Replicação descentralizada em bancos de dados distribuídos usando o algoritmo Paxos

Danilo J. P. Ferreira[1] , **Sahudy M. González**[1] , **Gustavo M. D. Vieira**[1]

[1]DComp – CCGT – UFSCar
Sorocaba, São Paulo, Brasil

danilojpferreira@gmail.com, sahudy@ufscar.br, gdvieira@ufscar.br

Abstract. *This paper proposes a way to integrate a replication mechanism using Paxos into any in-memory DBMS. This proposal adds the reliability and data durability provided by the Paxos algorithm to the DBMS, while maintaining its typical programming interface and other characteristics. Aditionally, this integration provides a robust fault recovery mechanism, which provides reliability comparable to disk-based DBMS. To validate the proposal a prototype was built using Redis and the Treplica framework, including a basic set of performance tests.*

Resumo. *Neste trabalho foi proposto uma forma de integrar um mecanismo de replicação usando o algoritmo Paxos a qualquer SGBD em memória principal. Esta proposta adiciona ao SGDB a confiabilidade e durabilidade de dados fornecidos pelo algoritmo Paxos, mantendo a interface de programação típica destes sistemas e preservando suas demais características. Outra consequência positiva desta integração é um robusto mecanismo de recuperação de falhas, que provê confiabilidade equiparável a SGBDs em disco. Para demonstrar a proposta construiu-se um protótipo usando o Redis e o* framework *Treplica, incluindo um conjunto inicial de testes de desempenho.*

1. Introdução

Por décadas os SGBDs relacionais forneceram consistência de dados por meio das transações e suas propriedades ACID (Atomicidade-Consistência-Isolamento-Durabilidade). Todavia, os SGBDs distribuídos precisam estender a consistência a nível das réplicas, inclusive para atender um alto volume de dados e de requisições concorrentes. A consistência de dados em SGBDs NoSQL que usam replicação e visam escalabilidade pode ser vista, no mínimo, em duas situações: a consistência de dados a nível de transação e a consistência a nível de replicação em que cada réplica no melhor caso precisa ficar atualizada antes de atender requisições de leitura e escrita [Sadalage and Fowler 2013].

Os sistemas NoSQL foram projetados de maneira que seus dados possam ser replicados e operados em *clusters*, ofertando assim um modelo distribuído de processamento de dados. Para preservar a disponibilidade e confiabilidade dos dados nesses sistemas, o ideal seria alcançar a consistência forte de replicação, em que todas as réplicas devem refletir a consistência em todos os momentos [Ozsu and Valduriez 2020]. Porém, a estratégia de replicação implementada em sistemas NoSQL é a assíncrona, em que as réplicas ficam divergentes na janela de sincronização [Sadalage and Fowler 2013].

Há várias razões pelas quais a sincronização entre nós é necessária. A primeira é relacionada a como manter a consistência e durabilidade de dados mesmo no caso de falhas. Uma falha na durabilidade da replicação ocorre quando um nó processa uma atualização, mas falha antes que a atualização seja replicada para os outros nós [Sadalage and Fowler 2013]. Por exemplo, em uma arquitetura de distribuição primário/secundário e mesmo na descentralizada, todas as escritas não transmitidas para as réplicas serão efetivamente perdidas em caso do nó falhar. Se o nó voltar a ficar disponível, suas atualizações entrarão em conflito com as atualizações que aconteceram desde então. A ocorrência de conflitos é a segunda razão que justifica a necessidade de sincronização.

Para tentar solucionar esses problemas, os efeitos da efetivação local de uma transação precisam ser estendidos de forma distribuída, e todos os nós envolvidos na execução de uma transação distribuída precisariam concordar em confirmar a transação antes que seus efeitos se tornem permanentes. O protocolo 2PC (*Two-Phase-Commit*), que é referência na literatura [Ozsu and Valduriez 2020], propõe que uma transação seja efetivada após todas as réplicas ficarem sincronizadas, e consiste em duas fases: votação e efetivação. A efetivação de uma transação é iniciada por um nó coordenador e um ou mais nós participantes. Neste processo consensual, a validação de uma transação fica por conta da votação, sendo que ao haver um voto "não", a transação será abortada globalmente. Porém, este protocolo depende fortemente de um nó coordenador de transação e não é tolerante a falhas. Se qualquer réplica falhar, *o protocolo chegará ao fim sem sucesso*.

Neste contexto, o objetivo deste artigo é descrever uma proposta de uso de replicação síncrona tolerante a falhas para SGBDs distribuídos, especificamente para sistemas NoSQL, de maneira a atingir uma consistência de replicação forte. Ainda, se um banco de dados puder ser mantido principalmente em memória primária, aplicar atualizações dos dados também na memória primária e descarregar periodicamente as alterações para memória secundária, ele poderá fornecer uma capacidade de resposta bem maior para as requisições. Com isto, o artigo apresenta o Treplica-Redis, que implementa replicação síncrona em uma arquitetura descentralizada para o Redis, além de um modelo de recuperação de falhas. Apesar da proposta desenvolvida poder ser aplicada para qualquer SGBD NoSQL, escolheu-se o Redis. Este sistema de armazenamento chave-valor em memória é inerentemente simples, possui primitivas facilmente implementáveis e é um dos cinco sistemas NoSQL mais usados no mundo [Kamaruzzaman 2021]. Por meio de um experimento, a pesquisa apresenta, adicionalmente, o *trade-off* entre desempenho e consistência ao utilizar um modelo de replicação síncrona.

O artigo está organizado da seguinte forma. Inicialmente descrevemos alguns conceitos básicos na Seção 2. Na sequência descrevemos a nossa proposta, sua arquitetura de software e os algoritmos de sincronização e recuperação nas Seções 3, 4 e 5. Na Seção 6 mostramos o nosso conjunto inicial de testes de desempenho. O artigo conclui com os trabalhos relacionados na Seção 7 e as considerações finais na Seção 8.

2. Fundamentos de Replicação de Dados

Esta seção apresenta o embasamento conceitual sobre replicação de dados em bancos de dados e sistemas distribuídos.

2.1. Arquiteturas de Replicação de Dados

A replicação de dados em SGBDs distribuídos é o processo de criar e manter cópias dos dados em diferentes máquinas, e geralmente em diferentes localidades físicas, visando garantir a disponibilidade destes dados, tolerância a falhas e a escalabilidade do sistema [Ozsu and Valduriez 2020]. Chamamos cada uma destas cópias de *réplicas*, e cada uma das máquinas onde as réplicas podem residir de *nós*.

Existem dois modelos principais de replicação de dados utilizados em SGBDs distribuídos [Sadalage and Fowler 2013]:

Primário/Secundário (ou P/S): no qual uma réplica é eleita como primária e apenas esta pode lidar com escritas, e as demais réplicas são cópias secundárias. As réplicas secundárias se sincronizam com a réplica primária e podem lidar apenas com as leituras.

Descentralizado (ou *P2P*): no qual qualquer nó da rede pode realizar escritas e leituras nas suas réplicas, e os mesmos coordenam-se para sincronizar a replicação de seus itens de dados.

A arquitetura de replicação de dados primário/secundário é útil quando há um conjunto de dados com uso intensivo de leitura. Porém, para aplicações com muitas requisições de escrita, o nó primário pode se tornar um ponto de gargalo e um ponto único de falha, o que pode introduzir inconsistências nos dados. Já a arquitetura descentralizada, evita o carregamento de todas as escritas em um único ponto de falha, pode passar por falhas de nó sem perder o acesso aos dados, e pode facilmente adicionar nós para melhorar seu desempenho. Porém, deve atender a consistência de replicação.

A consistência de replicação visa garantir que um registro tenha o mesmo valor quando lido em réplicas diferentes [Sadalage and Fowler 2013]. Portanto, nesta arquitetura, os nós coordenam a sincronização das cópias dos itens de dados. Na janela de sincronização entre nós, até todas as cópias ficarem atualizadas, podem ocorrer inconsistências devido a conflitos.

Os conflitos podem ser: (1) de escrita-escrita, que ocorrem quando dois clientes tentam, simultaneamente, atualizar o mesmo registro em nós diferentes; (2) de escrita-leitura, que ocorrem quando há uma requisição de leitura em um nó desatualizado, estando a versão mais atualizada desse registro em outro nó que ainda não completou a sincronização. O desafio de gerenciar conflitos, e assim garantir a consistência dos dados, passa a ser mais relevante quando há altas taxas de mudança nos dados, o que implica em mais trocas de mensagens entre os nós para manter os dados globalmente acessíveis e atualizados [Gribble et al. 2001].

2.2. Consistência de dados em Sistemas Distribuídos

SGBDs alcançam a consistência dos dados por meio de transações, cuja definição é garantida implementando as propriedades ACID [Ozsu and Valduriez 2020]. Uma transação é composta por um sequência de operações. Dentre outras, as operações podem ser de escrita ou leitura a itens de dados no banco de dados. Em um SGBD distribuído, para garantir a consistência de replicação ao utilizar uma arquitetura de replicação descentralizada, o SGBD deve cuidar da sincronização e dos conflitos entre as réplicas.

Uma abordagem para entender esta consistência é apresentada na área de sistemas distribuídos. Supondo que os dados são representados por registros, podemos entender

a consistência observando como estes se comportam na presença de concorrência. A consistência de um registro pode ser classificada em três níveis, sendo do menos para o mais confiável: seguro (*safe*), regular (*regular*) e atômico (*atomic*) [Pierce and Alvisi 2000]:

Registro seguro: se uma leitura, não concorrente a nenhuma escrita, retorna o valor da escrita mais recente, e uma leitura concorrente a uma ou mais escritas retorna o valor da última escrita finalizada ou de uma das escritas concorrentes.

Registro regular: se uma leitura, não concorrente a nenhuma escrita, retorna o valor da escrita mais recente, e uma leitura concorrente a uma ou mais escritas retorna ou o valor anterior às escritas concorrentes, ou o valor posterior às escritas concorrente, mas nunca um valor intermediário, e

Registro atômico: se as leituras e escritas forem ordenadas totalmente, de forma que não haja sobreposição entre transações, e qualquer leitura sempre reflita o último valor escrito.

A replicação dos dados em arquitetura P/S, na qual uma única réplica é utilizada como destino para todas as escritas de dados, torna simples manter a consistência dos dados. Isto ocorre pois não existe concorrência de escrita entre réplicas, já que todas as transações são processadas em um único nó, sendo então um registro regular. Porém, em uma arquitetura de replicação descentralizada, no qual qualquer réplica pode aplicar as atualizações nos dados, não há uma maneira trivial de controlar a concorrência e garantir que um mesmo registro não será divergente nas demais réplicas.

Em geral, as estratégias para resolver o problema de replicação descentralizada estão relacionadas ao uso de um quórum para tomar decisões, tal que estas decisões sejam consistentes em todos os nós da rede. Uma estratégia utilizada é o uso de algoritmos de consenso (por exemplo: Paxos [Lamport 1998]), nos quais um conjunto de nós deve chegar a um acordo sobre a identidade e, principalmente, a *ordem* das transações executadas no sistema.

2.3. Paxos e Replicação Ativa

Uma das técnicas para garantir a consistência dos dados em aplicações descentralizadas é a replicação síncrona [Schneider 1990], também conhecida como replicação ativa. Nesta, todos os nós se comportam como máquinas de estado deterministas que possuem o mesmo estado inicial. Ao serem submetidas a uma mesma sequência de eventos deterministas, que alterem seu estado, devem permanecer iguais. A consistência é garantida não por uma técnica de resolução de conflitos em si, mas por uma abordagem serial que impede que os conflitos ocorram. Desta forma, a chave para se implementar replicação ativa é ser capaz de propagar uma sequência de mensagens ordenadas, de tal forma que esta ordenação seja respeitada por todas as réplicas do sistema.

Criar uma ordenação total de mensagens em um sistema distribuído é um problema difícil, que pode ser resolvido por redução ao problema de consenso [Cachin et al. 2011]. Consenso corresponde a decisão por parte de um grupo de processos de um valor único que todos concordem. Um algoritmo muito eficiente que combina consenso, ordenação e replicação é o algoritmo Paxos [Lamport 1998]. Resumidamente, para resolver o consenso usando Paxos as réplicas executam várias rodadas onde uma proposta de decisão é enviada a uma réplica especial, o coordenador. O coordenador é capaz de decidir,

usando quóruns de réplicas, um valor de consenso único mesmo na presença de falhas parciais das réplicas envolvidas.

Usando a solução de consenso, Paxos define uma forma de entregar um conjunto de mensagens totalmente ordenadas. A ordem de entrega destas mensagens é determinada por uma sequência de inteiros positivos, tal que a cada inteiro corresponde uma instância individual de consenso. Cada instância i terá um valor decidido, que corresponde a i-ésima mensagem a ser entregue na sequência de mensagens. Cada instância de consenso é independente das demais e várias instâncias podem estar em curso ao mesmo tempo. Como Paxos suporta o modelo de falhas falha-e-recuperação, ambos os algoritmos exigem que os agentes armazenem estado em memória não volátil [Lamport 1998]. Este estado é composto primariamente por um registro das instâncias iniciadas e de suas decisões.

2.4. Treplica: Um Arcabouço para Replicação Síncrona

O *framework* Tréplica [Vieira and Buzato 2010] foi criado para construir aplicações distribuídas descentralizadas, em um modelo de replicação síncrona, fornecendo ao desenvolvedor uma camada de replicação responsável pela consistência, propagação dos dados e a recuperação dos mesmos em caso de falhas. Treplica aplica o algoritmo de ordenação e consenso Paxos e é construído na linguagem Java.

Treplica implementa o algoritmo Paxos e usa a ordenação de mensagens obtida para replicar o *estado* de cada réplica. Este estado é na prática um objeto Java que contém todos os dados da réplica, e fica sob responsabilidade do *framework*. A réplica deve acessar o seu estado através de um componente do *framework* chamado de *StateMachine*. Dois métodos são expostos por este componente: *getState()* e *execute(Action)*. O método *getState()* é responsável por obter o estado de replicado, na forma de uma referência ao objeto Java. O método *execute(Action)* é responsável por solicitar uma alteração no estado armazenado pelo *StateMachine* e recebe um parâmetro do tipo *Action*. Uma *Action* é um objeto que representa uma modificação do estado. Ou seja, uma *Action* implementa uma transação.

Após receber uma chamada de *execute(Action)*, Treplica usa a sua implementação do algoritmo Paxos para enviar como uma mensagem ordenada esta *Action* para todas as réplicas do sistema. Um importante ponto a ser destacado é que várias réplicas podem executar ações concorrentemente, mas o algoritmo Paxos garantirá que todas as ações serão conhecidas por todas as réplicas em uma mesma ordem. A execução de uma *Action* não ocorre imediatamente quando *execute(Action)* é chamado, mas somente após a ordenação da mesma em relação às demais ações desta ou de outras réplicas. Desta forma, Treplica possui uma *thread* própria que recebe os objetos *Action*, os executa e notifica quaisquer clientes *locais* que estejam esperando o retorno da execução.

Ao usar o algoritmo Paxos para realizar a ordenação das ações, Treplica herda o suporte ao modelo falha-e-recuperação do Paxos. Desta forma, todos os passos do algoritmo são registrados em memória persistente e podem ser refeitos em caso de falha. Na sua forma mais simples, o estado armazenado por *StateMachine* é recuperado reaplicando todas as ações na ordem que foram decididas, recompondo então o seu conteúdo no momento da falha. Estes passos necessários para a recuperação são armazenados em um arquivo de registro sequencial que armazena as ações (*command log*), uma opção mais

eficiente do que registrar o resultado das ações. Além do registro de ações, Treplica pode realizar um instantâneo (*checkpoint*), que grava em memória secundária todo o estado de réplica armazenado no *StateMachine*. Quando um nó colapsa e logo depois se recupera, Treplica restaura o estado do último instantâneo e caso haja registro de ações não inclusas no instantâneo, ele inicia a reexecução das ações ali registradas. A reexecução é idêntica a que acontece quando uma ação é propagada entre as réplicas.

3. Treplica-Redis

A proposta deste trabalho é um mecanismo que permita a integração entre um estado replicado usando-se o algoritmo Paxos e SGBDs em memória principal. A abordagem adotada para esta integração é a utilização de réplicas de um SGBD otimizado para memória principal sem alterações, e coordenar estas réplicas usando um mecanismo de replicação externo. Desta forma, o sistema proposto é dividido em três camadas interconectadas entre si, como mostra a Figura 1:

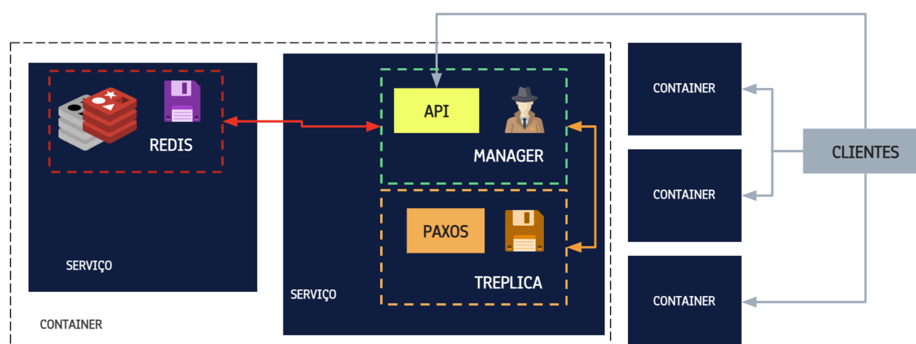

Figura 1. Arquitetura da Sistema Proposto

Manager: Esta camada é responsável pela interface com o cliente e pela integração entre as demais camadas. O Manager possui um estado (armazenado pelo Treplica) composto por uma *lista de transações* ordenadas, e um inteiro (*version_stamp*) que conta o número de transações que já passou por esta lista. Usando este estado o Manager faz a sincronização entre as transações que compõem a sua lista replicada de transações e o estado da camada do Redis, de forma que o estado do SGBD local corresponda ao estado replicado, como descrito na Seção 4. Em caso de falha, o Manager recupera o seu estado com o auxílio do Treplica e realiza um processo de recuperação similar a sincronização, descrito na Seção 5.

Treplica: Esta camada armazena e replica o estado do Manager, sendo indiretamente responsável pela ordenação das transações. O Manager define uma única *Action* do Treplica que insere uma nova transação na lista. Como responsável pelo estado (a lista de transações do Manager) o Treplica faz com que este estado chegue a todas a réplicas e também preserva e recupera este estado na presença de falhas, como descrevemos na Seção 2.4.

Redis: Esta camada contém o SGBD em memória principal onde os *dados* residem, em contraste à lista das transações armazenada pelo Manager. O SGBD usado nesta proposta é o Redis operando com suas configurações padrão de forma centralizada, ou seja, sem conhecimento das demais réplicas. Este SGBD guarda os

dados em memória principal junto a um *version_stamp*, que reflete o número de transações efetivamente executadas e é usado no processo de sincronização descrito na Seção 4. Mesmo sendo o repositório de dados em memória principal, o Redis permite a criação de um instantâneo armazenado em memória persistente, sendo este instantâneo usado para acelerar o processo de recuperação, descrito na Seção 5.

O Treplica-Redis lida com dois modelos de transações: modelo atômico e modelo regular. As transações no modelo atômico seguem a definição de consistência de dados operando um registro atômico. As transações no modelo regular seguem a definição de consistência de dados operando um registro regular. A nível de isolamento, trata-se de uma transação de leitura confirmada, ou seja, permite as violações de leitura não repetível e leitura fantasma.

Vale destacar que cada transação no Treplica-Redis é composta apenas por uma operação, que pode ser de leitura ou escrita. Porém, a implementação pode ser estendida para abarcar transações com mais operações. As transações segundo seu modelo são implementadas como explicado a seguir.

Modelo Atômico: as transações de leitura quanto de escrita são previamente ordenadas pelo algoritmo Paxos, antes de serem efetivadas e retornarem ao cliente. Assim, a requisição é recepcionada pelo Manager é encaminhada ao *framework* Treplica que deverá garantir a ordenação total da mesma. Ao ter um retorno das transações ordenadas, o Treplica informa ao Manager, realizando todo o processo de sincronização. Durante o processo de sincronização é verificado se no nó em questão há um cliente aguardando o retorno retorno da transação, e caso haja, o retorno do Redis é adicionado a uma tabela, de tal maneira que o processo responsável por responder o cliente pode acessar a resposta e completar sua execução.
As transações com mais de uma operação, não implementadas nesta primeira versão, devem operar apenas no modelo atômico a fim de garantir consistência forte de replicação.

Modelo Regular: no modelo regular, a leitura e escrita são abordadas de maneira diferentes. As transações compostas por operações de leituras são executadas imediatamente no Redis, retornando o dado encontrado para o cliente, sem o uso do algoritmo de consenso, podendo trazer um dado que não seja o mais recente (consistência relaxada). Já as transações que possuem escritas são inserida em uma fila, e antes mesmo de serem processadas retorna uma confirmação para o cliente. Esta confirmação indica que uma transação foi inserida em uma fila mas não ainda ordenada. Futuramente, está será ordenada e executada no Redis, porém, até que isso ocorra, violações de leitura confirmada podem ocorrer.

4. Sincronização

A sincronia entre as camadas Redis e Manager se dá por meio do uso de *version_stamp*. Este valor representa estados sutilmente diferentes em cada camada. Na camada Redis, o *version_stamp* é um registro armazenado no SGBD que guarda o número de transações efetivamente executadas no SGBD. Logo, ele é um identificador do estado atualmente armazenado no SGBD. Na camada Manager, o *version_stamp* é um inteiro que conta o número de transações que já passou pela lista replicada de transações. Usando o Treplica,

o Manager tem a garantia que, se uma transação é adicionada à esta lista, a mesma será replicada e aparecerá na mesma ordem na lista de transações de todas as réplicas.

Então, o Manager verifica os *version_stamps* seu e do Redis. Se os dois são iguais, indica que não existem transações a serem processadas pelo Redis e a lista de transações do Manager deve possuir zero transações. Por outro lado, se o Manager possui um inteiro maior, as transações do Manager são executadas no Redis, uma a uma, seguindo a ordenação da lista. Para cada transação executada, o *version_stamp* do Redis é incrementado e a transação é removida do estado do Manager, até que os *version_stamps* sejam iguais. Por fim, caso o Redis possua um inteiro maior, as transações na lista do Manager vão sendo descartadas, sem serem aplicadas ao Redis, até que mais transações sejam recepcionadas e os *version_stamps* sejam iguais.

Observe que os estados em si não são propagados entre as réplicas, apenas as transações. O fluxo das transações pelo Manager ocorre em dois momentos. O primeiro é quando ocorre a ordenação e a executada de uma ação do Treplica. Esta ação adiciona no final da lista de transações uma nova transação a ser executada no Redis e incrementa o *version_stamp* do Manager. O segundo momento é quando a transação em si é executada no Redis, de acordo com o algoritmo descrito no parágrafo anterior. Uma vez que esta transação já está no SGBD, não há necessidade de mantê-la na lista de transações do Manager, então esta transação é removida da lista sem gerar uma nova ação. Ou seja, esta remoção é uma alteração apenas no estado local. Desta maneira sabemos que o estado do Manager é a diferença simétrica entre todas as transações já ordenadas e as transações implementadas pelo Redis.

Descreveremos abaixo alguns detalhes das implementações de transações pelo Treplica-Redis nos modelos atômico e regular, partindo de uma requisição do usuário. Ao recepcionar uma transação o Treplica-Redis verifica qual o modelo da transação (atômico/regular) e se o método corresponde a uma leitura ou escrita.

Caso seja uma leitura no modelo regular, ele executa a transação imediatamente no Redis e retorna o resultado ao cliente. Caso seja uma escrita no modelo regular, ele adiciona a transação ao seu estado usando o Treplica e imediatamente retorna uma confirmação ao usuário que a transação foi enfileirada. Já no modelo atômico, o Treplica-Redis cria um identificador único para cada requisição no momento que ela é recebida e adiciona-o em uma lista de requisições pendentes. Este identificador será utilizado posteriormente para retornar a resposta ao usuário. Então, ele adiciona a transação ao seu estado usando o Treplica enquanto um *listener* aguarda cada uma das alterações de estado realizada pelo Treplica. Quando a transação solicitada por este nó for executada, seu valor de retorno é associado a uma requisição pendente usando o identificador criado anteriormente e retornado ao cliente. Independente do modelo de transação, toda vez que o estado do Manager é alterado o processo de sincronização descrito anteriormente é realizado.

Podemos ilustrar a aplicação destes algoritmos em um exemplo a seguir no qual o usuário realiza duas ações:

Ação 1 (Figura 2): O cliente envia uma transação com a operação *SET* para a chave *A* no valor 15. Como a chave *A* não existe, é esperado que a chave *A* valor 15 seja criado.

Ação 2 (Figura 3): Chave *A* já existe com o valor 15, como podemos ver, após receber

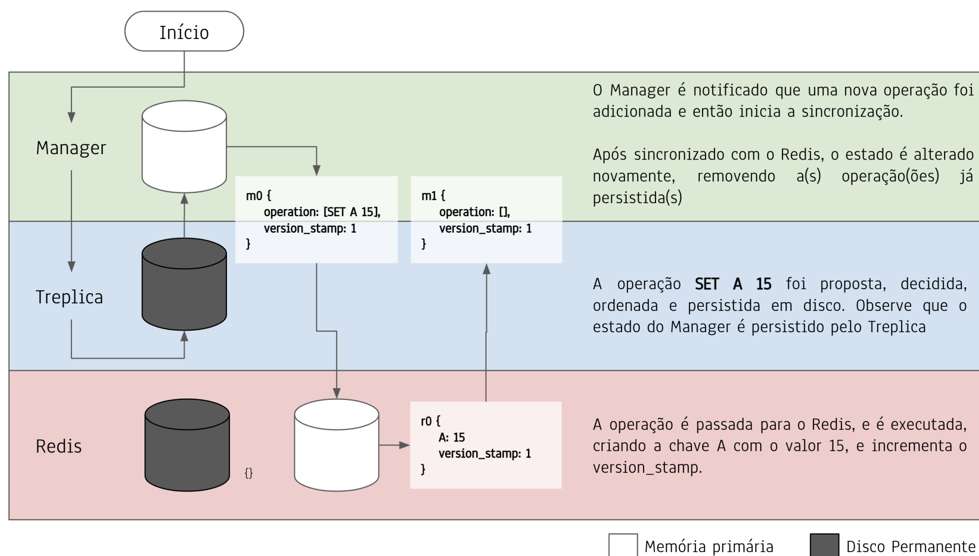

Figura 2. Ação 1: Criando um registro

uma transação com a operação *INCR* para a chave *A* no valor 3, é esperado que o valor desta chave seja atualizado para 18:

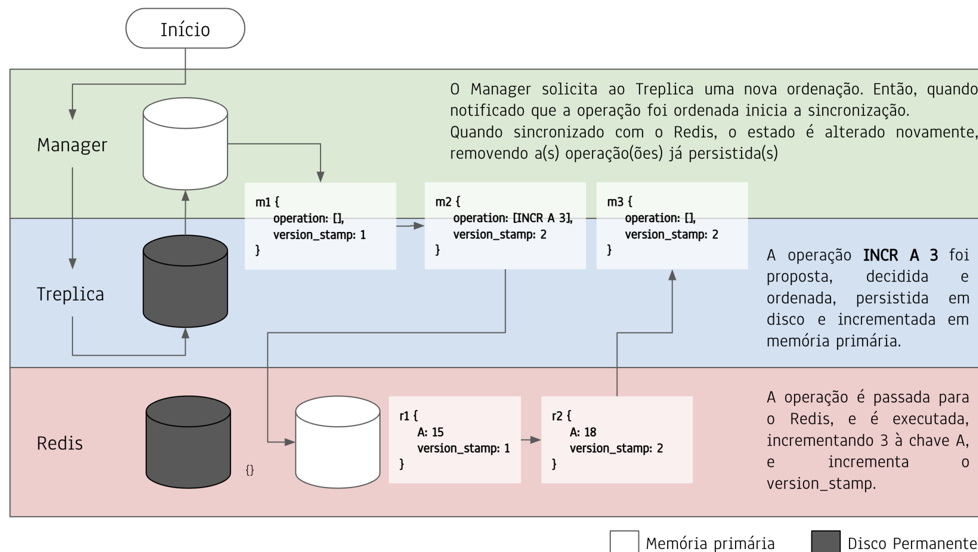

Figura 3. Ação 2: Atualizando um registro

5. Recuperação

Mesmo sendo um sistema para armazenamento de dados em memória principal, as camadas do Treplica-Redis preservam o estado em memória persistente para garantir a consistência dos dados e recuperação de falhas. Cada camada faz isto de maneira diferente, mas integrada, de forma a ser possível recuperar o estado do Treplica-Redis em caso de falha.

O Manager funciona apenas como coordenador e deixa todo o seu estado sob a guarda do Treplica, não se preocupando diretamente com escritas em memória persistente.

O Treplica armazena em disco todos os passos do algoritmo Paxos, como descrito na Seção 2.4, o que inclui o histórico de mensagens e, como consequência, o estado do Manager sob sua responsabilidade. Treplica também guarda instantâneos, que incluem não só o estado interno do algoritmo, mas o conteúdo da fila de transações do Manager. O Redis é responsável apenas por gerar instantâneos dos seus dados em memória persistente usando o seu mecanismo padrão. Tanto no Redis quanto no Treplica, os instantâneos têm como objetivo tornar o reinício mais rápido em caso de recuperação de falhas, não sendo essenciais para o funcionamento d o sistema proposto. Os instantâneos são criados em ambas camadas seguindo políticas implementadas no Manager.

No Treplica-Redis pressupõe-se a ocorrência de falha do tipo **falha-e-recuperação** nas camadas Manager (que inclui o Treplica) e Redis. Para falhas no Manager, o próprio contêiner que engloba o sistema proposto é capaz de, se identificado falhas, reiniciar toda a réplica mantendo o mesmo disco em todas as inicializações. Já para as falhas na camada do Redis, o Manager atua como um detector de falhas por meio de requisições de checagem de saúde, tal que a não obtenção de respostas um período de tempo pré-definido é considerado como uma falha que reinicia todo o sistema.

Sendo detectado uma falha, ocorre o reinício completo do Treplica-Redis, executado pelo Manager e Redis. O processo de inicialização do SGBD Redis busca o último instantâneo realizado e aplica-o como estado inicial. É importante lembrar que o *version_stamp* faz parte do estado do Redis carregado do disco, o identificando. O estado do Manager é recuperado pelo Treplica que, como descrito na Seção 2.4, primeiro restaura o último instantâneo e então reexecuta as ações, desta forma reconstruindo a lista de transações do Manager. Similarmente ao Redis, o estado do Manager também inclui o seu *version_stamp*. Ao término deste processo de recuperação, a Manager terá o estado imediatamente anterior à falha, porém não necessariamente sincronizado com o Redis. A recuperação então conclui com a execução do processo de sincronização exatamente como descrito na seção anterior.

6. Trabalho Experimental

Para compreender o nível de consistência e desempenho fornecidos pela proposta, foi desenvolvido um *benchmarking* entre duas variantes: a primeira já descrita neste artigo, e a segunda, nomeada de Sentinel-Redis [Souza 2022], é uma cópia na qual o componente Treplica é substituído pela ferramenta de replicação nativa no Redis, Sentinel. A diferença entre os dois sistemas é que enquanto o Treplica-Redis usa o Treplica para garantir consistência e replicação na arquitetura *P2P*, o Sentinel-Redis usa a combinação Redis e Sentinel que possui uma consistência relaxada e arquitetura de replicação P/S.

A fim de garantir um ambiente de uso similar e ofertar o máximo possível de consistência, o sistema de comparação aplica as seguintes configurações ao servidor Redis: *appendonly yes*, *appendfsync always*, *min-replicas-to-write 1*, *min-replicas-max-lag 5*, *replica-serve-stale-data no*,*sentinel down-after-milliseconds mymaster 5000*, *sentinel failover-timeout mymaster 30000*.

Mesmo com estas configurações, que buscam trazer ao Redis uma maior consistência dos dados, as operações de escrita são executadas apenas no nó primário. Para que, antes de qualquer confirmação, as escritas sejam confirmadas nas réplicas é preciso enviar adicionalmente o comando WAIT n, onde n é o número de réplicas que devem confirmar

uma escrita antes que ela seja considerada completa. Vale destacar que apesar desta configuração que proporciona uma máxima consistência no Redis utilizando sua replicação nativa, o uso do comando WAIT e o Sentinel apresenta problemas de perda de escritas em cenários específicos[1]. Portanto, ofertam uma consistência relaxada, enquanto a aplicação Treplica-Redis oferta uma consistência forte.

6.1. Cenários de Teste

Os testes foram executados na mesma configuração de ambiente, com os dois sistemas distribuídos em 3 nós com 1 réplica cada. Cada nó tem configurações idênticas, com processador Intel i3-9100, 8 GB de memória RAM, disco rígido de 500 GB a 7200 RPM e rede de baixa latência com capacidade até 1 Gbit/s.

Os experimentos desenvolvidos executam uma sucessão de escritas que criam uma chave aleatória e persiste como valor um arquivo JSON. O tamanho deste arquivo JSON é alterado de acordo com a instância do experimento, que pode ser de 4 B, 350 B, 1,4 kB ou 3,5 kB. Os testes enviam estas requisições de escrita com uma taxa de transações por segundo pré-definida. Uma taxa muito alta irá criar uma longa fila de espera, o que aumenta desnecessariamente os tempos de espera. Assim, determinamos por meio de tentativa e erro a maior taxa que refletisse o desempenho efetivo da aplicação. Para o Treplica-Redis a taxa definida foi de 500 t/s e para o Sentinel-Redis foi de 1100 t/s.

Cada teste tem a duração de 120 segundos é e executado quatro vezes para cada tamanho de arquivo escrito em cada aplicação (Treplica-Redis e Sentinel-Redis). Os sistemas sob teste são configurados para operar com três réplicas e as requisições são distribuídas de maneira uniforme entre todas as réplicas. Ao término do teste os dados brutos são processados para remover o segundo inicial e final, visto que este tempo é utilizado para o aquecimento e desaquecimento da aplicação.

6.2. Resultados consolidados e comparação

A comparação não tem como objetivo definir qual variante é melhor que outra, mas sim identificar o *trade-off* entre consistência e desempenho. Após avaliar os resultados (Tabela 1), foi possível observar que quanto maior o arquivo utilizado no teste, maior é o tempo de resposta médio dos sistemas e menor é o número de transações que os sistemas conseguem processar. Isto ocorre pois o conteúdo das mensagens registradas em disco e trocadas entre as réplicas é maior, consumindo mais recursos dos sistemas e da rede.

Tabela 1. Tabela de resultados do experimento

	Treplica-Redis			Sentinel-Redis			Comparativo
Tamanho	Trans./seg. (média)	Tempo de resp./Trans. (média)	Trans. totais	Trans./seg. (média)	Tempo de resp./Trans. (média)	Trans. totais	Trans./seg.
4 B	213,92	1.750 ms	25.885	1.796,09	243 ms	217.327	-8,34x
350 B	100,96	3.704 ms	12.217	1.251,83	371 ms	151.472	-12,40x
1,4 kB	79,70	4.526 ms	9.644	869,40	563 ms	103.329	-10,91x
3,5 kB	32,82	11.109 ms	3.972	333,33	1.318 ms	40.334	-10,16x

Trans. (Transação), resp. (resposta), seg. (segundo)

Percebe-se também que os resultados dos sistemas são bem distintos entre si. Para o Treplica-Redis em comparação com o Sentinel-Redis, os outros parâmetros indicam

[1] `https://aphyr.com/posts/313-jepsen-redis`

uma redução no desempenho, que vai de 8,4 a 12,4 vezes, porém, com desempenho satisfatório para atender grande parte das aplicações web.

7. Trabalhos relacionados

Nos últimos anos, pesquisas têm abordado a replicação de dados em SGBDs distribuídos seja usando replicação primário/secundário ou replicação descentralizada. No primeiro caso, tem-se, por exemplo, a proposta descrita em [Gonçalves et al. 2022], onde os autores apresentam um sistema de armazenamento chave-valor com garantias de consistência forte e escalabilidade. Os resultados experimentais mostram que o sistema é competitivo se comparado a sistemas comerciais.

Já, abordagens em arquiteturas de replicação descentralizadas se preocupam também com a confiabilidade das réplicas. Em [Ezéchiel et al. 2020], descreve-se uma adaptação do protocolo 2PC, o 4PC, que propõe uma técnica de aninhamento de transações, aplicando um método de consenso da maioria que leva em consideração os nós das réplicas disponíveis, a identificação de conflitos e nós participantes indisponíveis.

Os protocolos 2PC e suas variações têm como desvantagem a grande troca de mensagens entre os nós das réplicas. Para contornar isto, os autores de [Lu et al. 2021] propõem agrupar transações que chegam dentro de uma janela de tempo curta em *epochs* ou unidades básicas de tempo nas quais todas as transações são efetivadas ou nenhuma é. Dentre as limitações desta abordagem tem-se (1) o desempenho indesejável quando uma única transação de execução longa atrasa a efetivação de um lote inteiro de transações, e (2) todas as transações em uma época terão de ser abortadas e re-executadas na ocorrência de uma falha.

8. Conclusão

Neste artigo, apresenta-se o Treplica-Redis, uma nova abordagem para atingir consistência de replicação e durabilidade em bancos de dados distribuídos com desempenho satisfatório, por meio da replicação síncrona em arquitetura descentralizada. O Treplica-Redis utiliza um algoritmo de consenso, é tolerante a falhas e livre de conflitos, garantindo serviço ininterrupto mesmo com múltiplas falhas. Apesar do uso do Redis neste trabalho, a proposta desenvolvida pode ser aplicada para qualquer SGBD distribuído.

Vale destacar que mesmo com uma configuração que proporcione uma máxima consistência para o Redis, este ainda apresenta problemas de perda de escritas em cenários específicos, já evidenciado por outros autores [Kingsbury 2013], os quais são sanados no Treplica-Redis. Por isto, sugere-se o uso do Treplica-Redis para os seguintes casos: (1) alta carga de leitura no modelo regular e escrita dos dados no modelo atômico. Por exemplo, em redes-sociais, os dados armazenados precisam ficar consistentes e as leituras por milhões de usuários simultâneos podem acontecer com atraso; (2) a escrita não precisa ser confirmada, porém a leitura precisa ser consistente. Por exemplo, aplicações de *IoT* coletam grande volume de dados de ambiente e a eventual perda de um destes não é relevante, mas as leituras precisam ser precisas pois guiam a tomada de decisão; (3) tanto a leitura quanto a escrita devem acontecer no modelo atômico, optando por uma consistência de replicação forte, como por exemplo, em aplicações bancárias.

Referências

Cachin, C., Guerraoui, R., and Rodrigues, L. (2011). *Introduction to reliable and secure distributed programming*. Springer Science & Business Media.

Ezéchiel, K. K., Ojha, S. K., and Agarwal, R. (2020). A New Eager Replication Approach Using a Non-Blocking Protocol Over a Decentralized P2P Architecture. *International Journal of Distributed Systems and Technologies (IJDST)*, 11(2):69–100.

Gonçalves, J., Matos, M., and Rodrigues, R. (2022). SconeKV: A Scalable, Strongly Consistent Key-Value Store. *IEEE Transactions on Parallel and Distributed Systems*, 33(12):4164–4175.

Gribble, S., Halevy, A., Ives, Z., Rodrig, M., and Suciu, D. (2001). What can database do for peer-to-peer? In *Proceedings of Web and Databases*, pages 31–36.

Kamaruzzaman (2021). Top 10 databases to use in 2021. towardsdatascience.com/top-10-databases-to-use-in-2021-d7e6a85402ba. Accessed in Ago-17-2021.

Kingsbury, K. (2013). Jepsen: Redis. Accessed on 21 Jan. 2023.

Lamport, L. (1998). The part-time parliament. *ACM Transactions on Computer Systems (TOCS)*, 16(2):133–169.

Lu, Y., Yu, X., Cao, L., and Madden, S. (2021). Epoch-based commit and replication in distributed oltp databases. *Proceedings of VLDB Endowment*, 14(5):743–756.

Ozsu, M. T. and Valduriez, P. (2020). *Principles of Distributed Database Systems*. Springer, 4th edition.

Pierce, E. and Alvisi, L. (2000). A recipe for atomic semantics for byzantine quorum systems. Technical report, University of Texas at Austin, Department of Computer Sciences. Available in cs.utexas.edu/users/lorenzo/papers/byzatomic.ps. Accessed in Mar-13-2021.

Sadalage, P. and Fowler, M. (2013). *NoSQL distilled: a brief guide to the emerging world of polyglot persistence*. Pearson Education.

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

Souza, W. P. d. C. (2022). Desempenho e consistência de banco de dados chave-valor persistentes. Trabalho de Conclusão de Curso.

Vieira, G. M. and Buzato, L. E. (2010). Implementation of an object-oriented specification for active replication using consensus. *Technical Report IC-10–26*.