser geradas por processos não falhos participantes do protocolo.
- *sem-perda*: todas as mensagens são eventualmente entregues aos seus destinatários.
- *ordenada*: a ordem das mensagens entre emissor e receptor é a mesma.

As seções seguintes analisam soluções existentes sob a forma de tipos de dados replicados. Os protocolos de replicação das soluções são descritos com base em projetos de código aberto e práticas comuns da indústria. Para cada solução, os pseudocódigos dos diferentes papéis possibilitam a visualização de possíveis execuções. Estas execuções possibilitam a análise de garantias de ordenação e modelos de consistência. Os protocolos de replicação também possibilitam a análise do uso de rede de cada solução. Métricas como quantidade de mensagens trocadas por operação podem ser derivadas a partir dos protocolos.

4.2 Replicação Assíncrona com Cópia Primária

A replicação assíncrona com cópia primária é uma das formas mais simples de escalar horizontalmente um sistema de armazenamento de dados. Neste modelo de replicação uma lógica de resolução de conflitos não é necessária, uma vez que escritas não são executadas de forma concorrente entre diferentes processos. A facilidade na implementação fez com que este modelo de replicação se tornasse um modelo arquitetural popular em aplicações comercias. Implementações de código aberto de bases de dados relacionais (RDBMS, do inglês *Relational Database Management System*) amplamente utilizadas como MySQL e PostgreSQL implementam o modelo [2]. Na indústria e comunidades de código aberto o modelo é normalmente chamado de *Master-Slave*.

Esta seção dá continuidade às definições apresentadas no Capítulo 2 sobre replicação assíncrona com cópia primária. Todas as definições apresentadas até então para a solução continuam se aplicando para as definições apresentadas neste capítulo. A Figura 5 ilustra uma arquitetura com uma réplica primária e duas secundárias. Além das réplicas, a figura ilustra as interações entre clientes e *middleware*. As requisições feitas pelos clientes são intermediadas pela *middleware* que roteia todas as operações de escrita para a réplica

[2] Documentações detalhando modelos de replicação no MySQL e PostgreSQL podem ser encontradas respectivamente em: https://dev.mysql.com/doc/refman/5.7/en/replication.html e https://www.postgresql.org/docs/9.2/static/high-availability.html.

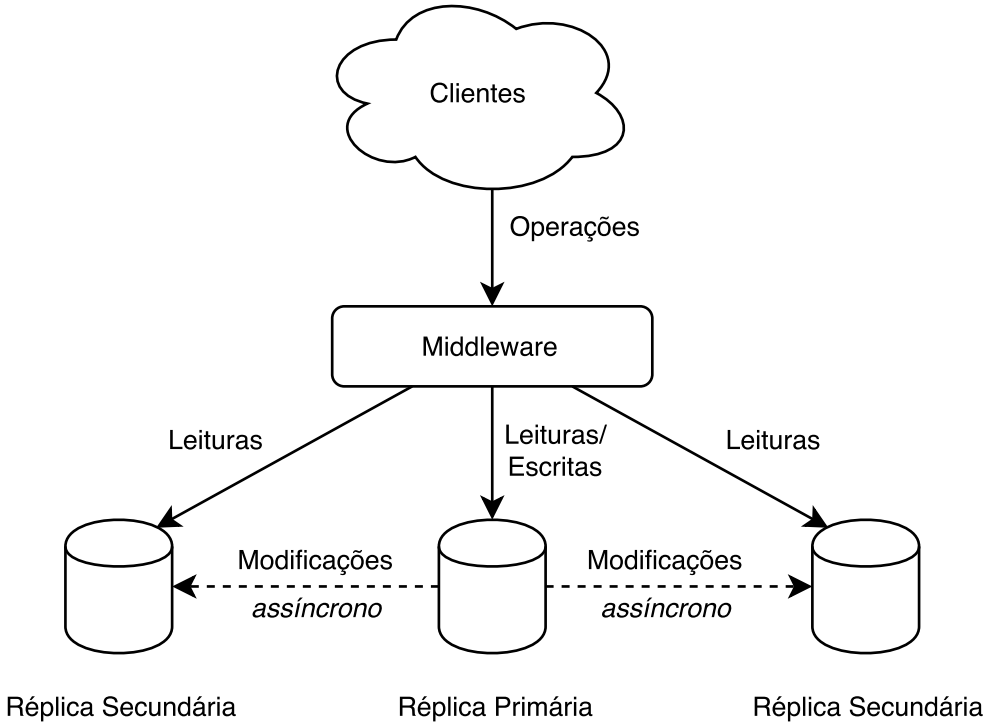


Figura 5 – Visão geral da cópia primária com replicação assíncrona

primária. Nas operações de leitura a *middleware* age como um balanceador de carga entre os três processos.

Com base nas definições apresentadas por Cecchet (2008), o protocolo de um tipo de dados replicado que implementa a replicação assíncrona com cópia primária pode ser derivado com os seguintes papéis:

- Uma réplica primária responsável por processar todos os tipos de operações e enviar modificações para todas as réplicas secundárias.
- N réplicas secundárias responsáveis por processar operações de leitura.
- Uma *middleware* responsável por receber as requisições dos clientes e rotear as operações de escrita para a réplica primária e leitura para as secundárias e primária.

A replicação dos dados entre a cópia primária e as secundárias é feita através do envio assíncrono e ordenado das escritas realizadas na réplica primária para as secundárias. O envio pode ocorrer periodicamente através de lotes de operações, ou assim que cada escrita for realizada. Por via de regra, a totalidade das escritas é replicada.

O pseudocódigo da réplica primária é descrito pelo algoritmo 1. O processo pode receber dois tipos de mensagens: GET pera leituras e PUT para escritas. Ao receber um GET, o processo apenas retorna o valor da chave solicitada encapsulado em uma mensagem GETRESP para a *middleware*. Ao receber um PUT, o processo atualiza a seu

armazenamento local com a chave e valor informados e retorna um ACK para a *middleware*. Somente após o envio do ACK o processo itera sobre as réplicas secundárias e envia um PUT contendo o objeto escrito para cada uma. As implementações podem variar, as mensagens individuais poderiam ser substituídas por um *broadcast* ou *multicast* por exemplo. Um dos mecanismos de replicação do PostgreSQL trabalha com as réplicas secundárias proativamente consultando a primária sobre novas atualizações[3]. Porém, a idéia principal da solução é mantida: a replicação ocorre após a confirmação de todas as escritas executadas.

```
Dados: A: armazenamento local;
       R: conjunto de réplicas secundárias;
       m: middleware;

1  Mensagem GET(k: Chave);
2  Mensagem GETRESP(v: Valor);
3  Mensagem PUT(k: Chave, v: Valor);
4  Mensagem ACK();

5  Recebe GET(k):
6  |  Envia GETRESP(A[k]) para m;
7  fim

8  Recebe PUT(k, v):
9  |  A[k] ← v;
10 |  Envia ACK para m;
11 |  para cada r ∈ R faça
12 |  |  Envia PUT(k, v) para r;
13 |  fim
14 fim
```

Algoritmo 1: Pseudocódigo da réplica primária na replicação assíncrona com cópia primária

A lógica de execução dos demais processos é simples e não contribui na análise de consistência da solução. Por tal motivo o pseudocódigo desses processos não é apresentado. As réplicas secundárias são sistemas de armazenamento de propósito geral que aceitam leituras e escritas. O *middleware* executa uma lógica de roteamento simples baseado no tipo da operação. Usos da estratégia para fins como *failover* certamente adicionam lógicas no sistema como um todo, porém estas também não afetam o modelo de consistência da solução.

[3] Documentação sobre a funcionalidade de *Replicação Lógica* do PostgreSQL disponível em: https://www.postgresql.org/docs/10/static/logical-replication.html

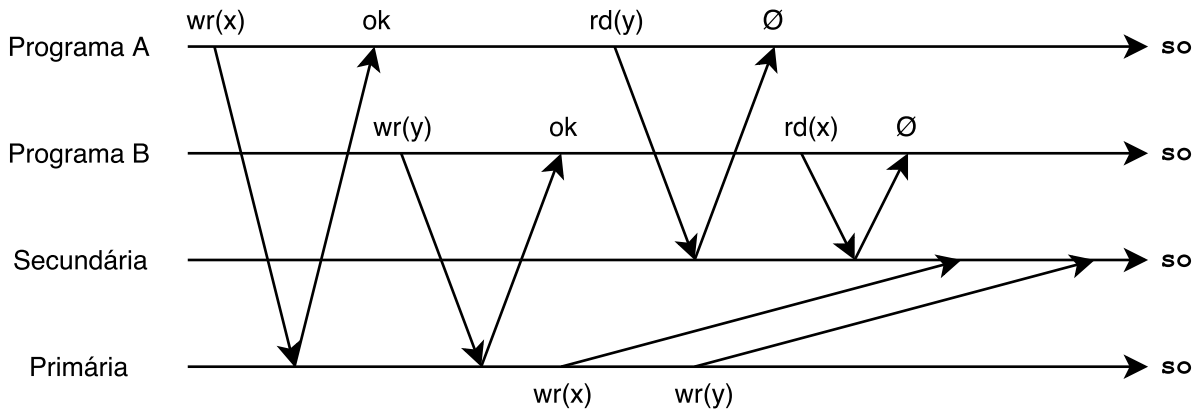


Figura 6 – Possível execução do teste de Dekker em um sistema de cópia primária com replicação assíncrona

4.2.1 Modelo de Consistência

Na análise do modelo de consistência provido pela replicação assíncrona com cópia primária, os modelos de consistência forte podem ser prontamente descartados. O motivo pode ser observado pela execução do teste de Dekker ilustrado pela Figura 6. A figura mostra os dois programas A e B do teste interagindo com uma cópia primária e uma secundária. As setas entre os processos representam as trocas de mensagens entre os mesmos. Os envios e recebimentos das mensagens estão ordenados pela ordem de sessão so de cada processo. Na Figura, devido a um possível atraso na comunicação entre os processos que mantém as cópias primária e secundária, a replicação não ocorreu a tempo e as duas leituras retornaram $\varnothing$, o que significa uma falha no teste.

A execução da Figura 6 não aponta somente uma falha ao atender a garantia de ordenação OrdemÚnica, que é o propósito do teste de Dekker. Caso as duas leituras tivessem como parâmetros os objetos que cada processo acabou de escrever, as duas leituras continuariam retornando $\varnothing$. Este cenário mostra que a ordem de sessão não é um subconjunto da visibilidade ($\mathsf{so} \not\subset \mathsf{vis}$), o que de acordo com a Equação 3.8 viola a garantia LerMinhasEscritas. Pode-se inferir também que se $\mathsf{so} \not\subset \mathsf{vis}$, logo $(\mathsf{so} \cup \mathsf{vis}) \not\subset \mathsf{vis}$, o que de acordo com as Equações 3.10 e 3.11 viola VisibilidadeCausal, o que descarta o modelo de consistência ConsistênciaCausal.

Como todas as operações de escrita são executadas pelo processo responsável pela cópia primária, a arbitragem pode ser definida como a ordem de sessão deste processo, ou $\mathsf{so}|_{primary}$. Com esta definição, pode-se inferir de acordo com as Equações 3.10 e 3.11 que a solução atende a garantia ArbitragemCausal.

A execução ilustrada pela Figura 7 demonstra por contradição que a garantia LeiturasMonotônicas não é atendida. Na execução um cliente se comunica com um sistema com duas cópias secundárias A e B. Após uma escrita, duas leituras são disparadas.

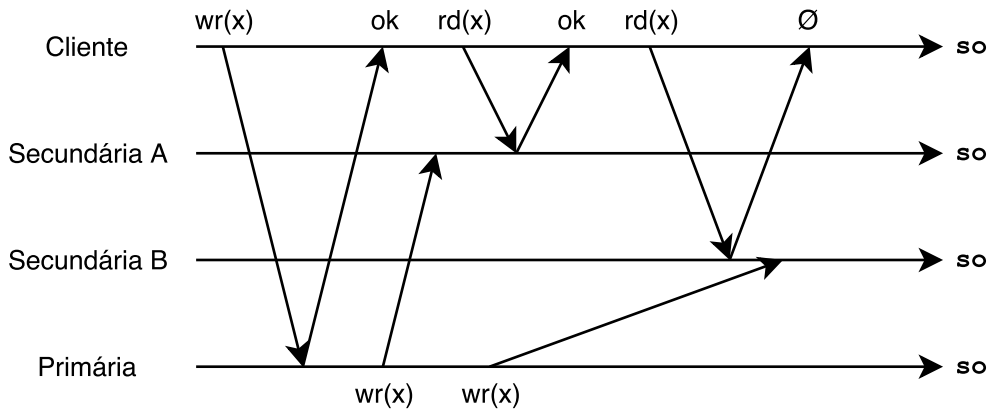


Figura 7 – Execução em um sistema de cópia primária com replicação assíncrona com dois processos secundários

A *middleware* (omitida na Figura) roteia cada uma das leituras para uma cópia secundária diferente. Um atraso na replicação para cópia secundária B fez com que a leitura retornasse ∅, ao contrário da leitura roteada para cópia secundária A. Esta execução, de acordo com a Equação 3.9 viola a garantia LeiturasMonotônicas.

Por fim, a garantia VisibilidadeEventual é atendida uma vez que a cópia primária envia todas as escritas para as secundárias. O que faz com que o protocolo atenda as garantias VisibilidadeEventual e ArbitragemCausal, o que de acordo com a Tabela 4 é suficiente para atender o modelo de consistência ConsistênciaEventualBásica.

4.2.2 Utilização de Rede

Este trabalho utiliza como métrica de uso de rede a quantidade de mensagens trocadas entre os processos participantes do sistema de armazenamento de dados. A quantidade de mensagens trocadas é medida por operação executada. Os tamanhos das mensagens trocadas são diretamente dependentes dos tamanhos dos objetos armazenados pelo sistema e por tal motivo não serão considerados por este trabalho.

Nas soluções abordadas neste trabalho, diferentes tipos de operações requerem a troca de diferentes quantidades de mensagens. Isto faz com que a carga de trabalho aplicada no sistema tenha relevância na análise de utilização de recursos de rede. As cargas de trabalho devem ser categorizadas de acordo com a proporção de operações de leitura e escrita. Uma carga de trabalho com proporções iguais de leituras e escritas apesar de representar uma carga de propósito geral, não representa um caso de uso típico para o tipo de solução abordado neste trabalho. Um trabalho recente realizou uma análise de diferentes cargas de trabalho em sistemas de armazenamento de dados em uso pelo Facebook (ATIKOGLU et al., 2012). As cargas de trabalho apresentam forte inclinação

para leituras, com casos extremos apresentando 99% de leituras. Baseados nos dados coletados, os autores concluem que uma carga de trabalho sintética de propósito geral para este tipo de sistema é compreendida de uma proporção de 30 leituras para cada escrita. No decorrer deste trabalho são apresentadas métricas das soluções abordadas com cargas de trabalho de proporção 1:1 e 30:1 entre leituras e escritas.

Na replicação assíncrona com cópia primária, todas as operações são enviadas para a *middleware* que, por sua vez, roteia a operação para uma das réplicas. Todas as operações de leitura são completadas com 4 mensagens trocadas entre a *middleware* e a réplica selecionada para processar a operação. As escritas são retornadas para os clientes com as mesmas 4 mensagens das leituras, porém o processo de replicação que ocorre após o retorno da operação acarreta em mais N mensagens, onde N é o número de réplicas secundárias do sistema. Logo, em um sistema com N réplicas escritas geram $4 + N$ mensagens.

Dadas as proporções de leituras e escritas juntamente com as quantidades de mensagens trocadas por tipo operação, a quantidade média de mensagens por operação A de uma determinada solução pode ser definida como:

$$A = \frac{P_{rd} \cdot M_{rd} + P_{wr} \cdot M_{wr}}{P_{rd} + P_{wr}} \tag{4.1}$$

Onde P_{rd} e P_{wr} representam as proporções de leituras e escritas da carga de trabalho e M_{rd} e M_{wr} representam as quantidades de mensagens trocadas por leituras e escritas da solução.

Quando consideradas as cargas de trabalho com proporções de 1:1 e 30:1 entre leituras e escritas, as quantidades médias de mensagens trocadas por operação na replicação assíncrona com cópia primária são:

- Proporção 1:1 entre leituras e escritas: $4 + \frac{N}{2}$
- Proporção 30:1 entre leituras e escritas: $4 + \frac{N}{31}$

4.3 *Cache-Aside*

A replicação assíncrona com cópia primária apesar de permitir que a carga de leitura de um sistema seja distribuída entre diferentes processos, não contribui com o desempenho individual de cada processo. Como as cópias secundárias armazenam uma cópia completa dos dados, operações de leitura tendem a ter desempenhos semelhantes em ambos os tipos de processos. Outro ponto negativo está no fato de que cargas de trabalho fortemente inclinadas para leituras de um pequeno subconjunto de objetos não se beneficiam de modelos que replicam o conjunto total dos dados.

Estas deficiências motivaram a adoção de uma solução que prioriza a leitura de um pequeno sub-conjunto dos dados de um sistema. Este sub-conjunto é dinamicamente modificado para conter somente os objetos com a maior probabilidade de acesso, tal qual um *cache* de processador. Nesta solução o sub-conjunto mantido em cache é normalmente armazenado em um tipo de processo diferente do que armazena a cópia principal dos dados. A lógica de replicação entre os dois processos fica a cargo da aplicação que utiliza os dados. Esta solução é amplamente utilizada na indústria, especialmente em aplicações *web* e normalmente chamada de *Cache-Aside* (NISHTALA et al., 2013).

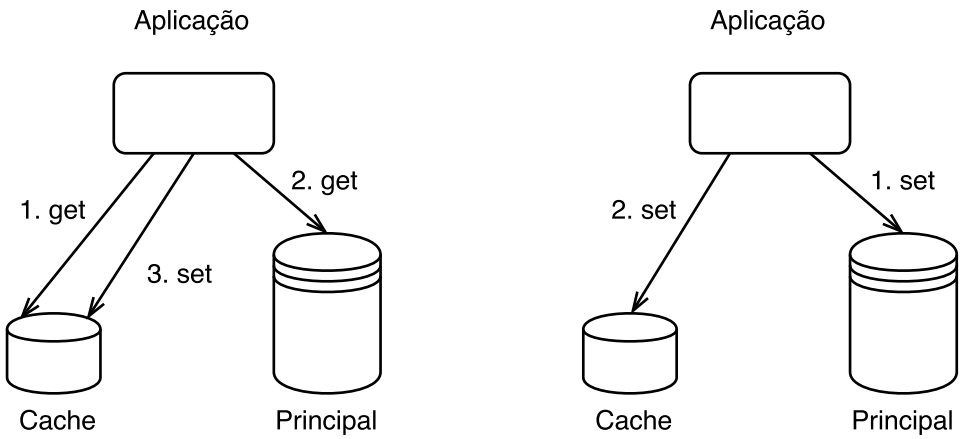
O protocolo de um tipo de dados replicado derivado do *cache-aside* possui três papéis:

- Um armazenamento "principal". Este processo é normalmente projetado para garantir durabilidade dos dados. A prática mais comum na indústria é utilizar uma base de dados relacional (RDBMS, do inglês *Relational Database Management System*) como armazenamento principal.
- Um armazenamento "cache". Este é um sistema de armazenamento otimizado para operações de leitura, por tal motivo o sistema normalmente mantém os dados em memória RAM. Isto faz com que o sistema tenha uma capacidade armazenamento reduzida e necessite de um mecanismo de evicção de dados. Tal mecanismo trabalha eliminando itens presentes no armazenamento a medida em que novos itens são inseridos. Além da evicção os objetos possuem um tempo máximo finito de permanência no cache. Caso um objeto ultrapasse esse período, o mesmo é removido do cache, Dentre os sistemas de cache em memória normalmente usados na indústria estão o Memcached [4] e o Redis [5], ambos de código aberto.
- A aplicação cliente do sistema. Nesta solução o processo responsável por replicar dados entre os dois sistemas de armazenamento é uma aplicação que utiliza o armazenamento. Como a solução é normalmente utilizada em aplicações *web*, este é normalmente o processo que recebe requisições HTTP dos clientes.

O fluxo básico de leitura é ilustrado pela Figura 8a. No fluxo a aplicação primeiramente verifica se o objeto solicitado existe no cache, caso sim o objeto é utilizado. Este cenário é chamado de *cache-hit*. Caso o objeto não exista no cache, uma operação de leitura é submetida para o armazenamento principal. O objeto retornado é então inserido no cache e utilizado pela aplicação. Este segundo cenário é chamado de *cache-miss*.

Operações de escrita seguem o fluxo ilustrado pela Figura 8b. Primeiramente uma operação de escrita é submetida para o armazenamento principal e após, a mesma operação é submetida ao cache.

[4] https://memcached.org/
[5] https://redis.io/



(a) Leitura *(b) Escrita*

Figura 8 – Fluxos de leitura (a) e escrita (b) do *cache-aside* (NISHTALA et al., 2013)

O pseudocódigo referente ao papel da aplicação é descrito no algoritmo 2. A troca de mensagens entre os processos trabalha de forma semelhante a um mecanismo RPC (*Remote Procedure Call*), ou seja, o envio de uma mensagem bloqueia a execução do processo até que a mensagem de resposta seja recebida. Por tal motivo, por simplicidade, os processos principal e cache são representados no pseudocódigo como *mapas*. O pseudocódigo descreve o tratamento das operações de leitura GET e escrita PUT. No GET a chave informada é buscada no *cache*, caso não exista a mesma chave é buscada no armazenamento principal. Após, o valor é retornado ao cliente. No PUT o objeto é inserido armazenamento principal, inserido no cache e a operação é retornada ao cliente.

Dados: *DB*: armazenamento de dados principal;
Cache: armazenamento para leituras;

```
1  Função GET(k: Chave):
2      v ← Cache[k];
3      se v = ∅ então
4          v ← DB[k];
5          Cache[k ↦ v];
6      fim
7      retorna v;
8  fim
9  Função PUT(k: Chave, v: Valor):
10     DB[k ↦ v];
11     Cache[k ↦ v];
12     retorna
13 fim
```

Algoritmo 2: Pseudocódigo base de uma aplicação na solução *cache-aside*

No *cache-aside*, tanto o armazenamento principal quanto o cache são processos de armazenamento de dados de propósito geral e não possuem nenhuma lógica específica. O código apresentado trata da parte genérica de comunicação com os armazenamentos de dados que uma aplicação deve implementar. Aplicações reais terão suas regras e lógicas específicas, porém, os fluxos básicos de comunicação são mantidos.

4.3.1 Modelo de Consistência

Na análise da consistência do *cache-aside* primeiramente consideramos uma configuração onde existe apenas uma instância de cada papel. Nesta configuração o único processo a ter acesso ao armazenamento principal é o processo aplicação. Pode-se observar que nesta configuração todas as operações são originadas do processo aplicação. Isto faz com que o processo da aplicação aja como um único cliente do sistema de armazenamento. A aplicação por sua vez, atende a vários outros clientes, porém o acesso aos dados é serializado. Assim o modelo mantém a mesma semântica de um processo interagindo com uma única cópia dos dados. Logo, a seguinte relação é mantida:

$$\mathsf{vis} = \mathsf{ar} = \mathsf{hb}$$

O que de acordo com as Equações 3.15 e 3.13 atende as garantias de ordenação TempoReal e OrdemÚnica. Logo, o modelo de consistência Linearização é atendido.

A configuração com uma instância de cada papel e com acesso restrito ao armazenamento principal é válida e certamente tem seus motivadores de uso. Uma aplicação web simples que atua como único cliente de uma base de dados é um exemplo deste tipo de configuração. Porém, uma análise completa do modelo deve considerar outras possíveis configurações. A seguir é analisada a consistência de configurações onde outros processos podem manipular objetos no armazenamento principal. Estas configurações podem ser generalizadas para configurações de múltiplos caches. Os demais processos podem ser aplicações com seus respectivos caches.

Em configurações com múltiplos processos manipulando objetos no armazenamento principal a linearização não é mantida. O motivo pode ser visualizado pela execução ilustrada pela Figura 9. Na Figura, o processo *Aplicação B* submete três operações de escrita $wr(x,1)$, $wr(x,2)$ e $wr(y,1)$. De acordo com a definição de hb, tem-se $wr(x,1) \xrightarrow{\mathsf{hb}} wr(x,2) \xrightarrow{\mathsf{hb}} wr(y,1)$. Da forma como a interação entre aplicação e cache funciona, as futuras leituras dos objetos x e y pela *Aplicação A* retornarão 1 enquanto não houverem evicções dos dois objetos. Isto demonstra que das três operações de escrita, a segunda ($wr(x,2)$) não é visível ao processo *Programa A*. Esta execução demonstra que as garantias OrdemÚnica e VisibilidadeCausal não são atendidas. Consequentemente, os modelos de consistência forte e causal não são atendidos quando se tem múltiplos processos acessando o armazenamento principal.

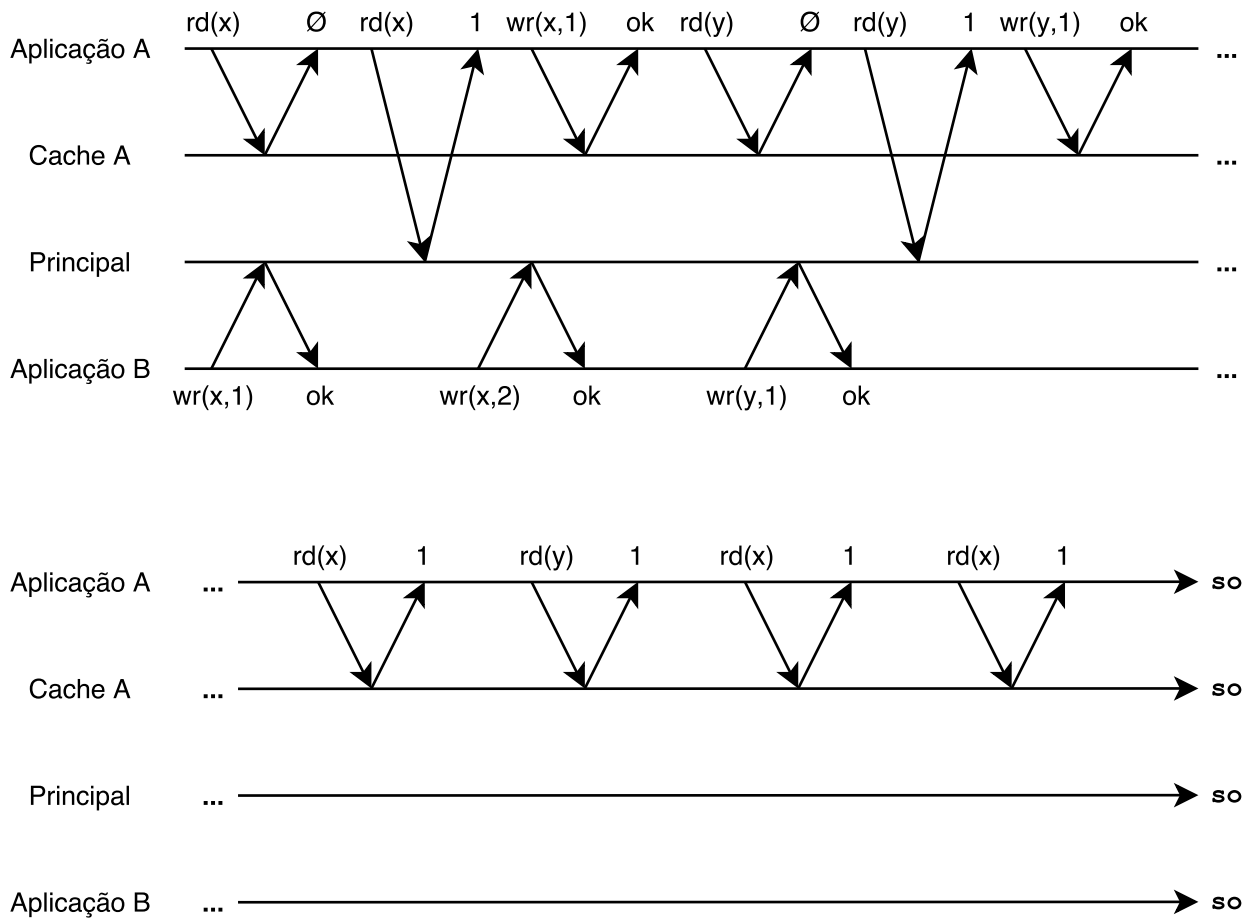


Figura 9 – Execução de *cache-aside* com dois clientes acessando o armazenamento principal

As demais garantias de ordenação continuam sendo atendidas com múltiplos processos acessando o armazenamento principal. As garantias de sessão LerMinhasEscritas e LeiturasMonotônicas são atendidas, pois o cache não interfere na visibilidade das operações submetidas pela aplicação. A garantia ArbitragemCausal é atendida pois todas as operações de escrita passam pelo armazenamento principal. Por fim a garantia VisibilidadeEventual é garantida pela fato de que o tempo máximo de permanência dos objetos no cache é finito. O conjunto das garantias atendidas faz com que o modelo de consistência provido seja o ConsistênciaEventualBásica de acordo com a Tabela 4.

4.3.2 Utilização de Rede

No *cache-aside*, todas as escritas são completadas com uma troca de mensagens entre aplicação e armazenamento principal seguido de outra troca entre aplicação e *cache*. O que totaliza 4 mensagens para qualquer operação de escrita. Já as leituras apresentam quantidades de mensagens trocadas diferentes para *cache-hits* e *cache-misses*. Nos *cache-hits* uma comunicação simples entre aplicação e *cache* basta, o que gera 2 mensagens trocadas. Já nos *cache-misses* três comunicações distintas são feitas, o que gera um total

Repl. Assíncrona com Cópia Primária	*Cache-Aside*
VisibilidadeEventual ArbitragemCausal	VisibilidadeEventual LerMinhasEscritas LeiturasMonotônicas ArbitragemCausal

Tabela 5 – Garantias de ordenação das soluções apresentadas

de 6 mensagens trocadas.

Uma solução que se baseia em uso de *caches* como o *cache-aside* deve sempre buscar maximizar os *cache-hits*. Atikoglu (2012) em seu trabalho que analisa cargas de trabalho nos sistemas de armazenamento de dados do Facebook, conclui que uma carga de trabalho sintética de propósito geral possui 81% das leituras como *cache-hits*. Este valor será utilizado como base na análise de utilização de rede do *cache-aside*.

Quando consideradas as cargas de trabalho com proporções de 1:1 e 30:1 entre leituras e escritas, sendo 81% das leituras *cache-hits*, as quantidades médias de mensagens trocadas por operação no *cache-aside* de acordo com a Equação 4.1 são:

- Proporção 1:1 entre leituras e escritas: 3,38
- Proporção 30:1 entre leituras e escritas: 2,8

4.4 Comparações e Observações sobre as Soluções

Apesar de o modelo de consistência provido pela replicação assíncrona com cópia primária e *cache-aside* ser o mesmo (ConsistênciaEventualBásica), as garantias de ordenação providas diferem. Ambas as soluções garantem VisibilidadeEventual e ArbitragemCausal, a última é resultado direto das duas soluções se utilizarem de um processo único para controlar as escritas. A principal diferença está nas garantias de sessão providas pelas duas soluções. O cache sendo acessado exclusivamente pela aplicação é capaz de garantir que a visibilidade das operações não divirja da ordem de sessão no *cache-aside*. Esta propriedade não se mantém no caso das múltiplas réplicas secundárias na replicação assíncrona com cópia primária. A Tabela 5 resume as garantias de ordenação providas pelas duas soluções.

As duas soluções tem casos de uso bastante distintos. A replicação assíncrona com cópia primária é uma solução de propósito geral que não altera a interface de comunicação básica de clientes com o sistema. Do ponto de vista dos clientes não existe diferença entre a *middleware* e uma réplica qualquer. Já a solução cache-aside requer esforço por parte do desenvolvedor da aplicação para gerenciar os dois armazenamentos.

Uma das métricas que mais distingue as duas soluções é o uso de espaço de armazenamento. Enquanto a replicação assíncrona com cópia primária requer que as réplicas secundárias mantenham uma cópia completa do conjunto de dados do sistema, o cache-aside permite que os caches tenham tamanho pré-determinado. Isto, associado ao fato de que o cache é populado sob demanda possibilita o uso de memória RAM como armazenamento. A população sob demanda dos objetos no cache é mais eficiente para cargas de trabalho onde um pequeno subconjunto dos dados corresponde à maioria dos acessos.

O uso de recursos de rede também difere entre as duas soluções. Na replicação assíncrona com cópia primária cada operação requer no mínimo 4 mensagens para ser completada. A quantidade exata de mensagens trocadas depende de quantas réplicas secundárias existem na configuração. Já no cache-aside o número de mensagens varia de acordo com o fluxo da operação. Uma escrita requer 4 mensagens, uma leitura *cache-hit* 2 e uma leitura *cache-miss* 6. Uma comparação com mais detalhes entre o uso de rede das soluções é abordada no próximo capítulo.

Por fim, a Tabela 6 apresenta um resumo dos prós e contras das soluções apresentadas neste capítulo. Os pontos negativos mostram que ainda existe espaço para aprimoramento em ambas as soluções. No próximo capítulo uma alternativa ao *cache-aside* e replicação assíncrona com cópia primária é apresentada sob a forma de um novo projeto de tipo de dados replicado. O projeto visa atender os mesmos requisitos atendidos pelas duas soluções ao mesmo tempo em que elimina os pontos negativos apresentados pelas mesmas.

	Replicação Assíncrona com Cópia Primária	**Cache-Aside**
Prós	• Solução de propósito geral • Pode ser utilizada como *failover*	• Eficiência no uso de espaço de armazenamento • Permite uso de memória volátil • Menor uso de rede
Contras	• Cópia completa dos dados • Não provê garantias de ordenação de sessão • Consistência eventual	• Gerenciamento da replicação pelo processo da aplicação • Consistência eventual

Tabela 6 – Prós e contras das soluções apresentadas

5 Nova Solução para Sistemas de Armazenamento de Dados com Alta Carga de Leitura

As estratégias empregadas para solucionar o problema de sistemas de armazenamento de dados com alta carga de leitura apresentados no Capítulo 4 sofrem de diversos problemas. Notoriamente, o modelo de consistência provido é um declínio se levado em consideração que as soluções utilizam como fonte de dados um sistema centralizado capaz de prover consistência forte. Além da baixa consistência, projetistas de aplicações tem que escolher entre a eficiência no uso de espaço de armazenamento ou na comunicação com apenas um processo, providas pelo *cache-aside* e replicação assíncrona com cópia primária respectivamente.

Neste capítulo é apresentada uma nova solução para sistemas de armazenamento de dados com alta carga de leitura sob a forma do projeto de um tipo de dados replicado chamado de *Cache Transparente*. O *Cache Transparente* possui modelo de dados chave-valor e um protocolo de replicação que visa solucionar os principais problemas das soluções anteriormente descritas. Adicionalmente, o projeto visa manter os benefícios de ambas as soluções. O protocolo de replicação do *Cache Transparente* é capaz de distribuir a carga de leitura entre diferentes processos. Em suma, os principais objetivos do *cache transparente* são:

- Prover um modelo de dados chave-valor de propósito geral.
- Distribuir carga de leitura entre diferentes processos.
- Garantir um modelo de consistência forte.
- Permitir que parte dos processos possam armazenar um sub-conjunto dos dados do sistema.
- Prover uma interface de comunicação onde aplicações clientes se comuniquem com apenas um processo.

Nas Seções seguintes o protocolo de replicação do *Cache Transparente* é apresentado e detalhado. Pseudocódigos dos papéis envolvidos são apresentados juntamente com as descrições das principais estruturas de dados utilizadas. Após, o modelo de consistência provido é abordado e demonstrado de acordo com os fundamentos apresentados no Capítulo 3. Vários aspectos do projeto são discutidos como leituras obsoletas, possíveis otimizações e utilização de recursos. Todas as discussões utilizam as soluções abordadas no Capítulo 4 como base de comparação.

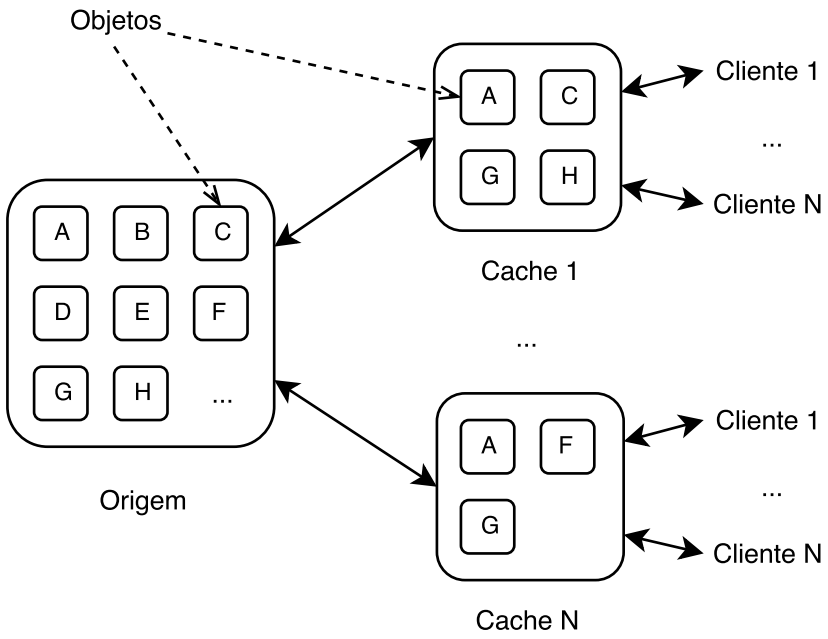


Figura 10 – Relações e cardinalidades entre *Origem*, *Caches*, clientes e objetos

5.1 Visão Geral

A idéia principal do *Cache Transparente* é semelhante às demais soluções para distribuição de carga de leitura: segregar operações de leitura e escrita entre diferentes processos. Assim como nas soluções *cache-aside* e replicação assíncrona com cópia primária, processos podem assumir dois papéis distintos. Um processo (ou conjunto de processos) é responsável por manter a durabilidade e integridade dos dados, este processo é chamado de *Origem*. Um outro conjunto de processos existe com o objetivo de processar operações de leitura da forma mais eficiente possível, estes processos são chamados de *Cache*. Os processos clientes são processos que executam diferentes aplicações que necessitam de um mecanismo para armazenar dados. A Figura 10 ilustra a relação e cardinalidade dos diferentes papéis.

O processo *Origem* tem como objetivo principal armazenar e garantir a durabilidade de um conjunto de dados. O projeto do protocolo de replicação não impõe limite de tamanho neste conjunto. A interface de troca de mensagens exposta pela *Origem* oferece operações para leitura e escrita de objetos. Para fins da apresentação do protocolo de replicação do *cache transparente*, assume-se que uma única instância do processo *Origem* existe, posteriormente são discutidos cenários onde a *Origem* trabalha de forma distribuída.

Os processos *Cache* são responsáveis por atender as requisições das aplicações clientes e armazenar um sub-conjunto dos dados. Em qualquer configuração, existe um número finito de instâncias de processos *Cache*. O sub-conjunto dos dados armazenados possui tamanho finito. Ao atingir o limite de objetos armazenados, o *Cache* libera espaço para

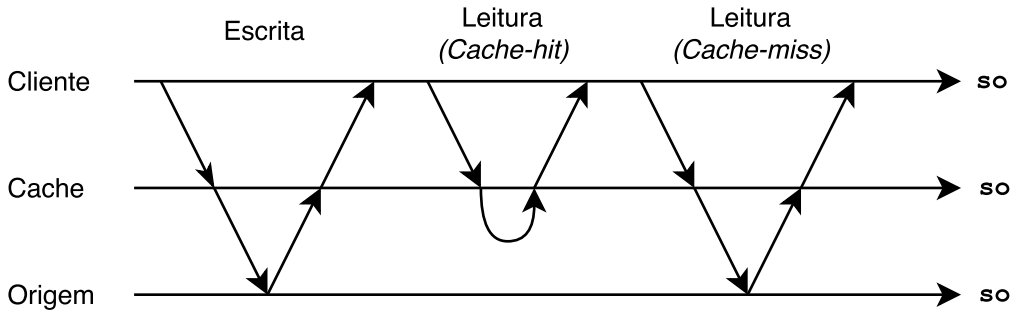


Figura 11 – Fluxos de troca de mensagens entre *Origem* e *Cache*

novas escritas excluindo outros objetos presentes no armazenamento por um mecanismo de evicção. Para efeitos do projeto do protocolo de replicação, o algoritmo de evicção utilizado não é relevante.

Os processos clientes emitem operações de leitura e escrita para os *Caches*. Para efeitos de demonstração do modelo de consistência, assume-se que uma sessão compreende operações emitidas por um único processo cliente para um único processo *Cache*. Um processo cliente pode gerenciar múltiplas sessões, porém por simplicidade, assume-se que um cliente sempre se conecta com a mesma instância de *Cache* em uma única sessão ao decorrer de sua execução.

O fluxo de mensagens trocadas em cada operação invocada pelos clientes segue um padrão semelhante ao *cache-aside*. Operações de escrita são roteadas para a *Origem* e são armazenadas no *Cache*. Leituras são servidas pelo armazenamento local do *Cache* caso o objeto requisitado seja encontrado (*cache-hit*), e são roteadas para a *Origem* caso contrário (*cache-miss*). A principal diferença é que o processo cliente não tem conhecimento do fluxo entre Origem e *Cache*, toda comunicação dos clientes ocorre como se o *Cache* contatado fosse uma cópia única dos dados. A Figura 11 ilustra os possíveis cenários de troca de mensagens entre clientes, *Cache* e *Origem*.

5.2 Projeto do Protocolo de Replicação

O projeto do protocolo de replicação é dividido entre dois papéis. Esta seção detalha o funcionamento e algoritmos desses dois tipos de papéis. Os pseudocódigos apresentados utilizam a notação $\mathbb{I}$ para denominar o conjunto de todas as instâncias de *Cache* ativas na configuração, e i (sendo $i \in \mathbb{I}$) para denominar uma instância específica de *Cache*.

5.2.1 Processos *Cache*

O processo *Cache* trabalha atendendo requisições de leitura e escrita dos processos clientes. Leituras de objetos presentes no armazenamento local (*cahce-hits*) são imediata-

mente retornadas, nas demais uma mensagem é enviada para a *Origem* solicitando o valor do objeto. As mensagens de retorno da origem sempre contém um *buffer* de modificações. Tal *buffer* é uma fila FIFO onde os elementos são operações enfileiradas pela *Origem*. As operações contidas no *buffer* são totalmente aplicadas no *Cache* antes de retornar resultados aos clientes.

O pseudocódigo do processo *Cache* é descrito no algoritmo 3. As instâncias do processo *Cache* expõem as operações de manipulação de objetos GET e PUT aos clientes do sistema. Uma instância do processo *Cache* mantém as seguintes estruturas de dados:

- O seu identificador de *Cache* $i \in \mathbb{I}$;
- Um *Mapa* C com capacidade de armazenar um número fixo predefinido de objetos. A cada inserção, o mapa retorna um objeto que foi removido para acomodar o novo.
- O tempo lógico p da última modificação aplicada em C.

A operação GET recebe como parâmetro a chave do objeto a ser buscado. Inicialmente, a chave é buscada no armazenamento local do processo, caso não exista, uma mensagem GET é enviada ao processo *Origem*. Após o recebimento do *buffer* de modificações do processo *Origem*, o mesmo é aplicado no armazenamento local (procedimento auxiliar APPLY) e o resultado da busca é retornado ao cliente.

A operação PUT envia uma mensagem de mesmo nome para o processo *Origem* e aplica o *buffer* de modificações recebido no armazenamento local. Pelo funcionamento da operação PUT do processo origem, o objeto inserido estará no próximo *buffer* de modificações retornado.

O procedimento auxiliar APPLY recebe como parâmetro um *buffer* de modificações e o aplica no armazenamento local do processo *Cache*. Enquanto existirem elementos no *buffer* recebido, o procedimento desenfileira estes elementos e verifica o tempo lógico da modificação. Caso o tempo lógico seja maior do que o tempo lógico da última operação aplicada no armazenamento local p, a modificação é aplicada. Se esta modificação resultar em uma evicção, uma mensagem EVICT é enviada à *Origem* contendo o identificador de *Cache* e a chave do objeto.

5.2.2 Processo *Origem*

O processo *Origem* atua como ponto principal de armazenamento de dados. Possíveis implementações certamente terão como objetivo proporcionar alta durabilidade aos dados armazenados pelo processo. Porém, sob o ponto de vista de um tipo de dados replicado, a principal funcionalidade do processo está na comunicação mantida com os diversos processos *Cache*.

Dados: O: processo *Origem*;
i: identificador de cache;
C: cache local;
p: última operação aplicada no cache local;

Mensagem GET*(i: Id Cache, p: Posição, k: Chave)*;
Mensagem PUT*(i: Id Cache, p: Posição, k: Chave, v: Valor)*;
Mensagem EVICT*(i: Id Cache, k: Chave)*;
Mensagem ACK*(b: Buffer)*;

Operação GET(k)**:**
    se $C[k] = \varnothing$ **então**
        Envia GET(i, p, k) **para** O;
        Aguarda ACK(b);
        APPLY(b);
    fim
    retorna $C[k]$;
fim

Operação PUT(k, v)**:**
    Envia PUT(i, p, k, v) **para** O;
    Aguarda ACK(b);
    APPLY(b);
    retorna;
fim

Procedimento APPLY(b)**:**
    enquanto existem elementos em b **faça**
        $op \leftarrow$ desenfileira de b;
        se $op.p > p$ **então**
            $k \leftarrow C[op.k] \leftarrow op.v$;
            $p \leftarrow op.p$;
            se $e \neq \varnothing$ **então**
                Envia EVICT(i, k)**para** O;
            fim
        fim
    fim
fim

Algoritmo 3: Pseudocódigo do processo *Cache*

A idéia principal do projeto do processo *Origem* está em manter registro de quais objetos estão presentes no armazenamento de cada instância de *Cache*. Isto permite que atualizações feitas por uma determinada instância de *Cache* sejam refletidas nas demais que, por ventura, também armazenem os mesmos objetos. Manter um conjunto de metadados que associe objetos a *Caches* de forma eficiente é um dos principais desafios deste projeto. Como o número de objetos armazenados é potencialmente grande e a associação dos objetos aos *Caches* é volátil (devido ao mecanismo de evicção), a sobrecarga de se manter tal registro pode inviabilizar possíveis implementações.

Para atacar este problema são utilizadas estruturas de dados probabilísticas. Estas são estruturas de dados que sacrificam respostas determinísticas em prol de uma maior eficiência da utilizaçãp de recursos. Uma das mais bem difundidas estruturas de dados deste tipo é o *Bloom Filter* (BLOOM, 1970). Esta é uma estrutura de dados que permite testar se um elemento faz parte de um conjunto utilizando um espaço muitas vezes menor do que o necessário para armazenar o conjunto. Em contrapartida, a estrutura permite um baixo índice de resultados falso-positivos, isto é, elementos não presentes no conjunto podem ser identificados como presentes. Esta desvantagem não impede o uso dos *Bloom Filters* em cenários onde o uso de estruturas como *Tabelas Hash* são impraticáveis devido ao seu uso de espaço.

Uma das principais limitações do *Bloom Filter* é a falta de suporte à remoções de elementos do conjunto. Por tal motivo o *Cuckoo Filter* (FAN et al., 2014) foi recentemente proposto com o objetivo de prover uma estrutura de dados com as mesmas funcionalidades do *Bloom Filter* ao mesmo tempo em que permite remoções dos elementos. No processo *Origem* os *Cuckoo Filters* são utilizados para manter registro de quais objetos estão presentes em cada instância de *Cache*.

Uma vez registrados os elementos presentes em cada instância de *Cache*, o processo *Origem* é capaz de manter um *buffer* de operações que devem ser aplicadas por cada *Cache*. Todas as operações executadas em objetos que estejam armazenados nos *Caches* são enfileiradas nos seus respectivos *buffers*. Cada *buffer* possui um contador associado. A cada operação inserida em um *buffer*, o respectivo contador é incrementado. Desta forma a *Origem* mantém um *Relógio Vetorial* (LAMPORT, 1978) que representa o *tempo lógico* em que cada instância de cache se encontra.

O pseudocódigo do processo origem é descrito no algoritmo 4. O processo pode receber as mensagens GET, PUT e EVICT dos processos *Cache*. As duas primeiras são mensagens básicas de busca e manipulação de objetos, a última serve como um mecanismo de atualização das evicções feitas pelos *Caches*. Uma instância do processo *Origem* mantém as seguintes estruturas de dados:

- Um *Mapa A* que atua como fonte de armazenamento principal do sistema.

- Um conjunto de *Cuckoo Filters* $F = \{f_i \mid i \in \mathbb{I}\}$, cujo objetivo é manter uma visão compacta de quais chaves estão armazenadas em quais *Caches*.
- Um conjunto de *buffers* $B = \{b_i \mid i \in \mathbb{I}\}$ que representam as modificações a serem executadas em cada *Cache*.
- Um conjunto de inteiros $P = \{p_i \in \mathbb{N} \mid i \in \mathbb{I}\}$ que mantém o maior tempo lógico inserido no *buffer* de cada *Cache*.

A mensagem GET contém o identificador do *Cache* que enviou a mensagem, tempo lógico da última operação aplicada pelo *Cache* e a chave de objeto que se deseja buscar. Ao receber a mensagem, o processo *Origem* busca o objeto requisitado pela chave no seu armazenamento local. Caso encontre, o processo adiciona a chave buscada no *Cuckoo Filter* referente ao *Cache* que enviou a mensagem, incrementa o tempo lógico do *Cache* e insere a operação no respectivo *buffer*. Por fim, o *buffer* é submetido a uma rotina de limpeza e é enviado ao *Cache* encapsulado em uma mensagem ACK.

A mensagem PUT contém os três dados da operação GET mais o valor que será associado a chave informada. Ao receber a mensagem o processo *Origem* inicialmente insere a chave no *Cuckoo Filter* do *Cache* que enviou a mensagem e atualiza o armazenamento local com a chave e valor informados. Após, para cada instância de *Cache* é verificado se a chave informada existe no respectivo *Cuckoo Filter*. Caso exista, uma modificação é inserida no respectivo *buffer*. Por fim, o *buffer* do cache que invocou a operação é submetido à rotina de limpeza e retornado ao *Cache* que enviou a mensagem.

A mensagem EVICT contém o identificador de *Cache* e uma chave de objeto. Ao receber a mensagem, o processo *Origem* remove a chave do *Cuckoo Filter* referente ao *Cache* que enviou a mensagem. O objetivo desta operação é fazer com que futuras atualizações no objeto que sofreu evicção deixem de ser inseridas no *buffer* do *Cache*.

Além das operações expostas, o algoritmo define o procedimento auxiliar CLEAN-BUF. Este procedimento é uma rotina que limpa operações já aplicadas por um determinado *Cache* do seu respectivo *buffer*. O procedimento atua recebendo como parâmetros um *buffer* e um tempo lógico de operação. A rotina desenfileira elementos do *buffer* até que encontre um que tenha tempo de operação maior do que o recebido como parâmetro.

5.3 Exatidão e Modelo de Consistência

Nesta seção a exatidão do *Cache Transparente* é abordada. As condições de retorno e término dos algoritmos são demonstradas bem como o modelo de consistência provido pela solução.

Dados: A: armazenamento local;
F: conjunto de *Cuckoo Filters*. Um por instância de *Cache*;
B: conjunto de *buffers* de modificações. Um por instância de *Cache*;
P: conjunto de inteiros. Um por instância de *Cache*;

Mensagem GET*(i: Id Cache, p: Posição, k: Chave)*;
Mensagem PUT*(i: Id Cache, p: Posição, k: Chave, v: Valor)*;
Mensagem EVICT*(i: Id Cache, k: Chave)*;
Mensagem ACK*(b: Buffer)*

Recebe GET(i, p, k)**:**
  se $A[k] \neq \varnothing$ **então**
    adiciona k em F_i;
    P_i++;
    enfileira $\langle k, A[k], P_i \rangle$ em B_i;
  CLEANBUF(B_i, p);
  Envia ACK(B_i) **para** i;
fim

Recebe PUT(i, p, k, v)**:**
  adiciona k em F_i;
  $A[k] \leftarrow v$;
  para cada $j \in \mathbb{I}$ **faça**
    se k existe em F_j **então**
      P_i++;
      enfileira $\langle k, v, P_j \rangle$ em B_j;
    fim
  fim
  CLEANBUF(B_i, p);
  Envia ACK(B_i) **para** i;
fim

Recebe EVICT(i, k)**:**
  remove k de F_i;
fim

Procedimento CLEANBUF(b, p)**:**
  enquanto elemento no topo de b tiver posição $\leq p$ **faça**
    desenfileira elemento de b;
  fim
fim

Algoritmo 4: Pseudocódigo do processo *Origem*

As condições de retorno e término são verificadas observando-se as duas operações expostas aos clientes GET e PUT. A operação PUT não possui retorno e o loop nos caches na linha 16 do algoritmo 4 termina, pois o número de caches é finito. O procedimento auxiliar CLEANBUF termina pois o *loop* nos elementos do *buffer* na linha 29 do algoritmo 4 não ultrapassa o tamanho do *buffer* de um determinado *Cache*, que é finito. O procedimento auxiliar APPLY é garantido de terminar pelo mesmo motivo. A operação GET sempre retorna um dado do tipo *Valor* (inclusive $\varnothing$) e não possui *loops* além dos existentes nos procedimentos CLEANBUF e APPLY.

Para determinar o modelo de consistência garantido, primeiramente definem-se as relações de arbitragem ar e visibilidade vis conforme apresentadas no Capítulo 3. Estas relações de ordenação servirão de base para determinar as garantias de ordenação e, consequentemente, o modelo de consistência provido pela solução.

Para definir a arbitragem, consideram-se as operações que alteram o estado do sistema. As operações de escrita obviamente alteram o estado pois modificam os *buffers* e armazenamentos dos dois tipos de processos. Já as operações de leitura podem ser diferenciadas entre *Cache-hits*, ou seja, leituras de objetos que estão presentes no armazenamento local do *Cache*, ou *Cache-misses*, ou seja, leituras que devem ser direcionadas para a *Origem*. Os *Cache-hits* não alteram o estado do sistema pois somente lêem um objeto do cache local, já os *Cache-misses* alteram o estado pois inserem novos elementos nos *buffers*. Como todas as operações que alteram o estado do sistema passam pelo processo *Origem* e tem seus retornos determinados pelo mesmo, pode-se deduzir que a ordem de execução da *Origem* determina a arbitragem do sistema. Ou formalmente:

$$\mathsf{ar} = \mathsf{eo}|_{orig}$$

A relação de visibilidade das operações é facilmente compreendida observando-se o histórico ilustrado pela Figura 12. Pode-se observar que uma determinada instância de *Cache* tem visibilidade sobre todas as operações que ocorreram até a última comunicação com a *Origem*. A partir do ponto na execução de um cache onde qualquer comunicação com a *Origem* seja feita, todos os elementos presentes no armazenamento do *Cache* são atualizados. Ou formalmente:

$$a \xrightarrow{\mathsf{vis}} b \overset{\text{def}}{=} a \xrightarrow{\mathsf{ar}} \mathsf{lc}(b)$$

Onde $\mathsf{lc}(op)$: *last_op* (*last communication*) é uma função que dada uma operação *op*, retorna a última operação que se comunica com a *Origem* que precede *op*. Caso *op* seja uma operação que se comunique com a *Origem*, $last\_op = op$.

Dadas as definições de visibilidade e arbitragem pode-se inferir que visibilidade implica em arbitragem, ou seja, se uma operação a é visível para uma operação b ($a \xrightarrow{\mathsf{vis}} b$), a é resolvido antes que b ($a \xrightarrow{\mathsf{ar}} b$), isto se dá pelo fato de que se as duas operações passam

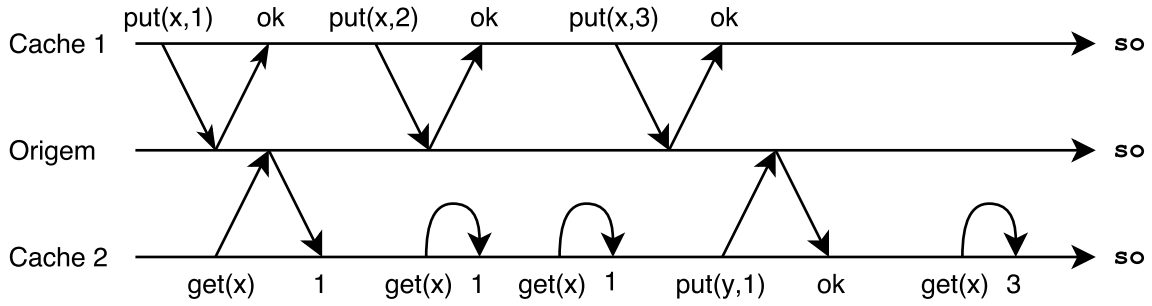
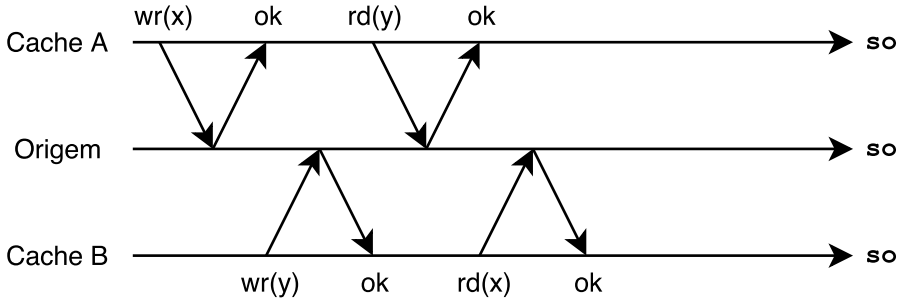


Figura 12 – Execução de uma configuração com dois *Caches*



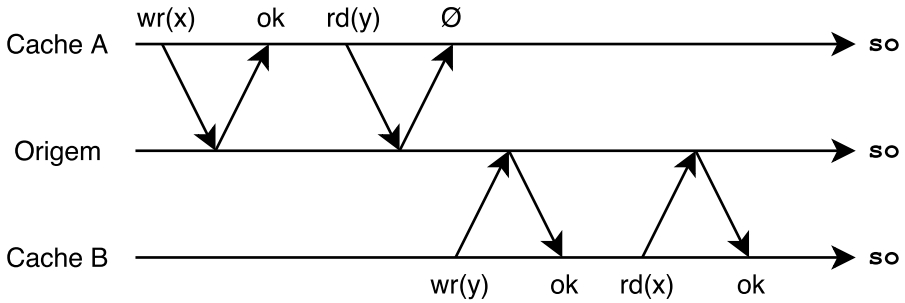


Figura 13 – Execuções do teste de Dekker no *Cache Transparente*

pela *Origem*, elas serão resolvidas de acordo com a ordem de execução da *Origem*. Tal condição é suficiente para demonstrar que a garantia OrdemÚnicaé obedecida.

Outra forma de observar a garantia OrdemÚnica é através do teste de Dekker. A Figura 13 demonstra duas execuções do teste de Dekker em uma instância do *Cache Transparente*. As duas execuções podem ser generalizadas para todas as demais possíveis invertendo-se as ordens das operações. Pelas execuções pode-se observar que todas as operações obrigatoriamente passam pela *Origem* e são ordenadas pela mesma. Esta condição demonstra ser impossível que as duas leituras retornem ∅.

Operações executadas numa mesma sessão, logo no mesmo *Cache*, tem relação de visibilidade equivalente à relação ordem de sessão so. Isto significa que uma leitura realizada em um determinado *Cache* sempre visualizará uma escrita realizada anteriormente no

mesmo *Cache*. Esta condição é suficiente para demonstrar que o protocolo obedece a garantia de ordenação LerMinhasEscritas.

Com as duas garantias de ordenação apresentadas pode-se definir o modelo de consistência garantido pelo protocolo. De acordo com a Tabela 4, OrdemÚnica e LerMinhasEscritas são suficientes para garantir o modelo de consistência ConsistênciaSequencial.

5.4 Utilização de Rede

No *Cache Transparente* as escritas são sempre roteadas do *Cache* para a *Origem*. Esta interação resulta em 4 mensagens trocadas por operação de escrita. Já as leituras tem interações diferentes entre *cache-hits* e *cache-misses*. Em um *cache-hit* apenas uma comunicação entre cliente e *Cache* basta, o que resulta em 2 mensagens trocadas. O *cache-miss* tem fluxo semelhante ao da escrita e requer 4 mensagens para ser completado.

Utilizando as mesmas características de cargas de trabalho descritas no Capítulo 4, ou seja, proporções 1:1 e 30:1 entre leituras e escritas e leituras sendo 81% *cache-hits*, as quantidades médias de trocas de mensagens do *Cache Transparente* de acordo com a Equação 4.1 são:

- Proporção 1:1 entre leituras e escritas: 3,19
- Proporção 30:1 entre leituras e escritas: 2,43

O *Cache Transparente* mostra alta eficiência no uso de rede se comparado à replicação assíncrona com cópia primária e *cache-aside*. Um comparação entre as três soluções que aborda o uso de rede é feita na Seção 5.7.

5.5 Leituras Obsoletas

A relação de visibilidade das operações mostra que uma instância de *Cache* que não se comunica frequentemente com a *Origem* pode observar um alto número de *leituras obsoletas* (do inglês, *stale reads*). A leitura obsoleta é uma anomalia possível em sistemas que não atendem a garantia de ordenação TempoReal. Uma leitura obsoleta ocorre quando um cliente lê uma versão de um objeto que, na ordem de tempo real, já foi modificada por outro cliente. A ausência completa de leituras obsoletas normalmente é obtida por sistemas que garantem o modelo de consistência Linearização. Porém, projetos de sistemas que não garantem este modelo podem adotar mecanismos que minimizem a ocorrência deste tipo de anomalia.

A falta de comunicação frequente entre *Origem* e *Cache*, além do aumento na possibilidade de leituras obsoletas, também acarreta no aumento do número de objetos nos *buffers*. O que gera um aumento na utilização de recursos pelo sistema. Pela forma como o protocolo de replicação do *Cache Transparente* funciona, apenas uma comunicação entre *Cache* e *Origem* é suficiente para atualizar todos os possíveis objetos obsoletos presentes no armazenamento local do *Cache*.

Para diminuir a possibilidade de leituras obsoletas, duas abordagens podem ser tomadas. A primeira envolve modificar a função de recebimento de mensagens PUT no processo *Origem* para que após a inclusão da modificação em cada *buffer*, a mesma envie o *buffer* para o *Cache* em questão. Com este método o tempo máximo em que um objeto pode permanecer obsoleto em um cache é igual ao tempo de troca de mensagens entre *Origem* e *Cache*. Este é o mesmo tempo em que um objeto pode permanecer obsoleto em sistemas cópia-primária com replicação assíncrona.

Outra abordagem é executar periodicamente uma rotina que envia os *buffers* para os *Caches*. Para que o método seja eficiente, o período de tempo em que a rotina é executada deve ser menor que o tempo médio entre diferentes comunicações entre *Origem* e *Cache*. Tal tempo é determinado de acordo com a carga de trabalho que se deseja executar no sistema, e por tal motivo deve ser configurado a cada implementação. Neste método o tempo máximo em que um objeto pode permanecer obsoleto em um *Cache* é o período configurado acrescido do tempo de comunicação entre *Origem* e *Cache*.

Ambas as abordagens acarretam num aumento de mensagens trocadas entre os processos em prol de uma característica que pode não ser relevante. Por exemplo, aplicações nas quais objetos raramente são modificados não sofrem de leituras obsoletas com frequência. Por tal motivo, a implementação destas abordagens deve ser considerada de acordo com a carga de trabalho que se planeja executar.

Considerando as soluções apresentadas no Capítulo 4, as leituras obsoletas são particularmente problemáticas para a solução *cache-aside*. A partir do momento que um objeto é inserido pela aplicação no cache, ele permanece no cache até que uma evicção ocorra. Dependendo do algoritmo de evicção utilizado o objeto pode permanecer indefinidamente no cache, ignorando todas as possíveis atualizações que possam vir a serem feitas. Por tal motivo, implementações da solução recorrem ao limite máximo de tempo que um objeto pode permanecer armazenado no cache. O *Cache Transparente* não sofre de tal problema, apenas uma operação que se comunique com a *Origem* basta para que todos os objetos armazenados no *Cache* tenham suas versões atualizadas.

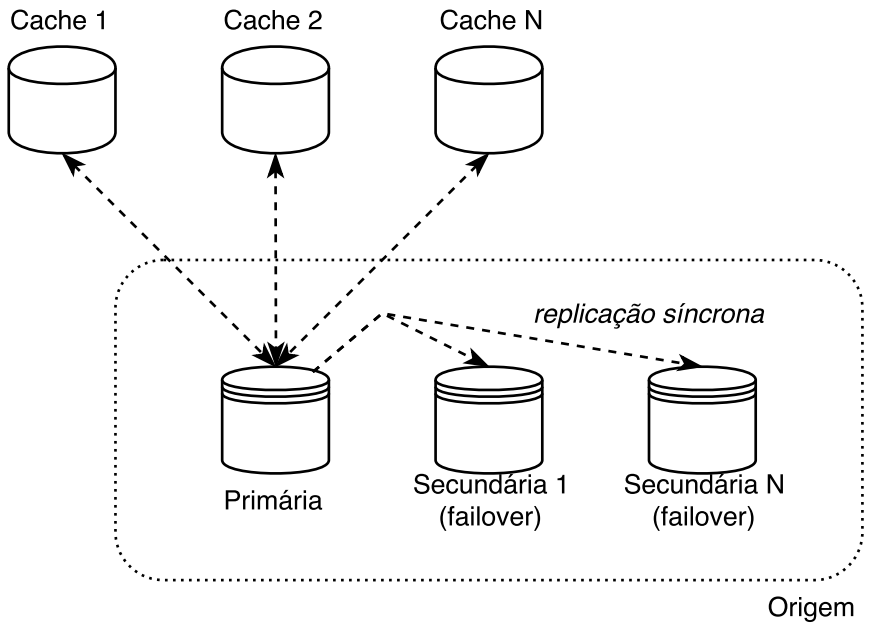


Figura 14 – Processo *Origem* replicado com cópia primária

5.6 Configurações com a *Origem* Distribuída

O processo *Origem* trabalhando de forma centralizada e com instância única certamente tem suas desvantagens. A principal delas é a falta de tolerância a falhas. Uma eventual falha no processo *Origem* significa que os *Caches* somente são capazes de responder a *cache-hits*. Esta situação pode ser considerada como uma indisponibilidade parcial do sistema e é dependente da lógica das aplicações que se utilizam do mesmo.

Para tratar estes problemas, técnicas de replicação de dados podem ser utilizadas. Porém, para manter o modelo de consistência, a *Origem* distribuída deve ser capaz de garantir a linearização como modelo de consistência. Esta condição restringe as opções de estratégia de replicação para aquelas com replicação síncrona.

Utilizando replicação síncrona com cópia primária, as réplicas secundárias devem trabalhar no modo *failover*. Todos os *Caches* devem ser conectados na réplica primária e migrar para qualquer secundária que assuma o papel de primária em caso de falha. Esta configuração permite que eventuais falhas na *Origem* sejam mitigadas, porém não balanceia a carga entre as réplicas da *Origem*. A Figura 14 ilustra a relação entre os componentes desta configuração.

Na replicação síncrona com atualização em todos os lugares, todas as réplicas da *Origem* podem ter *Caches* conectados. Em uma eventual falha de uma das réplicas, os caches conectados a esta réplica passariam a ser capazes de processar somente leituras que acarretem num *cache-hit*. Em contrapartida, a configuração permite que a carga da *Origem* seja dividida entre mais processos. A Figura 15 ilustra a relação entre os componentes

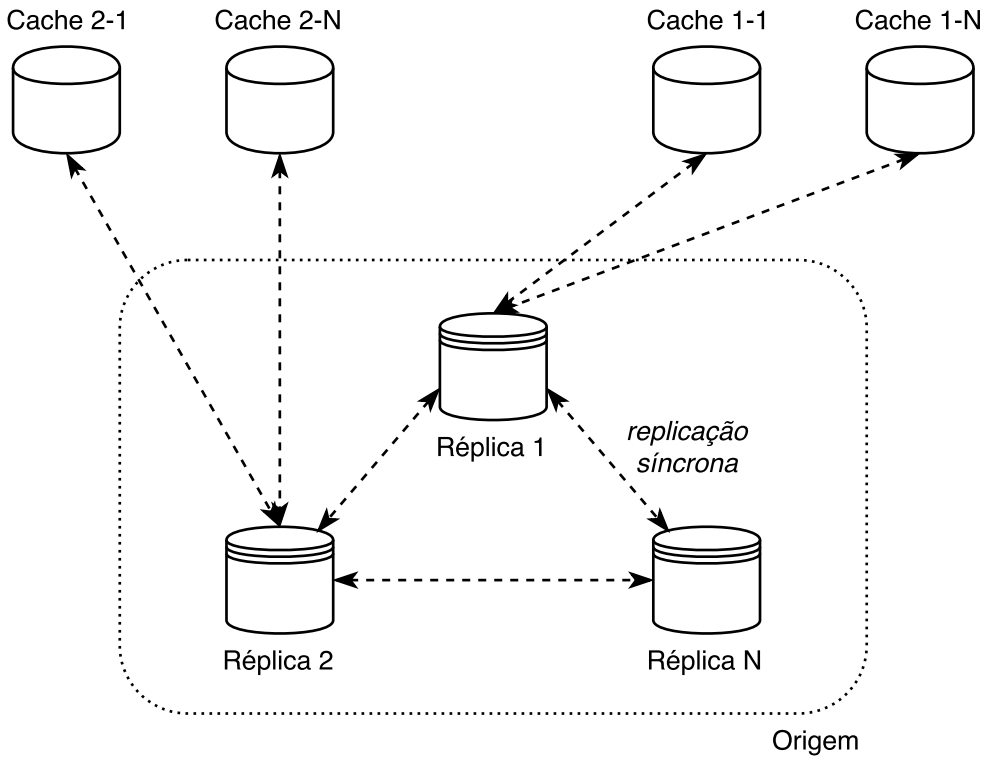


Figura 15 – Processo *Origem* replicado com atualização em todos os lugares

desta configuração.

As configurações com o processo *Origem* replicado permitem uma espécie de estratégia híbrida entre replicação síncrona e assíncrona. Ao mesmo tempo em que os benefícios da replicação síncrona como consistência forte e tolerância a falhas são obtidos, a solução pode escalar entre WANs. Com estas configurações, um conjunto de réplicas da *Origem* pode ser disponibilizado em um centro de dados enquanto os caches podem ser espalhados entre outros centros de dados conectados por WANs.

5.7 Comparações entre as Soluções

O *Cache Transparente* se destina a resolver o mesmo problema atacado pelas soluções descritas no Capítulo 4, a replicação assíncrona com cópia primária e o *cache-aside*. O *Cache Transparente* se destaca destas soluções principalmente pelo modelo de consistência provido e pela utilização de rede. Esta seção faz uma comparação direta entre as três soluções.

A Tabela 7 resume as garantias de ordenação providas pelas três soluções. A tabela mostra que as três soluções atendem a garantia VisibilidadeEventual, que define o conceito de eventualidade. Já as garantias de sessão LerMinhasEscritas e LeiturasMonotônicas são atendidas apenas pelo *cache-aside* e *Cache Transparente*.

	RACP	*Cache-aside*	*Cache Transparente*
VisibilidadeEventual	✓	✓	✓
LerMinhasEscritas		✓	✓
LeiturasMonotônicas		✓	✓
VisibilidadeCausal			✓
ArbitragemCausal	✓	✓	✓
OrdemÚnica			✓
TempoReal			

Tabela 7 – Garantias de ordenação providas pelas soluções

Isto se deve pelo fato dos clientes se comunicarem com as mesmas instâncias de processos durante uma determinada sessão. A garantia VisibilidadeCausal é atendida apenas pelo *Cache Transparente* bem como a garantia OrdemÚnica. Esta última em conjunto com a garantia LerMinhasEscritas faz com que a solução seja capaz de prover o modelo de consistência ConsistênciaSequencial.

Os benefícios providos pelas garantias de sessão são fáceis de serem observados. Basta considerar uma aplicação social onde um usuário escreve uma mensagem em um mural. Em seguida o mesmo usuário lê as mensagens deste mesmo mural. Sem as garantias de sessão o usuário pode não visualizar a mensagem que acabou de escrever. Esta anomalia e possível na replicação assíncrona com cópia primária.

Um exemplo clássico de anomalia relacionado a relações de causa pode ser descrito com a seguinte sequência de eventos (LLOYD et al., 2013; BURCKHARDT, 2014):

- Alice publica a mensagem $a1$ em seu mural: *"Estou no hospital."*
- Instantes depois Alice publica a mensagem $a2$ em seu mural: *"Já está tudo bem."*
- Bob lê as mensagens $a1$ e $a2$ e publica a mensagem $b1$ no mural de Alice: *"Que bom!"*

Em um sistema que não ofereça garantias de causa como a VisibilidadeCausal, um terceiro usuário Charlie pode ler o mural de Alice e receber como retorno somente as mensagens $a1$ e $b1$. Isto passaria a falsa impressão de que Bob está feliz em saber que Alice está no hospital. Tanto na replicação assíncrona com cópia primária quanto no *cache-aside* esta anomalia é possível.

As anomalias descritas são duas dentre muitas outras possíveis em sistemas que não oferecem modelos mais restritos de consistência. O *Cache Transparente*, por prover o modelo de consistência ConsistênciaSequencial, é livre de todas as anomalias causadas pela falta de garantias de sessão e causa. Isto simplifica consideravelmente o

	RACP	*Cache-aside*	*Cache Transparente*
1:1	$4 + N/2$	3,38	3,19
30:1	$4 + N/31$	2,8	2,43

Tabela 8 – Quantidades médias de mensagens trocadas entre as três soluções em cargas de trabalho com proporções de 1:1 e 30:1 entre leituras e escritas

desenvolvimento de aplicações clientes uma vez que os desenvolvedores não precisam inserir lógicas de tratamento de anomalias em seus programas.

Com relação a utilização de rede, as três soluções apresentam resultados diferentes. A Tabela 8 resume as quantidades médias de mensagens trocadas por operação em cada uma das três soluções. A tabela mostra as quantidades considerando cargas de trabalho com proporções de 1:1 e 30:1 entre leituras e escritas. Nas soluções *cache-aside* e *Cache Transparente* as leituras possuem 81% de *cache-hits* (ver Subseção 4.3.2). É importante notar que *cache-hits* e *cache-misses* não se aplicam à replicação assíncrona com cópia primária uma vez que a solução emprega replicação total dos dados.

Os efeitos em escala das médias de mensagens trocadas é ilustrado pela Figura 16. Na figura, as três soluções tem seus números totais de mensagens trocadas plotados em dois gráficos, um para cada tipo de carga de trabalho. Nos gráficos, o eixo y representa a quantidade de mensagens trocadas, os dois gráficos compartilham da mesma escala neste eixo. Já o eixo x representa a quantidade total de operações executadas. Na solução replicação assíncrona com cópia primária, o número de réplicas secundárias utilizadas N é 3 [1]. Nos gráficos pode-se observar a progressão linear na diferença de mensagens trocadas entre as três soluções. O *Cache Transparente* apresenta o melhor resultado dentre as três soluções analisadas.

Outra característica importante a se observar é a possibilidade do uso de armazenamento volátil para os processos destinados a tratar das operações de leitura. O uso de armazenamentos voláteis como a memória RAM pode reduzir significativamente a busca de dados, beneficiando assim as operações de leitura. Das três soluções apenas o *cache-aside* e o *Cache Transparente* suportam esta funcionalidade. Isto é possível pelo protocolos das duas soluções estar preparado para lidar com as operações que resultam em *cache-miss*.

De forma geral o *Cache Transparente* apresenta resultados melhores quando comparado com a replicação assíncrona com cópia primária e o *cache-aside*. O *Cache Transparente* é consideravelmente mais consistente e mais eficiente na utilização de rede do que as outras duas soluções. A replicação assíncrona com cópia primária apesar de ter mostrado os piores resultados, é uma solução mais versátil. Esta pode ser utilizada como solução para

[1] A quantidade de réplicas secundárias é dependente da implementação, 3 é um valor padrão comum em serviços que usam a solução em provedores cloud como o RDS da AWS (https://aws.amazon.com/documentation/rds/)

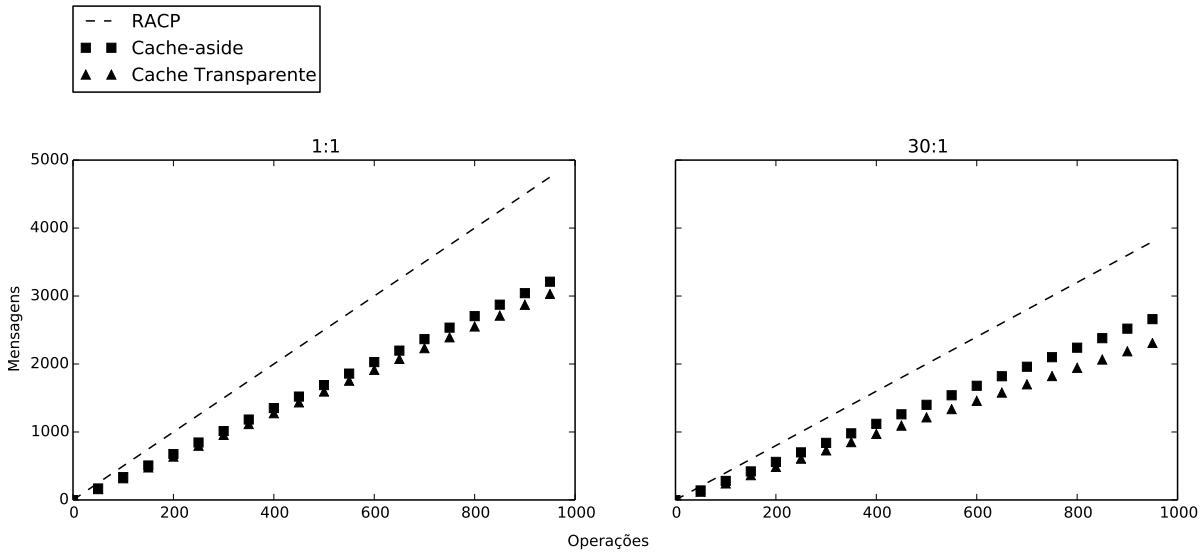


Figura 16 – Mensagens trocadas por operação entre as três soluções com cargas de trabalho 1:1 e 30:1

tolerância a falhas enquanto as outras duas soluções se destinam somente a processar cargas de trabalho intensivas em leituras.

6 Conclusão

Este capítulo apresenta observações finais sobre o trabalho. Primeiramente, uma seleção da literatura relacionada é apresentada e brevemente descrita. Estes trabalhos serviram como base ou inspiração para este trabalho. Após, são discutidos possíveis trabalhos futuros e temas que podem vir a complementar as contribuições deste trabalho. Por fim, as considerações finais concluem o trabalho.

6.1 Trabalhos Relacionados

A seguir são descritos alguns dos trabalhos que utilizam técnicas semelhantes as utilizadas no *Cache Transparente*.

6.1.1 Armazenamento Intermediário Entre Clientes e Sistemas de Armazenamento de Dados

Bailis et. al. (2013) emprega técnicas que serviram como inspiração para este trabalho. O trabalho aborda o uso de uma camada de armazenamento intermediária entre um sistema de armazenamento de dados e seus clientes. Os objetivos são diferentes dos deste trabalho, porém algumas das técnicas são semelhantes. Particularmente, o fluxo de troca de mensagens segue um padrão muito semelhante ao do *Cache Transparente*.

Os autores propõem uma camada intermediária entre uma base distribuída *Chave-Valor* eventualmente consistente e seus clientes com o objetivo de "aumentar" o modelo de consistência do sistema para o modelo causal. A camada funciona como um conjunto de processos que existem para cada uma das réplicas da base *Chave-Valor* eventualmente consistente. Estes processos se tornam os responsáveis por receber e processar as requisições dos clientes. Estes processos adicionam como parâmetros das operações um conjunto extra de objetos que representam as "operações-causa" da operação em questão. O objetivo é prover um modelo de "consistência causal explícita" onde os clientes explicitamente informam ao sistema quais são as causas de cada operação. Conforme mencionado no Capítulo 3 este tipo de consistência ainda não foi devidamente formalizado.

Os processos da camada intermediária armazenam os dados do sistema como uma espécie de cache. Ao contrário dos sistemas apresentados neste trabalho, não existe evicção nestes processos. Um algoritmo garante que em qualquer momento da execução o conjunto de dados presentes no processo intermediário não viola o modelo de consistência causal. O algoritmo trabalha constantemente analisando os objetos requisitados pelos clientes e verificando as dependências na base eventualmente consistente. A principal desvantagem

gerada pela abordagem é a sobrecarga causada no tráfego e armazenamento de dados gerada pelas operações-causa.

6.1.2 Replicação Dinâmica Ciente de Uso

Matri et. al. (2016) propõe um método de replicação que tenta posicionar objetos nas réplicas que são mais prováveis de receber requisições de operações que leiam ou manipulem tais objetos. Para isto, uma estratégia de associação de clientes a réplicas é utilizada juntamente com um algoritmo que encontra elementos com alta frequência em um fluxo de dados. O comportamento obtido é semelhante ao do *cache-aside* e *Cache Transparente*, onde os caches armazenam os objetos com maior probabilidade de serem acessados.

O método requer que clientes mantenham uma lista de todas as réplicas participantes do sistema ordenadas por um índice de preferência, onde a preferência é medida por atributos relacionados ao desempenho como latência e distância. Cada operação é enviada à réplica com melhor índice de preferência. Cada réplica mantém uma lista dos objetos mais "populares" usando o algoritmo *Space-saving* (METWALLY; AGRAWAL; ABBADI, 2005). A replicação é feita de forma assíncrona com um protocolo *Gossip* (DEMERS et al., 1987).

Leituras são feitas tentando localizar a réplica mais próxima contendo o objeto requisitado. Para isso, as réplicas enviam um *Bloom Filter* para os clientes com uma relação objeto-réplica. Os clientes usam esta estrutura para identificar a réplica mais próxima contendo o objeto requisitado e realizar a leitura.

O método traz bons resultados nas métricas de uso de espaço de armazenamento, objetos tem grandes chances que serem replicados somente nas réplicas onde serão realmente acessados. Outro bom resultado apresentado pelo trabalho é a latência média de invocações pelos clientes uma vez que esses são mais propensos a acessarem réplicas próximas. Um ponto negativo é a sobrecarga causada pelos *bloom filter* que devem ser armazenados em cada cliente. O autor não menciona uma métrica relacionando o tamanho do *bloom filter* com o tamanho da base, nem análises da sobrecarga gerada nos clientes.

6.1.3 Tipos de Dados Replicados

Trabalhos recentes tem focado esforços em definir tipos de dados distribuídos que garantam modelos de consistência relaxados por padrão. Os *Conflict-Free Replicated Data Types* (CRDT) (Tipos de Dados Replicados e Livres de Conflitos) (SHAPIRO et al., 2011b) são tipos de dados projetados com o propósito específico de prover consistência eventual. A principal vantagem provida pelos CRDTs é a garantia de replicação de estado de uma forma onde conflitos nunca ocorrem. Os CRDTs assumem um modelo de sistema onde

processos se comunicam sobre uma rede assíncrona e não confiável, isto é, a rede pode ser particionada e a entrega de mensagens pode arbitrariamente falhar. Assim como as soluções abordadas neste trabalho (replicação assíncrona com cópia primária, *cache-aside* e *Cache Transparente*), os CRDTs podem servir como base para sistemas de armazenamento de dados voltados para cargas de trabalho intensivas em leituras.

Em uma implementação de CRDTs, vários processos com as mesmas operações trabalham de forma totalmente conectada, ou seja, cada processo é capaz de alcançar todos os demais. Cada um destes processos pode receber requisições de leitura ou escrita de um ou mais processos clientes.

Existem dois tipos de CRDTs: *State-Based Convergent Replicated Data Type* (CvRDT) (Tipo de Dado Convergente Replicado Baseado em Estado) ou *Operation-Based Commutative Replicated Data Type* (CmRDT) (Tipo de Dado Comutativo Replicado Baseado em Operação).

CvRDTs são baseados em replicação de estado. Os processos de um sistema CvRDT trabalham enviando seu estado uns para os outros. Uma função que "fundi" diferentes estados é implementada por cada processo.

CmRDTs são baseados em operações comutativas, ou seja operações onde a mudança de ordem dos operandos não altera o resultado. Obviamente, uma operação comutativa deve ser projetada para manipular o estado de um CmRDT, o que limita o seu uso.

As propriedades dos dois tipos de CRDTs especificam de forma abstrata como uma implementação deve se comportar. Os autores apresentam três implementações de CRDTs como resultados do trabalho: Contador, Log, e Grafo Direcionado. Trabalhos recentes vem expandindo as implementações disponíveis de CRDTs, dentre as contribuições pode-se citar implementações de *Sets* e *Maps* (SHAPIRO et al., 2011a) e tipos para edição de texto colaborativa (PREGUICA et al., 2009).

6.2 Trabalhos Futuros

Como muitos dos trabalhos relacionados mencionados, este trabalho aborda projetos de sistemas *Chave-Valor*. Este é um modelo simples e de propósito geral que permite que o foco seja direcionado para o protocolo de replicação dos objetos. O modelo *Chave-Valor* em compensação pode ser utilizado como base para o desenvolvimento de modelos mais complexos. Como trabalho futuro, o protocolo apresentado neste trabalho pode ser extendido para suportar modelos de dados mais complexos como tabelas ou documentos. Uma extensão para o modelo *SQL-Like* é o primeiro passo para uma futura implementação do modelo de dados relacional.

Além de diferentes modelos de dados, modificar o protocolo apresentado neste

trabalho para suportar transações é outro possível trabalho futuro. Suporte a transações é uma funcionalidade de extrema importância em sistemas de armazenamento de dados, especialmente os que se destinam a processar operações *online*. Modificar o protocolo para suportar transações apresenta grandes desafios, especialmente pelo fato de que uma transação pode envolver operações que podem ser executadas no *Cache*, *Origem* ou ambos.

Uma implementação de referência do protocolo apresentado neste trabalho utilizando-se de sistemas de código aberto como base é outro possível trabalho futuro. RDBMSs de código aberto como PostgreSQL [1] e MySQL [2] podem ser utilizados como *Origem*. Para tal os mesmos devem ser modificados para suportar a troca de mensagens com os *Caches*. Já as funcionalidades dos *Caches* podem ser implementadas em sistemas de armazenamento em memória como o Redis [3] ou Memcached [4]. A principal dificuldade nesta implementação está em modificar os projetos de código aberto. Estes são projetos com vários anos de desenvolvimento pela comunidade e podem ter bases de código bastante complexas. O principal motivador de tal implementação é criar um sistema de altos níveis de desempenho e segurança próprio para uso em produção.

6.3 Considerações Finais

Soluções adotadas pela indústria para solucionar problemas de escalabilidade em cargas de trabalho intensivas em leitura sofrem de problemas. O principal deles é a consistência provida por tais soluções. Soluções como a replicação assíncrona com cópia primária e *Cache-aside* utilizam-se de sistemas de armazenamento com consistência forte como base mas acabam por prover apenas consistência eventual aos seus clientes. O preço pago pela escalabilidade vai ainda além. Tráfego de rede e uso de espaço em disco são recursos que podem ser mais bem utilizados. Algumas soluções são invasivas ao ponto de requererem que as aplicações sejam responsáveis por gerenciar a replicação dos dados.

Este trabalho apresentou o *Cache Transparente*, um tipo de dados replicado destinado a solucionar os mesmos problemas atacados pelas soluções acima mencionadas ao mesmo tempo em que evita muitas das desvantagens impostas pelas mesmas. O projeto do *Cache Transparente* faz uso de estruturas de dados probabilísticas recentes e estratégias de envio de *logs* de modificações para os diversos processos que armazenam os dados. O protocolo de replicação do *Cache Transparente* é compreendido de dois papéis: a *Origem*, responsável por manter a durabilidade dos dados; e os *Caches*, responsáveis por responder as solicitações dos clientes e armazenar os objetos acessados recentemente.

O modelo de consistência provido pelo *Cache Transparente* é um avanço se compa-

[1] https://www.postgresql.org/
[2] https://www.mysql.com/
[3] https://redis.io/
[4] https://memcached.org/

rado às soluções atuais. Enquanto as demais soluções oferecem consistência eventual, o *Cache Transparente* oferece consistência sequencial. O uso de recursos de rede também é reduzido e permite configurações conectadas por redes de longa distância. Por fim, toda a complexidade de replicação é abstraída dos clientes. Um processo cliente do *Cache Transparente* é incapaz de distinguir uma instância de *Cache* de um sistema *Chave-Valor* de instância única.

Referências

AHAMAD, M. et al. Causal memory: definitions, implementation, and programming. *Distributed Computing*, v. 9, n. 1, p. 37–49, 1995. ISSN 1432-0452. Disponível em: <http://dx.doi.org/10.1007/BF01784241>. Citado na página 26.

ALMEIDA, S.; LEITÃO, J.; RODRIGUES, L. Chainreaction: A causal+ consistent datastore based on chain replication. In: *Proceedings of the 8th ACM European Conference on Computer Systems*. New York, NY, USA: ACM, 2013. (EuroSys '13), p. 85–98. ISBN 978-1-4503-1994-2. Disponível em: <http://doi.acm.org/10.1145/2465351.2465361>. Citado na página 14.

ATIKOGLU, B. et al. Workload analysis of a large-scale key-value store. In: *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems*. New York, NY, USA: ACM, 2012. (SIGMETRICS '12), p. 53–64. ISBN 978-1-4503-1097-0. Disponível em: <http://doi.acm.org/10.1145/2254756.2254766>. Citado 3 vezes nas páginas 13, 37 e 43.

BAILIS, P. et al. The potential dangers of causal consistency and an explicit solution. In: *Proceedings of the Third ACM Symposium on Cloud Computing*. New York, NY, USA: ACM, 2012. (SoCC '12), p. 22:1–22:7. ISBN 978-1-4503-1761-0. Disponível em: <http://doi.acm.org/10.1145/2391229.2391251>. Citado na página 27.

BAILIS, P.; GHODSI, A. Eventual consistency today: Limitations, extensions, and beyond. *Queue*, ACM, New York, NY, USA, v. 11, n. 3, p. 20:20–20:32, mar. 2013. ISSN 1542-7730. Disponível em: <http://doi.acm.org/10.1145/2460276.2462076>. Citado 2 vezes nas páginas 14 e 25.

BAILIS, P. et al. Bolt-on causal consistency. In: *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. New York, NY, USA: ACM, 2013. (SIGMOD '13), p. 761–772. ISBN 978-1-4503-2037-5. Disponível em: <http://doi.acm.org/10.1145/2463676.2465279>. Citado 3 vezes nas páginas 27, 32 e 63.

BLOOM, B. H. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, ACM, New York, NY, USA, v. 13, n. 7, p. 422–426, jul. 1970. ISSN 0001-0782. Disponível em: <http://doi.acm.org/10.1145/362686.362692>. Citado na página 51.

BRESLAU, L. et al. Web caching and zipf-like distributions: evidence and implications. In: *INFOCOM '99. Eighteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*. [s.n.], 1999. v. 1, p. 126–134 vol.1. ISSN 0743-166X. Disponível em: <http://ieeexplore.ieee.org/abstract/document/749260/>. Citado na página 12.

BREWER, E. A. Towards robust distributed systems (abstract). In: *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing*. New York, NY, USA: ACM, 2000. (PODC '00), p. 7–. ISBN 1-58113-183-6. Disponível em: <http://doi.acm.org/10.1145/343477.343502>. Citado na página 14.

BRUNER, J. *Tweets loud and quiet*. 2013. Disponível em: <https://www.oreilly.com/ideas/tweets-loud-and-quiet>. Citado na página 12.

BURCKHARDT, S. *Principles of Eventual Consistency*. now publishers, 2014. 1-150 p. Disponível em: <https://www.microsoft.com/en-us/research/publication/principles-of-eventual-consistency/>. Citado 11 vezes nas páginas 7, 14, 21, 22, 23, 25, 27, 28, 29, 32 e 60.

BURCKHARDT, S. et al. Replicated data types: Specification, verification, optimality. In: *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. New York, NY, USA: ACM, 2014. (POPL '14), p. 271–284. ISBN 978-1-4503-2544-8. Disponível em: <http://doi.acm.org/10.1145/2535838.2535848>. Citado na página 32.

BURCKHARDT, S. et al. Global Sequence Protocol: A Robust Abstraction for Replicated Shared State. In: BOYLAND, J. T. (Ed.). *29th European Conference on Object-Oriented Programming (ECOOP 2015)*. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2015. (Leibniz International Proceedings in Informatics (LIPIcs), v. 37), p. 568–590. ISBN 978-3-939897-86-6. ISSN 1868-8969. Disponível em: <http://drops.dagstuhl.de/opus/volltexte/2015/5238>. Citado na página 32.

CECCHET, E.; CANDEA, G.; AILAMAKI, A. Middleware-based database replication: The gaps between theory and practice. In: *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*. New York, NY, USA: ACM, 2008. (SIGMOD '08), p. 739–752. ISBN 978-1-60558-102-6. Disponível em: <http://doi.acm.org/10.1145/1376616.1376691>. Citado 4 vezes nas páginas 13, 16, 20 e 34.

COOPER, B. F. et al. Pnuts: Yahoo!'s hosted data serving platform. *Proc. VLDB Endow.*, VLDB Endowment, v. 1, n. 2, p. 1277–1288, ago. 2008. ISSN 2150-8097. Disponível em: <http://dx.doi.org/10.14778/1454159.1454167>. Citado 2 vezes nas páginas 14 e 21.

DECANDIA, G. et al. Dynamo: Amazon's highly available key-value store. In: *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*. New York, NY, USA: ACM, 2007. (SOSP '07), p. 205–220. ISBN 978-1-59593-591-5. Disponível em: <http://doi.acm.org/10.1145/1294261.1294281>. Citado 2 vezes nas páginas 14 e 21.

DEMERS, A. et al. Epidemic algorithms for replicated database maintenance. In: *Proceedings of the Sixth Annual ACM Symposium on Principles of Distributed Computing*. New York, NY, USA: ACM, 1987. (PODC '87), p. 1–12. ISBN 0-89791-239-X. Disponível em: <http://doi.acm.org/10.1145/41840.41841>. Citado na página 64.

FAN, B. et al. Cuckoo filter: Practically better than bloom. In: *Proceedings of the 10th ACM International on Conference on Emerging Networking Experiments and Technologies*. New York, NY, USA: ACM, 2014. (CoNEXT '14), p. 75–88. ISBN 978-1-4503-3279-8. Disponível em: <http://doi.acm.org/10.1145/2674005.2674994>. Citado na página 51.

GILBERT, S.; LYNCH, N. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, ACM, New York, NY, USA, v. 33, n. 2, p. 51–59, jun. 2002. ISSN 0163-5700. Disponível em: <http://doi.acm.org/10.1145/564585.564601>. Citado na página 14.

GRAY, J. Notes on data base operating systems. In: *Operating Systems, An Advanced Course*. London, UK, UK: Springer-Verlag, 1978. p. 393–481. ISBN 3-540-08755-9.

Disponível em: <http://dl.acm.org/citation.cfm?id=647433.723863>. Citado na página 18.

GRAY, J. et al. The dangers of replication and a solution. In: *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*. New York, NY, USA: ACM, 1996. (SIGMOD '96), p. 173–182. ISBN 0-89791-794-4. Disponível em: <http://doi.acm.org/10.1145/233269.233330>. Citado 3 vezes nas páginas 13, 16 e 17.

HERLIHY, M. P.; WING, J. M. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, ACM, New York, NY, USA, v. 12, n. 3, p. 463–492, jul. 1990. ISSN 0164-0925. Disponível em: <http://doi.acm.org/10.1145/78969.78972>. Citado 2 vezes nas páginas 14 e 27.

LAKSHMAN, A.; MALIK, P. Cassandra: A decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, ACM, New York, NY, USA, v. 44, n. 2, p. 35–40, abr. 2010. ISSN 0163-5980. Disponível em: <http://doi.acm.org/10.1145/1773912.1773922>. Citado 2 vezes nas páginas 14 e 21.

LAMPORT, L. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, ACM, New York, NY, USA, v. 21, n. 7, p. 558–565, jul. 1978. ISSN 0001-0782. Disponível em: <http://doi.acm.org/10.1145/359545.359563>. Citado 3 vezes nas páginas 21, 26 e 51.

LLOYD, W. et al. 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*. New York, NY, USA: ACM, 2011. (SOSP '11), p. 401–416. ISBN 978-1-4503-0977-6. Disponível em: <http://doi.acm.org/10.1145/2043556.2043593>. Citado 2 vezes nas páginas 14 e 21.

LLOYD, W. et al. Stronger semantics for low-latency geo-replicated storage. In: *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*. Lombard, IL: USENIX, 2013. p. 313–328. ISBN 978-1-931971-00-3. Disponível em: <https://www.usenix.org/conference/nsdi13/technical-sessions/presentation/lloyd>. Citado 3 vezes nas páginas 14, 21 e 60.

MATRI, P. et al. Towards efficient location and placement of dynamic replicas for geo-distributed data stores. In: *Proceedings of the ACM 7th Workshop on Scientific Cloud Computing*. New York, NY, USA: ACM, 2016. (ScienceCloud '16), p. 3–9. ISBN 978-1-4503-4353-4. Disponível em: <http://doi.acm.org/10.1145/2913712.2913715>. Citado na página 64.

METWALLY, A.; AGRAWAL, D.; ABBADI, A. E. Efficient computation of frequent and top-k elements in data streams. In: _____. *Database Theory - ICDT 2005: 10th International Conference, Edinburgh, UK, January 5-7, 2005. Proceedings*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005. p. 398–412. ISBN 978-3-540-30570-5. Disponível em: <http://dx.doi.org/10.1007/978-3-540-30570-5_27>. Citado na página 64.

NISHTALA, R. et al. Scaling memcache at facebook. In: *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*. Lombard, IL: USENIX, 2013. p. 385–398. ISBN 978-1-931971-00-3. Disponível em: <https://www.usenix.org/conference/nsdi13/technical-sessions/presentation/nishtala>. Citado 4 vezes nas páginas 6, 12, 39 e 40.

PACITTI, E. p.; MINET, P.; SIMON, E. *Fast Algorithms for Maintaining Replica Consistency in Lazy Master Replicated Databases*. [S.l.], 1999. Projet RODIN, Projet REFLECS. Disponível em: <https://hal.inria.fr/inria-00077204>. Citado na página 13.

PREGUICA, N. et al. A commutative replicated data type for cooperative editing. In: *2009 29th IEEE International Conference on Distributed Computing Systems*. [s.n.], 2009. p. 395–403. ISSN 1063-6927. Disponível em: <http://ieeexplore.ieee.org/abstract/document/5158449/>. Citado na página 65.

ROH, H.-G. et al. Replicated abstract data types: Building blocks for collaborative applications. *J. Parallel Distrib. Comput.*, Academic Press, Inc., Orlando, FL, USA, v. 71, n. 3, p. 354–368, mar. 2011. ISSN 0743-7315. Disponível em: <http://dx.doi.org/10.1016/j.jpdc.2010.12.006>. Citado na página 32.

SHAPIRO, M. et al. *A comprehensive study of Convergent and Commutative Replicated Data Types*. [S.l.], 2011. 50 p. Disponível em: <https://hal.inria.fr/inria-00555588>. Citado na página 65.

SHAPIRO, M. et al. Conflict-free replicated data types. In: *Proceedings of the 13th International Conference on Stabilization, Safety, and Security of Distributed Systems*. Berlin, Heidelberg: Springer-Verlag, 2011. (SSS'11), p. 386–400. ISBN 978-3-642-24549-7. Disponível em: <http://dl.acm.org/citation.cfm?id=2050613.2050642>. Citado 4 vezes nas páginas 21, 25, 32 e 64.

TERRY, D. B. et al. Session guarantees for weakly consistent replicated data. In: *Proceedings of the Third International Conference on Parallel and Distributed Information Systems*. Washington, DC, USA: IEEE Computer Society, 1994. (PDIS '94), p. 140–149. ISBN 0-8186-6400-2. Disponível em: <http://ieeexplore.ieee.org/abstract/document/331722/>. Citado na página 21.

VIOTTI, P.; VUKOLIć, M. Consistency in non-transactional distributed storage systems. *ACM Comput. Surv.*, ACM, New York, NY, USA, v. 49, n. 1, p. 19:1–19:34, jun. 2016. ISSN 0360-0300. Disponível em: <http://doi.acm.org/10.1145/2926965>. Citado 4 vezes nas páginas 13, 14, 21 e 23.

VOGELS, W. Eventually consistent. *Queue*, ACM, New York, NY, USA, v. 6, n. 6, p. 14–19, out. 2008. ISSN 1542-7730. Disponível em: <http://doi.acm.org/10.1145/1466443.1466448>. Citado 2 vezes nas páginas 14 e 21.

WIESMANN, M. et al. Understanding replication in databases and distributed systems. In: *Proceedings 20th IEEE International Conference on Distributed Computing Systems*. [s.n.], 2000. p. 464–474. ISSN 1063-6927. Disponível em: <http://ieeexplore.ieee.org/abstract/document/840959/>. Citado na página 18.

ZIPF, G. *Human Behaviour and the Principle of Least-Effort*. [S.l.]: Addison-Wesley, 1949. Citado na página 12.