Danilo José Pereira Ferreira

Replicação síncrona para sistemas de armazenamento chave-valor em memória primária

Sorocaba, SP

28 de Abril de 2023

Danilo José Pereira Ferreira

Replicação síncrona para sistemas de armazenamento chave-valor em memória primária

Dissertação de mestrado apresentada ao Programa de Pós-Graduação em Ciência da Computação (PPGCC-So) da Universidade Federal de São Carlos como parte dos requisitos exigidos para a obtenção do título de Mestre em Ciência da Computação. Linha de pesquisa: 2.

Universidade Federal de São Carlos – UFSCar

Centro de Ciências em Gestão e Tecnologia – CCGT

Programa de Pós-Graduação em Ciência da Computação – PPGCC-So

Orientador: Prof. Dr. Gustavo Maciel Dias Vieira

Coorientador: Profa. Dra. Sahudy Montenegro González

Sorocaba, SP

28 de Abril de 2023

Ferreira, Danilo José Pereira

Replicação síncrona para sistemas de armazenamento chave-valor em memória primária / Danilo José Pereira Ferreira -- 2023.
113f.

Dissertação (Mestrado) - Universidade Federal de São Carlos, campus Sorocaba, Sorocaba
Orientador (a): Gustavo Maciel Dias Vieira
Banca Examinadora: Gustavo Maciel Dias Vieira, Angelo Roncalli Alencar Brayner, José de Oliveira Guimarães
Bibliografia

1. Replicação síncrona. 2. Banco de dados em memória. 3. Replicação descentralizada. I. Ferreira, Danilo José Pereira. II. Título.

Ficha catalográfica desenvolvida pela Secretaria Geral de Informática (SIn)

DADOS FORNECIDOS PELO AUTOR

Bibliotecário responsável: Maria Aparecida de Lourdes Mariano - CRB/8 6979

UNIVERSIDADE FEDERAL DE SÃO CARLOS

Centro de Ciências em Gestão e Tecnologia
Programa de Pós-Graduação em Ciência da Computação

Folha de Aprovação

Defesa de Dissertação de Mestrado do candidato Danilo José Pereira Ferreira, realizada em 28/04/2023.

Comissão Julgadora:

Prof. Dr. Gustavo Maciel Dias Vieira (UFSCar)

Prof. Dr. Angelo Roncalli Alencar Brayner (UFC)

Prof. Dr. José de Oliveira Guimarães (UFSCar)

O Relatório de Defesa assinado pelos membros da Comissão Julgadora encontra-se arquivado junto ao Programa de Pós-Graduação em Ciência da Computação.

Agradecimentos

Agradeço a todas as pessoas envolvidas na produção deste artigo, em especial ao meu orientador Prof. Dr. Gustavo Maciel Dias Vieira e à minha co-orientadora Profa. Dra. Sahudy Montenegro González, que me guiaram e me inspiraram ao longo do caminho. Seus conselhos, incentivos e apoio foram fundamentais para que eu pudesse alcançar este nível de conhecimento.

Também, agradeço ao discente Willian Pereira da Cruz Souza, que colaborou com este projeto como parte essencial do *benchmarking* realizado.

Por fim, quero agradecer a minha família, em especial a minha mãe e esposa, por todo o apoio e motivação que me deram ao longo desta jornada. Suas palavras de encorajamento e compreensão foram cruciais para que eu pudesse manter o foco e a determinação para continuar nos estudos.

Resumo

A partir de desafios de consistência e replicação em aberto no Redis, o presente trabalho aborda o uso de replicação síncrona como uma estratégia para fornecer ao banco de dados uma consistência forte de dados, com opção de persistência de dados em disco e suporte a recuperação. Além disto, é possível por meio deste trabalho compreender de maneira geral o custo de tal consistência em prol do desempenho.

Desenvolveu-se neste trabalho um *proxy* utilizando o *framework* de replicação síncrona Treplica que implementa o algoritmo de consenso Paxos e dá ao Redis as propriedades de replicação de dados no modelo *P2P*, consistência forte em ambiente onde processos podem falhar e se recuperar. Foi realizado um *benchmarking* comparando o sistema proposto com uma aplicação que utiliza o modelo de replicação e recuperação a falhas nativo do Redis, com o objetivo de compreender o *trade-off* entre consistência e desempenho do sistema. Obervamos que a troca entre consistência relaxada e consistência forte nesta aplicação reduz o desempenho entre 8,4 a 12,4 vezes, conforme esperado, mas ainda entregando uma alta taxa de escrita, entre 32,82 a 213,92 transações por segundo nos testes realizados.

Palavras-chaves: Replicação Síncrona. Redis. Treplica. Paxos. Consistência. Desempenho. Banco de dados. Peer-to-peer. P2P.

Abstract

In response to consistency and replication challenges found in Redis, this paper addresses the use of synchronous replication as a strategy to provide fully data consistency to a database, with the option of on disk data persistence and recovery support. In addition, this work sheds some light on the trade off between consistency and performance in replicated in-memory databases.

In this work, a proxy was developed using the Treplica synchronous replication framework, which implements the Paxos consensus algorithm and gives Redis the properties of data replication in a P2P model, data consistency, and supports the fail-recovery failure model. A benchmark was carried out comparing the proposed system with an application that uses Redis's native replication and failure recovery model, with the aim of understanding the trade-off between consistency and system performance. We have observed that the trade-off between eventual data consistency and fully data consistency in this application reduces performance by 8.4 to 12.4 times, as expected, but still delivers a high write rate, between 32.82 to 213.92 transactions per second in the tests performed.

Key-words: Synchronous Replication. Redis. Teplica. Paxos. Consistency. Performance. Database. Peer-to-peer. P2P.

Lista de ilustrações

Lista de tabelas

Lista de abreviaturas e siglas

NoSQL	*Not Only SQL*
SGBD	Sistema de Gerenciamento de Bancos de Dados
RAM	Memória de Acesso Aleatório (do inglês *Random Access Memory*)
E/S	Entrada e Saída
P2P	Par a par, descentralizado (do inglês *Peer-to-peer*)
P/S	Primário/Secundário
KVS	*Key-Value Store*
t/s	Transações por segundo
XML	*Extensible Markup Language*
JSON	*Javascript Object Notation*
DBMS	*Database Management System*
ACID	Atomicidade, Consistência, Isolamento e Durabilidade
BASE	Basicamente Disponível, estado volátil e eventualmente consistente (do inglês *Basically Available, Soft state and Eventually consistent*)
CAP	Consistente, Disponível e Tolerante a Partição (do inglês *Consistence, Avaiiability and Partition Tolerance*
BSON	*Binary JSON*
AOF	*Append on file*
TPC-W	Transaction Processing Performance Council - Web
UDP	*User Datagram Protocol*
API	*Application Programming Interface*
REST	*Representational State Transfer*
HTTP	*Hypertext Transfer Protocol*
CPU	Unidade central de processamento (do inglês *Central Processing Unit*)
IoT	Internet das coisas (do inglês *Internet of Things*)

Lista de símbolos

$\in$	Pertence
$\leftarrow$	Recebe
$\vee$	Ou
$\wedge$	E
$\emptyset$	Conjunto vazio
$=$	Igual
$\notin$	Não pertence
$\geq$	Maior ou igual
$<$	Menor
$+$	Soma
$\triangleright$	Comentário

Sumário

1 Introdução

Com a expansão da Ciência da Computação e o crescente acesso à internet pela população global, a demanda por novas soluções tecnológicas reflete em um maior uso de artefatos tecnológicos, físicos e virtuais, e por consequência, uma maior demanda para armazenamento de dados. Neste contexto, surgiram novas classes de bancos de dados para armazenar, processar e distribuir dados semi-estruturados ou não estruturados, chamados de Não Relacionais ou pela abreviação inglesa *NoSQL* (LI; MANOHARAN, 2013).

Dentre os modelos de Sistemas de Gerenciamento de Banco de Dados (SGBDs) não relacionais, aquele que apresenta premissas mais simples é o armazenamento chave-valor (SADALAGE; FOWLER, 2013), no qual atua como uma coleção de registros identificados por uma chave. Para inserir um novo dado, é informado ao sistema de armazenamento uma tupla com a chave única e o dado que se deseja armazenar. Para consumir ou buscar um dado, basta informar a chave associada (ANDERSON et al., 2010). Neste trabalho, optou-se por abordar este modelo, parte pela sua simplicidade em termos de estrutura de dados, parte pela sua variabilidade em termos de local de armazenamento e replicação, e parte pela sua abrangência, figurando entre os modelos mais utilizados (KAMARUZZAMAN, 2021).

Alguns dos sistemas de armazenamento, sobretudo do tipo chave-valor, realizam o armazenamento dos dados em memória primária, mais conhecida como Memória de Acesso Aleatório ou *RAM* (Do inglês *Random Access Memory*). Essa abordagem tem como principal objetivo garantir uma maior velocidade nas transações, composta tanto por operações de leitura quanto de escrita (KABAKUS; KARA, 2016), uma vez que o sistema não lida com E/S do disco, atendendo de maneira mais adequada aplicações com um alto número de requisições, como jogos *online*, redes sociais, *streaming*, entre outros (ATIKOGLU et al., 2012).

Um desafio presente nos SGBDs que operam seus dados em *RAM*, e que influencia na consistência dos dados, é a durabilidade destes, uma vez que estes estão sujeitos a deleção espontânea, caso a aplicação tenha uma falha ou o servidor seja desligado (TAN et al., 2015). Dentre as abordagens que visam sanar tais problemas, podem ser encontradas na literatura:

i Escrita de arquivos de registros de *logs*, utilizado para restaurar os dados após uma falha e recuperação do sistema (REDIS, 2020b).

ii Bibliotecas, que realizam transações atômicas, inserindo as tuplas tanto em memória e as replicando em disco (PAKSULA, 2010b).

iii Paralelização de dois sistemas de armazenamento de dados, sendo um em memória primária e outro em disco (MA; YANG, 2016).

Tais abordagens não só complementam o arcabouço teórico deste projeto bem como servem de inspiração para a aplicação de abordagens de tolerância a falhas e recuperação, propostas neste trabalho.

É importante destacar que a replicação distribuída dos dados em múltiplos nós também é uma abordagem que pode ampliar a consistência dos dados, sobretudo para as falhas citadas de sistemas que operam em memória principal. 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. Dentro desta abordagem, destacam-se os modelos *peer-to-peer* e primário/secundário. O modelo *peer-to-peer* é também conhecido pelo acrônimo *P2P*. Neste, cada réplica (nó) do banco de dados opera como uma coleção de repositórios locais autônomos que interagem entre si em um estilo ponto a ponto. O modelo primário/secundário (P/S), no qual um determinado nó é responsável por todas as escritas de dados, e as replica para os demais, que por sua vez podem atuar nas tarefas de leitura de dados (GRIBBLE et al., 2001).

Dentre os modelos de replicação apresentados, a grande maioria dos desenvolvedores dos SGBDs optam pelo P/S. Esta decisão ocorre, em grande parte, devido à complexidade de resolução de conflitos de escrita em modelos descentralizados (BONIFATI et al., 2008). O modelo P/S apresenta uma abordagem pessimista quanto a conflitos de escrita, uma vez que só um único nó é responsável pela escrita, impedindo a existência de conflitos (BONIFATI et al., 2008). Já no modelo descentralizado, a resolução de conflitos de escrita é complexa. Uma vez que ocorre uma falha de escrita, como por exemplo se dois nós editam simultaneamente o mesmo registro, a resolução depende de intervenções humanas, da escolha arbitrária do dado um nó, ou ainda de uma solução de domínio específico. De tal maneira, os desenvolvedores das aplicações precisam desenvolver suas próprias estratégias de resolução de conflitos, não triviais em termos de tolerância a falhas (ELMASRI; NAVATHE, 2005).

Entretanto, a consistência de dados em sistemas de replicação descentralizados já vem sendo estudada na área de algoritmos distribuídos no decorrer das últimas décadas (SCHNEIDER, 1990), com tecnologias promissoras para resolver conflitos, principalmente de escrita. Uma das abordagens apresentadas na literatura para a resolução de tais conflitos é a replicação síncrona. Na replicação síncrona todos os nós possuem o mesmo estado inicial e se comportam como máquinas de estado deterministas, de tal maneira que se são submetidos a uma mesma sequência de eventos determinísticos, devem permanecer iguais. Sendo assim, na replicação síncrona, os eventos são ordenados e replicados para os outros nós que por sua vez devem aplicar estes eventos seguindo uma ordenação prévia.

Uma das tecnologias que utiliza tal abordagem de replicação é o *framework* Treplica (VIEIRA; BUZATO, 2010), que aplica algoritmos de consenso para ordenar as transações e sincronizar dados entre as réplicas, garantindo assim a replicação dos dados livre de conflitos de escrita em todos os nós da rede. O sistema aborda ainda o armazenamento de dados tanto em memória primária quanto em disco, com uma abordagem de recuperação a falhas, de forma a garantir a consistência dos dados mesmo no caso de falhas de um ou mais nós.

A consistência dos dados é uma das preocupações de todos os SGBDs replicados. Se por um lado a replicação aumenta a disponibilidade de um sistema, também torna maior o desafio de manter os dados consistentes. Um dos SGBDs desenvolvidos com o objetivo de prover uma maior disponibilidade e garantir um alto desempenho é o Redis, um sistema de gerenciamento de dados do tipo chave-valor (Também conhecido pelo acrônimo inglês *KVS*), que armazena seus dados em memória primária, com replicação P/S. Uma das características do Redis é que a comunidade de desenvolvedores pode criar "extensões" para o mesmo, chamadas de *"proxy"* (pl. *"proxies"*) (REDIS, 2020c). Estes por sua vez podem manipular as características de funcionamento do SGBD, alterar como comandos primitivas operam e implementar novos comandos e políticas, como por exemplo alterar a arquitetura de replicação de dados.

Este trabalho aborda o uso de replicação síncrona em SGBDs distribuídos que operam seus dados em *RAM*, objetivando aumentar a consistência dos dados. Esta proposta pode ser aplicada a qualquer SGBD com tal característica, porém, optou-se por utilizar, neste trabalho, o Redis. Esta decisão está pautada nos seguintes pontos:

i o *framework* Treplica atualmente opera em modelo descentralizado e em memória primária, tendo maior compatibilidade com sistemas que possuem estas características (VIEIRA; BUZATO, 2010).

ii a estrutura *KVS* é inerentemente simples, com primitivas facilmente implementáveis (BONIFATI et al., 2008).

iii o Redis é um dos cinco SGBDs mais utilizados, e o mais utilizado da classe *KVS* (KAMARUZZAMAN, 2021).

iv o Redis é um *software* de código livre, com ampla documentação, comunidade de desenvolvedores ativa, e possui políticas e modelos para a criação de *proxy* (MEHTA, 2020).

Neste trabalho desenvolveu-se um *proxy*, também mencionado como aplicação, denominado Treplica-Redis, que implementa replicação síncrona por meio do *framework* Treplica. Este *proxy* é composto por três entidades (ou camadas): [i] Manager, [ii] Redis, e [iii] Treplica, sendo a primeira camada a responsável por oferecer uma *API* do tipo *REST* pela qual

são acionadas os comandos do Redis (ao invés de conexão direta ou o uso de algum outro cliente). Esta camada também é responsável pelo gerenciamento e interconexão das demais camadas que compõem a aplicação.

Com a utilização deste *proxy*, foi possível adicionar uma arquitetura de replicação descentralizada ao SGBD, bem como um modelo de recuperação de falhas que garante a consistência dos dados. O objetivo é compreender o *trade-off* entre desempenho e consistência ao utilizar um modelo de replicação síncrona. Além do desenvolvimento do Treplica-Redis em si, foram realizados testes de benchmarking com uma aplicação derivada deste, que não utilizava o *framework* Treplica e mantinha o modelo de replicação nativo do Redis. Conhecido como Redis-Sentinel, o modelo replicação nativo do Redis, implementa replicação de dados no modelo P/S com protocolos de *failover* (CARLSON, 2013).

Como resultados deste trabalho, foi possível observar e quantificar o *trade-off* consistência versus desempenho, que realiza de 8,4 a 12,4 vezes menos t/s se comparado ao uso do Redis com replicação nativa. Porém, a principal contribuição deste trabalho foi quantificar esta troca, de maneira que o desenvolvedor tenha melhores condições de realizar a escolha do seu modelo de replicação, em detrimento da consistência e desempenho que quer fornecer à aplicação.

Uma outra contribuição deste trabalho é que o software Treplica-Redis é operacional e pronto para o uso, e permite ao desenvolvedor aplicar tanto a consistência forte ou uma consistência relaxada[1]. Também, implementa um modelo modular, em que qualquer outro *framework* de replicação ou SGBD pode ser implementado com pouco esforço de desenvolvimento.

Ao leitor será apresentado o arcabouço teórico acerca de Sistemas distribuídos e Bancos de dados relevantes para este trabalho, no Capítulo 2. Então, os Capítulos 3 e 4 apresentam um aprofundamento teórico nos sistemas Redis e Treplica, respectivamente, como peças fundamentais para o desenvolvimento deste trabalho. Na sequência, o software resultante deste trabalho, Treplica-Redis, é minuciosamente detalhado no Capítulo 5, seguido dos testes realizados descritos no Capítulo 6, e, por fim, os resultados obtidos são discorridos no derradeiro Capítulo 7.

[1] A consistência forte é o modelo de atuação padrão do *proxy*, na qual as transações são processadas pelo *framework* Treplica e foi o modelo utilizado nos testes. Já a consistência relaxada implica apenas no uso Redis, contornando o Treplica, e no contexto do Treplica-Redis pode ser utilizado somente para leitura. Detalhes sobre os modelos de transação são providos no capítulo 5

2 Revisão Bibliográfica

As evoluções tecnológicas demandaram modelos de armazenamento de dados com maior disponibilidade e maior compatibilidade entre os dados armazenados e as estruturas da memória principal, sendo cunhado em 2009 o termo *NoSQL* para representar os bancos não relacionais (SADALAGE; FOWLER, 2013). Dentre este novo grupo, um modelo de armazenamento se destaca pela simplicidade de sua estrutura: o chave-valor. A simplicidade é um dos fatores que os fazem promissores no armazenamento em memória primária (PAKSULA, 2010b). Os *NoSQL* adotam estratégias de armazenamento distribuídos, seja no modelo P/S ou no modelo *P2P*. Porém, a consistência dos dados é um desafio nos modelos em memória primária e *P2P* (TAN et al., 2015; BONIFATI et al., 2008).

O problema da consistência dos dados na área de sistemas distribuídos já é discutida desde a década de 80 (LAMPORT, 1985; PIERCE; ALVISI, 2000), e mais recentemente na área de banco de dados (BREWER, 2000). Uma das soluções para este problema é a replicação síncrona (SCHNEIDER, 1990). A fim de garantir a consistência dos dados em sistemas distribuídos, o *framework* Tréplica (VIEIRA; BUZATO, 2010) foi desenvolvido para simplificar a aplicação da replicação síncrona e garantir a consistência de dados entre nós.

As próximas seções tem como objetivo apresentar alguns dos conceitos das área relevantes a este trabalho. Será apresentado uma breve descrição da origem e uso dos bancos de dados não relacionais, com aprofundamento no modelo chave-valor e nos modelos de armazenamento em memória primária. Também, os tipos de replicação de dados mais comuns para bancos de dados *NoSQL*. Na sequência será abordado a consistência dos dados e resolução de conflitos, bem como uma introdução a alguns problemas frequentemente encontrados na literatura. Por fim, apresenta-se a abordagem de replicação síncrona, como uma das possíveis soluções para garantir a consistência dos dados mesmo na presença de conflitos.

2.1 Bancos de dados não relacionais

Os bancos de dados não relacionais, surgiram em meados dos anos 2000, motivados por dois importantes fatores. O primeiro é a necessidade de suportar aplicações que precisavam lidar com grandes volumes de dados, consumindo assim mais recursos computacionais até um ponto em que a escala vertical não era possível e/ou economicamente viável. Assim, havia uma demanda de escala horizontal porém os sistema tradicionais não conseguiam atender de maneira eficiente (STONEBRAKER et al., 2010). O segundo fator está relacionado à necessidade de lidar com estrutura de dados que não se encaixavam no modelo

relacional, como dados semiestruturados e não estruturados, como XML e JSON (HAN et al., 2011). Tais estruturas podem ser manipuladas diretamente em memória primária sem a necessidade de serialização e conversão, melhorando o desempenho e tornando mais simples o desenvolvimento das aplicações.

Diferente dos Sistemas de Gerenciamentos de Bancos de Dados relacionais, com esquemas bem definidos, os sistemas não relacionais são divididos em quatro grupos quanto ao seu modelo de armazenamento de dados, sendo:

i o modelo orientado a documentos, em que se armazena arquivos com estruturas análogas a um objeto *JSON*.

ii orientado a coluna, em que os dados são gravados por coluna ao invés de linhas, aportando um maior desempenho em leituras específicas.

iii orientado a grafos, que armazena dados como nós e arestas interligados entre si.

iv armazenamento chave-valor, que armazena um dado qualquer (valor) e este é vinculado a uma chave única.

Conforme citado no Capítulo 1, o modelo utilizado neste trabalho é o *KVS*, especificamente com o SGBD Redis, que opera os dados em memória principal é uma das grandes referências globais em SGBD com alta disponibilidade e desempenho, por meio de um afrouxamento da consistência dos dados. O teorema CAP, proposto por Eric Brewer (BREWER, 2000), afirma que é impossível para um sistema de banco de dados *NoSQL* fornecer simultaneamente o máximo em consistência (C), disponibilidade (A) e tolerância a partições de rede (P). Tais SGBDs devem escolher entre uma consistência forte ou alta disponibilidade.

Se um sistema opta por uma alta consistência dos dados, ele deve sacrificar parte da sua disponibilidade, sobretudo em momentos de falhas (em que um nó não está operando adequadamente) ou quando os dados estão sendo replicados e o estado de uma determinada réplica não é o mais atualizado, ou mesmo sacrificar a tolerância a partição quando existem falhas da rede. Por outro lado, ao ter uma alta a disponibilidade, pode não ser capaz de garantir consistência forte em todos os momentos, e nestes casos, é comum ver abordagens de consistência eventual, em que algumas réplicas podem ter um banco de dados não consistente com o último estado da aplicação.

A consistência dos dados de um Sistema de Gerenciamento de Banco de Dados (SGBDs ou em inglês DBMS) pode ser definida por meio de 4 propriedades denominadas ACID (Acrônimo para Atomicidade, Consistência, Isolamento e Durabilidade):

1. **Atomicidade**: cada transação é executada como uma única unidade indivisível. Se falhar, todas as alterações são desfeitas para manter o banco de dados consistente.

2. **Consistência**: cada transação deve levar o banco de dados de um estado válido para outro estado válido, mesmo em caso de falhas ou conflitos de transações, inclusive revertendo a transação se necessário.

3. **Isolamento**: cada transação é executada em ambiente isolado de outras transações. Alterações de outras transações são invisíveis até que concluam.

4. **Durabilidade**: alterações feitas por transação bem-sucedida são permanentemente salvas no banco de dados, mesmo em caso de falhas do sistema.

Ao garantir estas 4 propriedades, podemos afirmar que um SGBD mantém a consistência dos dados, porém, os SGBDs *NoSQL* foram idealizados com foco na escalabilidade e na disponibilidade, permitindo que os usuários realizem transações em um grande volume de dados com alta velocidade e menor custo. Em alguns casos, podem sacrificar a consistência em favor de maior disponibilidade e velocidade de leitura/gravação de dados (KUMAR; PABREJA, 2013).

Assim, em 2008, desenvolveu-se o conceito de Propriedades BASE (basicamente disponível, estado volátil e eventualmente consistente, do acrônimo inglês *Basically Available, Soft state and Eventually consistent*) (PRITCHETT, 2008) como uma alternativa às propriedades ACID compatível com o modelo *NoSQL*. as propriedades BASE são uma resposta ao compromisso entre consistência e disponibilidade, no qual, ao invés de priorizar a consistência, como no modelo ACID, permite-se uma consistência eventual. A consistência eventual é uma técnica que garante que as atualizações de dados sejam propagadas eventualmente para todas as réplicas, sem a necessidade imediata de uma confirmação. Assim, diferentes réplicas do banco de dados podem divergir temporariamente, mas eventualmente convergirão para um estado consistente (VOGELS, 2008).

2.2 Arquiteturas de distribuição de dados

Dado um SGBD distribuído, portanto tolerante à partição, a distribuição em banco de dados é a área da computação que refere-se à maneira em que os dados estão espalhados ou organizados em diferentes réplicas. Dentre as estratégias de distribuição, destaca-se a replicação e fragmentação.

2.2.1 Replicação

A replicação de dados em SGBDs é o processo de criar e manter cópias dos dados em diferentes instâncias, e geralmente em diferentes localidades físicas, visando garantir a disponibilidade destes dados, tolerância a falhas e a escalabilidade do sistema, sobretudo para atividades de computação distribuída (OZSU; VALDURIEZ, 2011).

Existem dois modelos de replicação utilizados em SGBDs *NoSQL* (SADALAGE; FOWLER, 2013):

i Primário/Secundário (ou P/S), no qual um nó é eleito como primário e apenas este pode lidar com escritas, e os demais são cópias autorizadas, que por sua vez sincronizam com o nó primário e podem lidar apenas com as leituras, vide Figura 1.

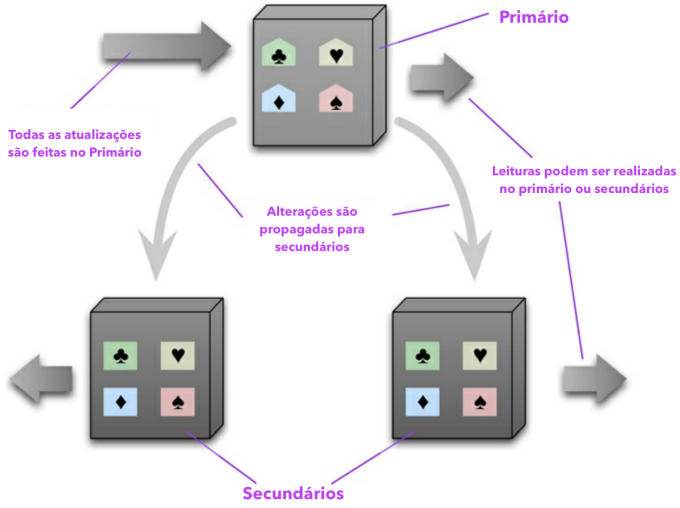

Figura 1 – Arquitetura de replicação Primário/Secundário. Tradução (SADALAGE; FOWLER, 2013)

ii Descentralizado (ou *P2P*), no qual qualquer nó da rede pode realizar escritas e leituras, e os mesmos coordenam-se para sincronizar a replicação de seus dados. vide Figura 2.

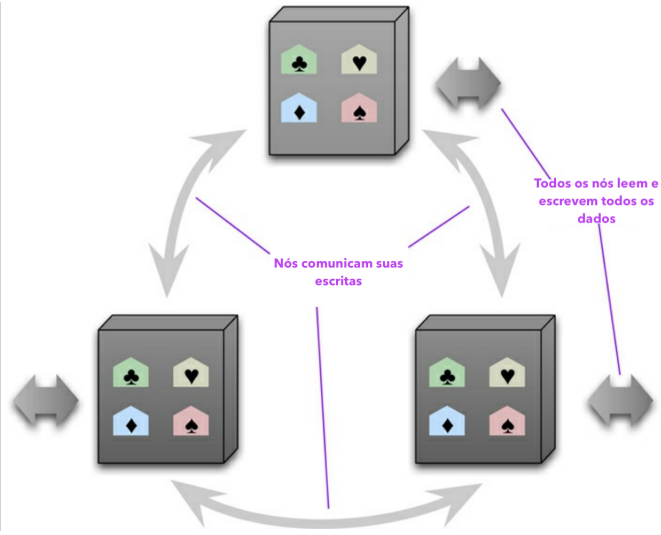

Figura 2 – Arquitetura de replicação descentralizada. Tradução (SADALAGE; FOWLER, 2013)

A abordagem *P2P* se destaca por oferecer às aplicações distribuídas o acesso e operação aos dados em qualquer nó (BONIFATI et al., 2008). Mesmo assim, a grande maioria dos SGBDs que oferecem suporte a replicação de dados o fazem no modelo P/S. Isto ocorre devido ao desafio de gerenciar conflitos e assim garantir a consistência dos dados. Este desafio 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).

2.2.2 Fragmentação

Um outro método de distribuição dos dados é a fragmentação (do inglês *sharding*), é um modelo em que os nós não são copias umas das outras. Neste, o conjunto de dados de um sistema é dividido em subconjuntos e distribuído nos diversos nós. Assim, cada nó terá um ou mais subconjunto de dados, conforme ilustrado na Figura 3.

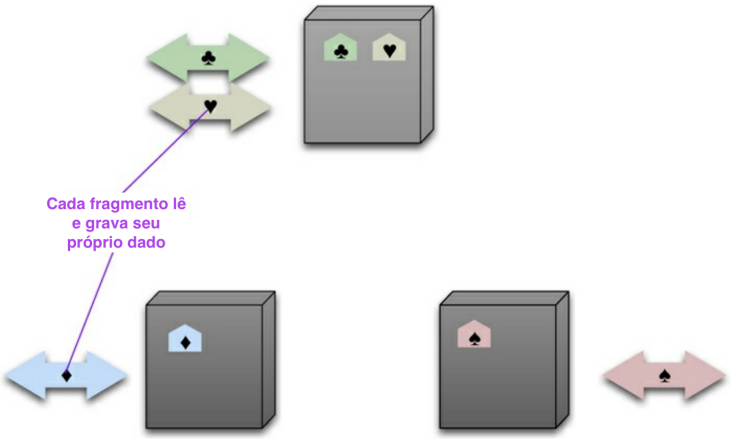

Figura 3 – Arquitetura de particionamento. Tradução (SADALAGE; FOWLER, 2013)

Enquanto no modelo de distribuição de replicação as réplicas possuem todos os dados, no modelo de distribuição fragmentado existe a necessidade de processos adicionais para identificar em qual nó está o subconjunto de dados que se quer operar. Por outro lado, distribui-se a responsabilidade da escrita para um determinado fragmento (para arquitetura P/S), de tal maneira que se um nó vem a falhar, apenas os fragmentos nestes nós ficam inacessíveis.

Ainda, é possível que um SGBD tenha a distribuição dos seus dados híbrido, mesclando fragmentação e distribuição. Assim, cada um dos subconjuntos de fracionamento possuam réplicas em um ou mais nós (BAGUI; NGUYEN, 2015), conforme apresentado na Figura 4. Cada subconjunto e suas réplicas podem operar tanto na arquitetura P/S, quanto na arquitetura *P2P*.

Neste trabalho levou-se em consideração apenas a estratégia de replicação dos dados, como uma maneira de garantir a consistência dos dados entre as réplicas.

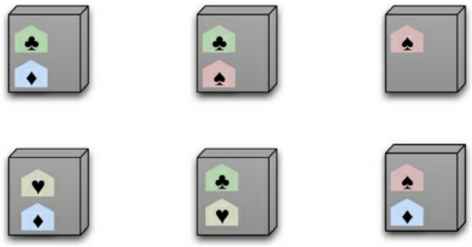

Figura 4 – Arquitetura de particionamento em modelo distribuído. Tradução (SADALAGE; FOWLER, 2013)

2.3 Consistência dos dados

A consistência dos dados sob ponto de vista de bancos de dados ocorre em dois aspectos. O primeiro, do ponto de visto de uma transação (consistência transacional), em que o conjuntos de operações de operações de uma transação leva o banco de dados de um estado consistente para outro estrado consistente. (SADALAGE; FOWLER, 2013).

2.3.1 Consistência transacional

Os SGBDs relacionais garantem a consistência dos dados por meio de transações que respeitam as propriedades ACID. A consistência transacional está sujeita a violação sempre que houver transações concorrentes. Já em caso de falhas, as transações devem ser desfeitas ou revertidas (ELMASRI; NAVATHE, 2005). Sem a garantia de mecanismos de controle de concorrência e de tolerância a falhas, as transações estão sujeitas às seguintes anomalias de consistência:

i **Leitura suja**: esta violação se dá quando ocorre a leitura de um dado de uma gravação ainda não confirmada, e a gravação sofre uma falha, logo a leitura trará um dado não existente e incorreto. Esta violação pode ocorrer em um registro seguro.

ii **Leitura não repetitiva**: ocorre quando uma transação realiza a leitura de um determinado valor ao menos duas vezes, e as leituras apresentam valores divergentes, por conta de gravações concorrentes. Esta violação pode ocorrer em registros seguros e regulares.

iii **Atualização perdida**: ocorre quando uma transação de leitura de um determinado valor inicia sem gravações paralelas, porém antes de seu término uma ou mais novas gravações são iniciadas e finalizadas. O valor retornado pode ser o valor anterior às gravações ou posterior às mesmas. Esta violação pode ocorrer em registros seguros e regulares.

Com base nestas, os SGBDs podem apresentar quatro níveis de isolamento (ELMASRI; NAVATHE, 2005): Leitura não confirmada (ou *Read Uncommitted*), que permite

todas as anomalias de consistência. Leitura confirmada (ou *Read Committed*) que não permite leituras sujas. Leitura repetível (ou *Repeatable read*) que não permite leituras sujas nem leitura não repetitivas. Por fim, Serializáveis (ou *serializable*) que não permite nenhuma das violações.

2.3.2 Consistência de replicação

Por outro lado, a consistência dos dados pode ainda ser avaliada do ponto de vista de replicação, no qual deve-se garantir que todas as réplicas são idênticas entre si. Os bancos de dados projetados para operar em rede caracterizam-se pelo uso das propriedades BASE (PRITCHETT, 2008), na qual afirma que até que os dados sejam replicados, o SGBD estará em um estado de consistência relaxada, em prol de prover uma maior disponibilidade do sistema. Esta troca é feita com base na suposição de que uma consistência eventual já é suficiente para grande parte dos serviços *online* (ANDERSON et al., 2010).

Observe que neste tópico a consistência de replicação não implica em um afrouxamento da consistência transacional, mas adiciona uma nova camada de complexidade, devendo garantir que os estados das réplicas são consistentes e iguais.

Ainda do ponto de vista de replicação, uma abordagem mais profunda de consistência é apresentada na área de sistemas distribuídos, afirmando que a consistência de um dado pode ser classificada em três níveis, sendo do menos para o mais confiável: seguro (*safe*), regular (*regular*) e atômico (*atomic*) (LAMPORT, 1985; PIERCE; ALVISI, 2000):

i Um registro é **seguro** se uma leitura, não concorrente a nenhuma gravação, retorna o valor da gravação mais recente, e uma leitura paralela a uma ou mais gravações deve retornar o valor da última gravação finalizada ou de uma das gravações concorrentes.

ii Um registro é **regular** se uma leitura, não concorrente a nenhuma gravação, retorna o valor da gravação mais recente, e uma leitura paralela a uma ou mais gravações deve retornar ou o valor anterior às gravações paralelas, ou o valor posterior às gravações paralelas, mas nunca um valor intermediário, e

iii Um registro é **atômico** se as leituras e gravações forem ordenadas totalmente, de forma que não haja sobreposição entre transações, e qualquer leitura sempre reflita ao último valor escrito.

Assim, tanto as anomalias tratadas nos níveis de isolamento, quando ocorre uma falha no controle de concorrência, quanto às anomalias de replicação, que podem ocorrer em registros seguros e regulares, representam a presença de dados inválidos ou divergentes entre as réplicas, conhecidos por conflitos de dados. Vale destacar que neste trabalho preocupa-se apenas com a consistência dos dados a nível da replicação.

2.4 Conflitos de dados

De maneira geral, o nível de consistência de um banco de dados está ligado à aplicação de técnicas que lidam com controle de concorrência e replicação de dados. Um conflito de dados ocorre, em geral, em duas ocasiões. A primeira é quando ocorre alguma falha, podendo tanto gerar registros inválidos em alguma réplica quanto impedir a replicação dos dados em um ou mais nós. Já a segunda ocorre quando quando duas ou mais transações tentam modificar o mesmo dado simultaneamente (OZSU; VALDURIEZ, 2011).

Na literatura, estes os conflitos de dados são divididos em duas classes (ELMASRI; NAVATHE, 2005):

i Conflito escrita-escrita: ocorre quando dois clientes tentam, simultaneamente, atualizar o mesmo dado, ocasionando uma atualização perdida.

ii Conflito escrita-leitura: ocorre quando há uma leitura concorrente a uma gravação, e portanto o dado adquirido contém partes incompletas ou desatualizadas, ou, quando um cliente lê um dado em um nó, e existe uma versão mais atualizada deste dado em outro nó que ainda não foi devidamente propagado, ocasionando uma leitura não repetitiva ou leitura suja.

Existem duas abordagens a fim de resolver tais conflitos (ELMASRI; NAVATHE, 2005). Enquanto a abordagem pessimista atua na prevenção da ocorrência de conflitos, criando estratégia que limitam a execução de transações concorrentes, a abordagem otimista pode permitir que transações conflitantes sejam executadas em paralelo, podendo comprometer a consistência dos dados, e caso ocorra, deve tentar medidas para resolvê-las (GRAY; REUTER, 1993). Dentre estas medidas, por exemplo, destaca-se abortar as transações envolvidas (*rollback*), reexecutar as transações utilizado algum algoritmo de consenso, eleger uma transação como válida e abortar as demais, mesclar os resultados (geralmente utilizado para atualização de dados do tipo numéricos), resolução manual, etc.

A replicação dos dados em um modelo P/S, na qual um único nó é utilizado como destino para todas as gravações de dados, torna simples manter a consistência dos dados, uma vez que é possível controlar a concorrência (já que todas as transações que possuem escrita são processadas por um único nó) e assim aplicar uma abordagem pessimista de resolução de conflitos. Porém, em modelo de replicação descentralizado, no qual qualquer nó pode aplicar as atualizações nos dados, não há uma maneira trivial de controlar a concorrência, e, garantir que um mesmo dado não será divergente nos demais nós.

Em geral, as estratégias para resolver este problema estão relacionadas ao uso de um quorum para tomar decisões, tal que estas decisões sejam consistentes em todos os nós da rede. Um dos modelos utilizado é o uso de algoritmos de consenso (e.g. Paxos

(LAMPORT, 1998a) e Raft (ONGARO; OUSTERHOUT, 2014)), no qual um conjunto de nós deve chegar a um acordo para controlar a concorrência e este acordo será eventualmente replicado e aplicado para os nós que não participaram do consenso.

2.5 Replicação síncrona

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 que impede que os conflitos ocorram garantindo o isolamento das transações.

O *framework* Tréplica (VIEIRA; BUZATO, 2010), por exemplo, 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. Este aplica o algoritmo de ordenação e consenso Paxos (LAMPORT, 1998b), que garante que as transações sejam totalmente ordenadas por um quórum de nós.

O *framework* apresenta uma abordagem pessimista para manter a consistência dos dados, subjugando eventuais conflitos por meio da ordenação total de todas as transações, o que, na prática, faz com que as transações sejam não concorrentes. Por outro lado, o desempenho do sistema pode ser parcialmente sacrificado por necessitar de pares de troca de mensagens entre os nós a cada nova transação, garantido pelo funcionamento do algoritmo Paxos (LAMPORT, 1998b).

O Capítulo 4 busca expandir a arquitetura do Treplica e suas premissas sob a ótica da aplicação desenvolvida neste trabalho.

2.6 Bancos da dados chave-valor

No modelo de armazenamento chave-valor, um valor é associado a uma chave única, e este valor pode ser recuperado posteriormente informando-a. Este modelo também é conhecido como endereçamento *hash* (MORRIS, 1968; PAKSULA, 2010b). Adicionalmente, alguns SGBDs chave-valor, como é o exemplo do Redis, permitem operações de intervalo, comparação, união e interseção, baseados na chave (SADALAGE; FOWLER, 2013).

Segundo Sadalage, os SGBDs chave-valor destacam-se por sua simplicidade e ao mesmo tempo flexibilidade para armazenamento dos dados, que são livres de esquemas. Ainda, são capazes de prover alta velocidade na recuperação de dados, escalabilidade

horizontal, o que significa que novos nós podem ser adicionados para aumentar a capacidade de armazenamento e processamento. No entanto, destaca-se também algumas das limitações dos *KVS*, como a falta de recursos de consulta sofisticados (por exemplo buscar uma propriedade de um valor) e a necessidade de escrever código personalizado para lidar com a consistência dos dados, pois geralmente não possuem transações ACID.

Desta maneira, o modelo de bancos de dados *NoSQL* chave-valor é mais utilizado para armazenar metadados, registros, filas de mensagens, armazenamento em *cache* e serviço de dados pré-computados (SADALAGE; FOWLER, 2013). Esta característica torna o chave-valor uma opção atraente para sistemas que buscam alto desempenho na gravação e leitura dos dados. Combinado a isto, uma abordagem que vem se tornando cada vez mais comum é o armazenamento em memória primária.

2.7 Armazenamento em memória primária

Na década de 90, quando a programação orientada a objetos tornou-se popular, houve uma grande interesse em bancos de dados com orientação a objetos, pautado na complexidade do mapeamento de estruturas de dados na memória para tabelas relacionais. A ideia era evitar essa complexidade com um modelo que gerencia automaticamente o armazenamento de estruturas em memória primária e em disco, sendo este um dos primeiros motivadores a armazenar dados em memória primária (SADALAGE; FOWLER, 2013).

Outro motivador deste modelo de armazenamento de dados é que manter os dados na memória oferece uma vantagem de desempenho, uma vez que não há E/S de disco quando um evento é processado (SADALAGE; FOWLER, 2013; KABAKUS; KARA, 2016; TAN et al., 2015). Por outro lado, uma desvantagem é que em caso de erros não é possível realizar a reversão para um estado anterior. Portanto, é necessário ter um sistema próprio de reversão ou garantir uma validação completa antes de começar a aplicar alterações. Um problema comum também está relacionado à perda de dados em caso de falha de energia, já que a memória primária é apagada quando o sistema é desligado (SADALAGE; FOWLER, 2013). Este foi um dos motivadores para a realização deste trabalho pelo autor, quando em um momento de falha, uma base de dados tornou-se inconsistente. Assim, em busca de soluções para este caso, observou-se na literatura os efeitos causados pela aplicação de uma abordagem que garante alta disponibilidade em detrimento de uma consistência forte em sistemas de armazenamento de dados em memória primaria, e foi entendido este como um desafio em aberto com possibilidade de uma contribuição cientifica relevante para a área de estudo em questão.

3 Redis

Redis é um SGBD de código aberto que utiliza o modelo de chave-valor para armazenar dados. Ele é amplamente utilizado porque oferece acesso rápido aos dados e pode ser usado em diversas situações.

Desenvolvido por Salvatore Sanfilippo em 2009 e escrito em linguagem de programação C, é reconhecido por seu alto desempenho e capacidade de crescer para atender a aplicações que precisam de acesso rápido a grandes conjuntos de dados. Ele é especialmente útil em casos em que é necessário lidar com altas taxas de escrita e leitura de dados, como em aplicações em tempo real (REDIS, 2020a). Segundo Lavin (LAVIN, 2012) o software é bem versátil e pode ser usado para vários propósitos, como:

- Armazenar dados em cache para melhorar o desempenho de aplicações
- Armazenar dados de sessão para aplicações web
- Gerenciar filas de trabalho em aplicações distribuídas
- Armazenar dados de ranking em jogos online ou aplicativos de mídia social
- Armazenar dados de estatísticas em aplicações de análise de dados

Como um sistema de armazenamento distribuído que visa garantir a alta disponibilidade para leituras, alto desempenho e tolerância a falhas, o Redis conta com uma aplicação especifica para detectar falhas e coordenar *failover*. Ainda, conta com modelo de replicação de dados nativo Primário/Secundário, e suporte a fragmentação dos dados em diversos servidores. Do ponto de vista de armazenamento e operação, o Redis suporta diferentes tipos de dados, como textos, listas, conjuntos e mapas (MACEDO; CALHEIROS, 2013). Tais propriedades serão apresentadas nas seções deste Capítulo.

3.1 Tipos de dados

O Redis armazena seus dados em tuplas, compostas por uma chave única e um valor. O Redis é capaz de lidar com cinco estruturas de dados (CARLSON, 2013), sendo: [i] `STRING`: uma sequência de *bytes* (binário) que pode possuir até 512 *megabytes* de tamanho. [ii] `LIST`: uma lista ligada e ordenada de dados do tipo `STRING`. [iii] `SET`: uma coleção desordenada de dados do tipo `STRING`. [iv] `HASH`: um mapeamento de chaves para valores, sendo estes valores `STRING`. Por fim, [v] `ZSET`: mapeamento de dados do tipo `STRING` para decimal, ordenados por este.

Para todas as estruturas de dados, três primitivas são aplicadas:

- `GET`: busca um dado armazenado associado a chave indicada.
- `SET`: armazena uma nova tupla dado chave e valor.
- `DEL`: remove uma tupla dado uma chave.

Apesar de já possuir diversos modelos de estruturas de dados e mais de 250 comandos primitivos, que vão de operações exclusivas para cada tipo de dado armazenado até comandos de gerenciamento do SGBD, as capacidades do Redis podem ser estendidas por meio de bibliotecas desenvolvidas pela comunidade. Conhecidos na terminologia como *proxy* (REDIS, 2020c).

3.2 Proxies

Os *proxies* do Redis tem como principal objetivo ampliar as capacidades do SGBD, provendo desde novas estruturas de dados até modelos de replicação e gerenciamento do SGBD. Estas extensões podem ser implementadas como novas camadas sobre a API, ou bibliotecas de nível superior que implementam alterações a nível de aplicação.

Dos 174 *proxies*, extraídos principalmente do próprio site mantenedor do Redis (REDIS, 2020a), 27 foram selecionados como *proxies* que de alguma maneira implementam alterações na consistência e/ou conflitos de escrita e leitura no Redis, e que possam ser relevantes para este trabalho. Dentre estes agrupa-se por funcionalidade os *proxies* que:

i permitem transações concorrentes (*Multi-thread*) (MEHTA, 2020; LABS, 2020; LIMITED, 2018).

ii permitem a criação de *cluster* descentralizado de nós primários (CODISLABS, 2020; HUAWEI, 2020; PROJECT, 2016; LABS, 2018; LABS, 2020; LIMITED, 2018; LABS, 2019).

iii permitem a criação de *cluster* descentralizado (LABS, 2018; LABS, 2020; LIMITED, 2018; LABS, 2019).

iv criam travas distribuídas (*distributed locks*) (CHEPRASOV, 2016; WUJUNWEI, 2020; LEIFER, 2011; WESTCOTT, 2020; LEIFER, 2018)

v replicam as escritas para arquivo local (*streaming*)(GUERREIRO, 2012a; BOTROS, 2021).

vi replicam as escritas para outros SGBDs (*streaming*) (KEWANG, 2010; PAKSULA, 2010a; IENAGA, 2016).

vii implementam replicação síncrona entre nós primários e secundários (GUERREIRO, 2012b).

viii criam fila de requisições assíncrona entre os nós (COMMUNITY, 2012; BOULTON, 2017; GEORGE, 2009; MARTENS, 2011; LIBRELIST.COM, 2009; CARLSON, 2015), e.

ix cria fila de ações assíncrona entre os nós com função de espera (CELERY, 2009; BANGERT, 2011; LEIFER, 2011; WESTCOTT, 2020).

Uma das contribuições deste trabalho, a ser citado no Capítulo 5 é o desenvolvimento de um *proxy* que permite a criação de *cluster* com replicação descentralizada de nós primários que previne a ocorrência de conflitos de escritas, ao mesmo tempo que replica as transações com operações de escrita para arquivo local e implementa técnicas de recuperação a falhas. Ressalta-se que dentre os *proxies* citados, todos implementam ao menos alguma das garantias similares às citadas, porém nenhum implementa todas elas.

3.3 Replicação de dados

Além dos protocolos de replicação fornecido por meio de *proxies*, como os citados na seção anterior, o Redis possui nativamente um modelo de replicação de dados P/S (MACEDO; CALHEIROS, 2013). Do ponto de vista de replicação, este funciona da seguinte maneira:

1. O Redis possui um servidor principal, conhecido como primário, responsável por receber e replicar todas as escritas de dados para os servidores secundários.

2. Os servidores secundários se sincronizam com o primário e replicam os dados.

3. Quando um cliente envia uma solicitação de escrita para o primário, este aplica a escrita em sua própria cópia do banco de dados e, em seguida, envia a escrita para todos os secundários.

4. Os secundários aplicam a escrita em suas próprias cópias do banco de dados e enviam uma confirmação de sucesso de volta para o primário.

A replicação é um recurso útil para garantir a alta disponibilidade dos dados para leitura e tolerância a falhas. No entanto, é importante lembrar que a replicação pode afetar o desempenho das escritas, pois em determinadas configurações o primário precisa aguardar a replicação e as confirmações de todos os secundários antes de confirmar uma escrita. Outras configurações porém, permitem que o Redis não aguarde o retorno

das réplicas, ganhando um maior desempenho e afrouxando a consistência dos dados (MACEDO; CALHEIROS, 2013).

Sendo assim, os clientes podem ler os dados de qualquer um dos servidores, incluindo o primário ou os secundários, porém a gravação é restrita ao primário. Isto faz com que do ponto de vista de leitura, a consistência dos dados seja relaxada, pois pode-se ler em um secundário uma versão do dado anterior ao primário, por exemplo (PAKSULA, 2010b).

Uma outra desvantagem do modelo P/S, do ponto de vista de gravação dos dados, é que se de um lado o modelo evita conflitos de escrita, por outro lado cria um gargalo, uma vez que, dentre todos os nós, apenas um nó pode gravar dados. Esta abordagem reduz a consistência dos dados principalmente em momentos de *failover*, no qual escritas podem não ser recepcionadas ou serem confirmadas pelo primário anterior, mas perdidas após a eleição de um novo.

A fim de reduzir os impactos deste modelo de replicação, o SGBD permite diversas configurações de replicação. Dentre elas a fragmentação de dados entre os diferentes nós da rede.

3.4 Fragmentação

A fragmentação é uma técnica utilizada para dividir os dados em vários fragmentos, e distribuí-los entre diferentes servidores. O objetivo é aumentar o desempenho e a escalabilidade de um sistema de banco de dados permitindo que os dados sejam armazenados e processados em paralelo em várias máquinas. Isso ajuda a gerenciar grandes conjuntos de dados de maneira mais eficiente e aumenta a capacidade de processamento do sistema. (KLEPPMANN, 2017)

No Redis, a fragmentação é realizado através da divisão do conjunto de chaves em vários intervalos, cada um dos quais é atribuído a um servidor ou máquina específica (MACEDO; CALHEIROS, 2013). Quando um cliente envia uma solicitação de leitura ou escrita, o servidor determina em qual intervalo a chave está incluída e encaminha a solicitação para o servidor ou máquina responsável pelo intervalo em questão.

Existem duas opções de configuração de fragmentação: manual e automatizada. Na configuração manual, o administrador do sistema define os intervalos de dados e as máquinas responsáveis por armazená-los. Já na configuração automatizada, o próprio SGBD se encarrega de dividir os intervalos e distribuir os dados automaticamente, podendo este contar com o suporte dinâmico, onde os intervalos são redimensionados de acordo com as mudanças no conjunto de dados.

Apesar de ser uma técnica útil para melhorar o desempenho e a escalabilidade do Redis, a fragmentação pode aumentar a complexidade do sistema e pode requerer

um gerenciamento cuidadoso para garantir que os dados estejam sempre sincronizados entre os servidores. Seja com o objetivo de garantir que os dados estão consistentes entre as réplicas, seja para garantir que a distribuição foi feita de forma equilibrada entre os servidores, uma vez que pode exigir ajustes constantes da configuração de fragmentação conforme as mudanças do conjunto de dados e das necessidades de acesso.

É importante ressaltar que o *proxy* desenvolvido neste trabalho não aplica nenhum método de fragmentação de dados, dedicando-se exclusivamente a estratégias de replicação, a ser detalhada no Capítulo 4, visando garantir a consistência dos dados e compreender o *trade-off* deste com disponibilidade.

3.5 Consistência, disponibilidade e tolerância a partição

A consistência, segundo a interpretação de Sadalage (SADALAGE; FOWLER, 2013) do teorema *CAP* (BREWER, 2000), refere-se à ideia de que todos os nós possuem a mesma versão dos dados. Assim, mesmo que o sistema esteja lidando com várias transações simultaneamente, todas as leituras subsequentes a uma escrita deverá retornar a versão mais atualizada dos dados.

A consistência é uma das principais preocupações em qualquer sistema de gerenciamento de banco de dados, e o Redis oferece algumas opções para garantir a consistência dos dados. Para Carlson (CARLSON, 2013) um dos métodos mais comuns é a persistência em disco, que permite que os dados sejam armazenados em um arquivo em disco de forma a serem mantidos mesmo após o processo Redis ser finalizado. Outra opção é a replicação de dados, que permite que os dados sejam replicados em vários servidores Redis para garantir a maior disponibilidade e tolerância a falhas.

Outro conceito intrinsecamente ligado à consistência é o conceito de disponibilidade, que refere-se a um sistema que consiga receber e responder de forma confiável as requisições dos clientes, mesmo que ocorra uma falha em um ou mais nós. Em geral, aplica-se técnicas de replicação e fragmentação de dados de maneira que se um nó falhar, outro nó possa assumir as leituras e escritas de dados (SADALAGE; FOWLER, 2013).

Do ponto de vista do Redis, este tem suporte à replicação P/S, na qual qualquer nó é capaz de atender leitura dos dados, porém a escrita pode ser realizadas apenas pelo nó primário. A propriedade de disponibilidade é afrouxada na ocorrência de falhas no nó primário, podendo, por exemplo, estar indisponível para escritas em caso de falhas.

Dentre as possíveis falhas que lidam com a consistência e disponibilidade, uma classe de falhas é tratada sob uma ótica distinta no teorema de Brewer (BREWER, 2000), a partir da propriedade *NoSQL* de tolerância a partição. Sadalage (SADALAGE; FOWLER, 2013) refere-se a tolerância a partição como a capacidade de um sistema distribuído de

funcionar mesmo quando ocorre uma falha de comunicação ou divisão da rede que conecta os diversos nós. Desta maneira, mesmo na presença de tais falhas o sistema deve ser capaz de operar.

A propriedade de tolerância a partição no Redis é herdada do modelo P/S e utiliza um protocolo de *failover*, gerenciado por uma aplicação externa, a fim de detectar e recuperar-se desta falha. Este componente externo foi lançado na segunda versão do Redis, e implementa uma abordagem de tolerância a falhas no SGBD.

3.6 Tolerância a falhas

A capacidade de tolerar falhas diz respeito a identificar e recuperar-se rapidamente delas, o que é sutilmente distinto da recuperação de conflitos, na qual o sistema utiliza estratégias para manter os dados consistentes. A tolerância a falhas no Redis pode se dar a nível do sistema, com abordagem de substituição de instâncias defeituosas (*failover*), ou a nível da réplica, com estratégias de recuperação que abordam a consistência dos dados (MACEDO; CALHEIROS, 2013).

No Redis, o *failover* é gerenciado em um componente chamado Sentinel. Este componente é projetado para identificar falhas no primário e coordenar um algoritmo de eleição de líder, auxiliando na garantia da disponibilidade. O Sentinel utiliza um sistema de quorum tanto para detectar falhas quanto na eleição de um novo líder. O quorum é formado por 50% + 1 réplicas, as quais devem entrar em consenso. Observe que entre a ocorrência da falha e a eleição de um novo primário, escritas podem ser rejeitadas (perdendo a garantia de disponibilidade) ou escritas podem ser confirmadas mas não replicadas (perdendo a garantia de consistência). Assim, torna-se essencial o uso de abordagens que lidem com tais problemas da ótica do SGBD.

Já a nível da réplica, o Redis possui duas estratégias de recuperação de dados após uma falha, o [i] *Append on file* (*AOF*), na qual as transações são armazenadas em um arquivo no disco e são reexecutadas em caso de recuperação de falhas, e/ou [ii] a criação de instantâneos, em que salva-se o estado do SGBD para um arquivo do disco, proporcionando um reinício mais rápido em caso de falhas.

Apesar dos grandes esforços para detectar e recuperar de falhas, vale destacar que o Redis não é projetado para ser um banco de dados persistente em primeiro lugar. Em vez disso, ele é projetado para ser usado como um sistema de armazenamento em memória com opção de persistência em disco (CHEN et al., 2016). Isto torna o Redis uma opção para aplicações que buscam alto desempenho ao invés de consistência. A partir desta constatação, tem-se um dos motivadores deste trabalho: é possível garantir uma alta consistência no Redis, e ainda assim se beneficiar de seu desempenho?

3.7 Desafios em aberto

Por ser um software de código aberto, e mantido por uma grande empresa, o Redis conta com diversas estratégias na qual o usuário pode, por meio de suas configurações, afrouxar a consistência em prol da disponibilidade ou vice-versa (MACEDO; CALHEIROS, 2013). Ainda assim, segundo Chen et al (CHEN et al., 2016) há portanto alguns casos em que o Redis pode perder dados, como:

1. **Quando o processo Redis é finalizado**: se o processo for finalizado de forma abrupta, os dados em memória podem ser perdidos. No entanto, os dados armazenados em disco serão mantidos e podem ser recuperados quando for reiniciado.

2. **Quando ocorrem falhas na persistência em disco**: em alguns casos, o processo de persistência em disco pode falhar, o que pode resultar na perda de alguns dados.

3. **Quando ocorrem problemas de sincronização de dados durante a replicação**: se a replicação de dados não estiver configurada corretamente ou se houver problemas de sincronização entre os nós, pode haver perda de dados. Isso pode ocorrer, por exemplo, se um nó primário falhar e um secundário for promovido a primário, mas alguns dados não foram replicados para o secundário antes da falha.

4. **Quando ocorrem problemas de desempenho**: em alguns casos, o Redis pode enfrentar problemas de desempenho devido ao tamanho do conjunto de dados ou à quantidade de transações em paralelo. Isso pode levar a falhas no sistema e, possivelmente, perda de dados como as citadas anteriormente.

Quanto à consistência dos dados, para contornar esse problema, pode-se aplicar *proxies* para paralelização em conjunto com outros sistemas de banco de dados persistentes de captura instantânea ou gravação de transações em arquivo (*AOF*). Porém, esta abordagem não garante a consistência de dados em caso de falhas ou interrupções, sobretudo durante o processo de *failover*. Já em relação à resolução de Conflitos, é possível utilizar estruturas P/S, como uma abordagem pessimista, evitando o surgimento de conflitos.

A solução proposta neste trabalho visa implementar um *framework* de replicação síncrona que garante a consistência de dados mesmo em caso de falhas e evita conflitos por meio de uma abordagem pessimista, aplicando um modelo de replicação descentralizado. Buscando assim o desenvolvimento de um sistema que ao mesmo tempo se beneficie com o alto desempenho proporcionado pelo Redis, e garanta a consistência dos dados, dado pelas propriedades de replicação síncrona do Treplica.

4 Treplica

O Treplica é um *framework* de replicação síncrona desenvolvido por Gustavo Vieira (VIEIRA; BUZATO, 2008) e que fornece uma interface de replicação síncrona baseada em consenso, por meio da implementação do algoritmo de ordenação total Paxos. O *framework* ainda é capaz de superar obstáculos como desempenho e disponibilidade em caso de falhas em componentes, aplicando uma complexa abordagem de recuperação de falhas. Do ponto de vista de usabilidade, oferece aos desenvolvedores um modelo de programação simples baseado em especificação orientada a objetos. Os trabalhos anteriores mostram que o Treplica é capaz de fornecer a capacidade de processamento necessária para garantir bons tempos de resposta (VIEIRA; BUZATO, 2010; BUZATO; VIEIRA; ZWAENEPOEL, 2009; VIEIRA; BUZATO, 2013).

O Treplica foi avaliado experimentalmente com o *benchmark TPC-W*, que é um conjunto de testes comumente usado para medir o desempenho de sistemas de banco de dados. Os resultados mostraram que é capaz de garantir bons rendimentos e tempos de resposta para aplicativos e, além disso, é capaz de garantir serviço ininterrupto mesmo com múltiplas falhas (VIEIRA; BUZATO, 2010).

Assim, oferece aos desenvolvedores de aplicativos um modelo de programação simples e orientado a objetos, desenvolvida em linguagem Java. O *framework* oferece uma interface que abstrai a replicação entre componentes de um sistema distribuído e seja resiliente a falhas, ofertando ainda um protocolo de recuperação, tal que seja possível programar seus aplicativos sem se preocupar com o estado da aplicação, que é gerenciado e replicado pelo Treplica.

4.1 Replicação

O Treplica implementa um modelo de replicação descentralizado (*P2P*), no qual, diferentemente do modelo P/S, qualquer nó é capaz de realizar escrita e leitura. Do ponto de vista de rede, a comunicação entre os nós se dá por um sistema de mensagens que opera no protocolo UDP/IP, o qual exige que todas as instâncias estejam em uma mesma rede UDP (podendo esta ser uma rede física ou virtual). Não é necessário indicar o endereço das réplicas entre si, já que são capazes de identificarem seus processos pares, sendo necessário apenas configurar qual é o número esperado de réplicas de um determinado processo (VIEIRA; BUZATO, 2010).

A replicação síncrona ou ativa é um tipo de replicação utilizado em sistemas distribuídos para garantir a consistência de dados em um sistema mesmo em caso de falhas

ou atrasos nas mensagens entre os processos. Nessa forma de replicação, os processos que compartilham o mesmo estado, chamados de réplicas, se comportam como máquinas de estado determinísticas. Todas as réplicas compartilham a mesma fonte de eventos que acionam transições em suas máquinas de estado subjacentes. Como resultado, todas as réplicas se mantêm iguais enquanto processam a mesma sequência de eventos (LYNCH, 1996).

Uma das principais vantagens da replicação síncrona é que ela garante a consistência dos dados em todas as réplicas. Isso significa que, mesmo em caso de falhas ou atrasos nas mensagens entre os processos, os dados mantidos pelas réplicas ainda serão consistentes entre si. Isso é importante em sistemas em que os dados precisam ser acessíveis e confiáveis mesmo em caso de falhas ou problemas no sistema. Além disso, a replicação síncrona também pode aumentar a disponibilidade do sistema, pois permite que os processos continuem a operar mesmo em caso de falhas em algumas das réplicas.

Existem várias formas de implementar a replicação síncrona, como o uso de primitivas de broadcast com ordenação total para propagar os eventos de forma ordenada entre as réplicas, ou o uso de algoritmos de consenso, como o algoritmo Paxos. No entanto, a replicação síncrona pode ser lenta em comparação com outros tipos de replicação, pois depende de uma série de mensagens trocadas entre os processos para garantir a consistência dos dados. Além disso, ela também pode exigir mais recursos do sistema, como largura de banda e memória, para garantir a replicação eficiente dos dados.

O algoritmo Paxos é um algoritmo de consenso amplamente utilizado em sistemas distribuídos para garantir que os processos de um sistema concordem em um valor ou estado compartilhado. Este algoritmo é baseado em uma série de fases nas quais os processos do sistema concordam em um valor ou estado compartilhado. Na primeira fase, chamada de fase de proposta, um processo propõe um valor para ser aceito pelos demais processos. Na segunda fase, chamada de fase de aceitação, os processos aceitam ou recusam a proposta. Se a proposta for aceita por um número suficiente de processos (definido no Treplica como $50\% + 1$), ela é considerada válida e é aceita pelo sistema como um todo. Caso contrário, o processo que propôs o valor deve tentar novamente.

O algoritmo Paxos é baseado em algumas suposições sobre o sistema no qual ele é utilizado, como a existência de um número fixo de processos e a capacidade de cada processo de enviar e receber mensagens de qualquer outro processo. Ainda, ele supõe que os processos podem ter falhas por colapso, mas que eles são capazes de se recuperar e continuar a participar do sistema após a falha. Por fim, o Paxos assume que os processos são honestos e não tentam subvertê-lo de alguma forma.

4.2 Como o Treplica opera

Um dos principais componentes do Treplica é a implementação do algoritmo de consenso Paxos, não para definir se uma solicitação será ou não atendida, mas quando ela será atendida (ordenação das transações). Assim, o *framework* abstrai não apenas a implementação do algoritmo, como é responsável por tomar conta do estado de cada instância, chamado de `StateMachine` [1], o qual é acessível à aplicação, como também é responsável pela consistência e persistência dos dados.

Dois métodos são expostos: `getState()`, responsável por obter o estado de memória replicado, retornando um objeto `Object`; e `execute(Action)`, este método é uma solicitação de alteração do `StateMachine` que recebe um parâmetro do tipo `Action`. Uma `Action` é uma função que modifica o estado. Assim, a aplicação possui um estado gerenciado pelo Treplica.

Um importante ponto a ser destacado é que o estado de um nó somente é alterado por meio de uma chamada à função `execute(Action)`, porém, qualquer nó poderá executar esta função, tal que as atualizações de estado podem ocorrer a qualquer momento. Neste contexto, todo acesso ao objeto de estado retornado pelo Treplica precisa ocorrer dentro de um bloco sincronizado e as transações que dependem deste estado devem operar utilizando a abordagem Java `wait/notify` sendo desenvolvidas sob algum *thread* da própria aplicação, em vez da abordagem de *callback*. Isto ocorre pois as ações são reexecutadas em todas as réplicas, e até são reexecutadas quando ocorre uma recuperação de falha. Logo, *callbacks* podem ser chamados em momentos inoportunos.

Após receber uma solicitação `execute(Action)`, que recebe uma transação Redis, composta por uma ou mais operação, é agrupada em um bloco com uma ou mais transações, e será enfileirada em uma fila assíncrona persistente (pela função denominada `enqueue()`). Destaca-se que a partir deste ponto todas as transações são registradas em disco por meio da função `write()` e podem ser recuperadas pela função `read()` para fins de recuperação.

A partir deste ponto, o estado da fila assíncrona persistente, contendo grupos de solicitações, é compartilhado entre diversos nós, fazendo assim o sistema tolerante a falhas. Então, com base neste estado compartilhado entre as réplicas, o algoritmo Paxos (LAMPORT, 1998b) é acionado a fim de obter um consenso sobre a ordenação deste grupo. Perceba que as transações dentro do grupo respeitam uma certa ordenação, com base em sua adição pelo `execute(Action)`. Um processo propõe uma rodada Paxos com base nos registros nesta fila e a medida que as réplicas andam com os consensos (implementação do algoritmo Paxos), os grupos de transações devidamente decididos são removidos desta fila e entregues de volta à aplicação da camada superior pela função `dequeue()`. A aplicação então, ao receber o retorno de `dequeue()`, pode utilizar o valor do retorno, caso esteja

[1] Classe Java customizada, disponível no pacote Treplica

relacionado às suas requisições.

Mesmo que o nó falhe neste ponto, a implementação de recuperação do Treplica irá substituir este nó por uma novo nó, cuja aplicação tem o estado inicial vazio e utiliza as informações gravadas em disco para recuperar seu estado. O estado é recuperado reaplicando todas as transações na ordem que foram decididas, e tornando-se assim disponível para atender clientes e participar de futuras rodadas.

4.3 Recuperação

O algoritmo Paxos, implementado pelo Treplica, é projetado sob o modelo falha-e-recuperação. Neste modelo, assume-se que um dos componentes de um sistema distribuído (réplica) pode sofrer um colapso total, e que a recuperação deve ser realizada por meio da substituição deste nó por outro, em perfeito funcionamento, recuperando assim o estado da réplica armazenado em memória permanente antes da falha (TANENBAUM; STEEN, 2006; LYNCH, 1996).

No Treplica, todos as mensagens trocadas com um determinado nó são escritas em seu disco de maneira sequencial. Ocasionalmente o Treplica poderá substituir estes arquivos de logs chamado de `ledger` (que funcionam de maneira similar ao *AOF* do Redis) por um instantâneo, que contêm não uma serie de mensagens, mas um estado da aplicação a ser substituído.

Quando um nó colapsa e logo depois se recupera, o Treplica restaura o estado do último instantâneo de seu disco e caso haja arquivos de logs não inclusos no instantâneo ele inicia o *"replay"* das transações ali registradas. A reexecução é idêntica a que acontece quando uma ação é propagada entre as réplicas. Entenda que, tratando-se de um sistema de replicação ativa, não é necessário que um nó copie o estado de outra réplica para garantir a sincronia, mas sim que dado a premissa de que os nós contêm um mesmo estado inicial (vazio ou vindo de um instantâneo), e sejam submetidos a uma mesma sequência de eventos deterministas, seus estados finais serão iguais. Isto implica que este nó terá ao fim do processo de recuperação o mesmo estado que tinha antes da falha.

No caso de falha, recuperação ou lentidão na rede, as réplicas podem ter um conjunto de dados distinto das instâncias envolvidas nas últimas rodadas de decisão, e portanto um estado diferente. Porém isto não causa problemas de escrita, uma vez que ao envolver-se em uma rodada de decisão subsequente, a instância desatualizada procura a instância mais atual e sincroniza seus dados. Em caso de leitura, para garantir a consistência dos dados, faz-se uma transação com escrita vazia inicialmente (que garante à replica a versão mais recentes dos dados) e a posteriori a leitura do dado em si.

4.4 Liberação de memória

O Treplica apresenta uma estratégia de gerenciamento de disco que depende da realização de instantâneos em todas as instâncias. Quando uma réplica realiza um instantâneo, esta armazena um indicador que é a última versão do `ledger` presente neste instantâneo. Este valor é incluído em toda mensagem propagada. Não obstante incluir seu próprio, a replica adiciona em um vetor a maior versão das demais réplicas com as quais trocou mensagem, criando assim um vetor de versões do `ledger` nos instantâneos de cada réplica. Assim, ao perceber que uma mensagem na qual a versão de todas as demais réplicas é maior que a sua, indica que todas as decisões tomadas até aquele momento estão persistidas em disco em todas as réplicas, e pode-se excluir com segurança todos os registros do `ledger` anteriores a este momento, reduzindo assim a quantidade de dados salvos em disco.

5 Treplica-Redis

A fim de solucionar os problemas citados na Seção 3.7, desenvolveu-se neste projeto a aplicação nomeada Treplica-Redis. Esta foi desenvolvida como um *Proxy* do Redis, e oferece a implementação de uma arquitetura de distribuição de dados com replicação síncrona e ordenação total para o Redis, usando o *framework* Treplica, descrito no Capítulo 4, que por sua vez implementa o algoritmo Paxos.

Assim, objetiva-se fornecer consistência dos dados em sistemas distribuídos no modelo *P2P*. Isto é feito por meio de uma abordagem pessimista, que evita possíveis conflitos realizando uma ordenação total das transações, de tal maneira que não existam concorrentes entre si. É importante observar que este modelo se ancora em algumas premissas, das quais podemos destacar a seguir:

i **Determinismo**: Todos os nós se comportam como máquinas de estado deterministas que possuem o mesmo estado inicial e ao serem submetidos a uma mesma sequência de eventos deterministas, que alterem seu estado, devem permanecer iguais.

ii **Consenso**: Se um nó que não falha decide algo, todos os outros nós que não falham decidem a mesma coisa.

iii **Difusão**: Todo nó que não falha recebe, mais cedo ou mais tarde, todas as transações a serem executadas.

iv **Escalonamento Serial**: As transações são executadas de maneira sequencial, a partir de uma ordenação total, sendo portanto um registro atômico, tal que se B é posterior a A, B só poderá ser executado após A.

A aplicação é dividida em três camadas interconectadas entre si: a primeira refere-se ao **Manager**, que é responsável pela comunicação com o cliente bem como pelo gerenciamento da aplicação como um todo e pela troca de dados entre as camadas. A segunda camada, **Treplica**, armazena o estado do Manager, orquestra a ordenação das transações e a replicação destas, e por fim persiste em disco o estado do manager, tornando-se a ferramenta central no processo de recuperação de falhas. Por fim, é apresentado o SGBD **Redis**. Esta camada é acessível e manipulada pelo Manager, e contém o banco de dados que em última instância realizam-se as transações.

5.1 Arquitetura

Em termos técnicos a aplicação é desenvolvida sob o *framework* SpringBoot (SHARMA, 2019) e projetada para ser operada em contêineres e/ou com orquestrador, que já fornece uma série de bibliotecas e pacotes, simplificando o seu desenvolvimento.

O código em si pode ser abstraído em três camadas, ilustradas na Figura 5.

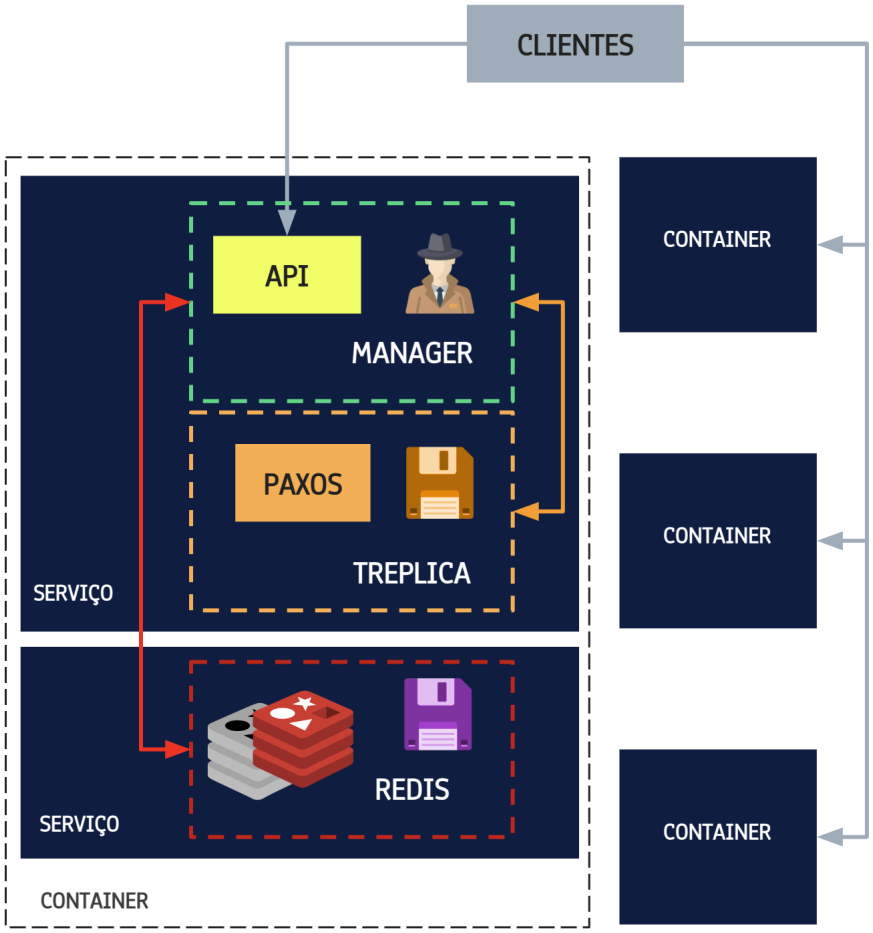

Figura 5 – Arquitetura da aplicação

As camadas que compõem a aplicação são:

i **Manager**: O Manager, é uma ponte que conecta o Treplica ao Redis. O Manager abre mão de seu estado para o que mesmo seja armazenado pelo Treplica. Em termos de estrutura, o estado é composto por duas variáveis, a primeira contém a lista de transações ordenadas pelo Treplica, e a segunda é um inteiro (*version_stamp*), que contabiliza o número de transações já ordenadas. O Manager é classificado como uma aplicação *stateless*, seu estado não é persistido e pode ser reconstruído desde que tenha acesso ao Redis e ao estado compartilhado do Treplica. Assim, esta camada tem como principais atribuições a sincronização entre as demais e fornecer

um protocolo de comunicação com o cliente, recepcionando e gerenciando todo o fluxo das transações solicitadas.

ii **Treplica**: O Treplica, por sua vez, armazena o estado do Manager em disco. Porém, ele não manipula este; apenas recebe solicitações de alteração de estados. Estas solicitações carregam uma estrutura chamada `Action`, que é um modificador do estado. Estas `Actions` são os itens ordenados pelo algoritmo de consenso, replicados com os demais nós e então aplicadas no estado.

iii **Redis**: Esta camada implementa um servidor Redis. Este pode operar como *single instance* ou no modelo P/S, e opcionalmente com Sentinel. Este componente implementa dois repositórios de dados, o primeiro, em memória primária, executa as transações entregues a ele pelo Manager, alterando o estado e, adicionalmente, guarda um *version_stamp* para cada transação. O segundo repositório por sua vez é um instantâneo do primeiro, armazenado em memória persistente. Vale destacar que o objetivo de um desenvolvimento em camadas é poder desacoplar e personalizar as camadas de acordo com o objetivo da aplicação. Apesar de neste trabalho ter sido utilizado o SGBD Redis, poderia ser aplicado a qualquer outro SGBD.

Apesar dos repositórios possuírem conjuntos distintos de dados, existe uma interdependência entre eles, de tal maneira que a aplicação possui um modelo bem estruturado de sincronia e transformação de dados entre as camadas.

5.2 Sincronização

A sincronia entre as camadas Redis e Manager se dá por meio do uso de *version_stamp*, que é um registro do SGBD que armazena informações sobre a versão do armazenamento. Do ponto de vista destes componentes, o Treplica fornece uma garantia que, se uma `Action` é adicionada ao estado do Manager, esta foi entregue na ordem correta. E cada `Action` entregue incrementa o valor do *version_stamp*.

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 replicados entre as réplicas, apenas as `Actions`. Isto possibilitou a construção de um modelo que manipula o estado do Manager em dois momentos: O primeiro é quando há o retorno de uma ordenação e a `Action` é executada. No Treplica-Redis uma `Action` é uma função que adiciona 1 ao *version_stamp* e adiciona no final da lista de transações uma transação para ser executada no Redis. Já o segundo momento é quando a transação em si é executada no Redis. Dado que esta transação já está no SGBD, não há necessidade de manter seu registro no Manager, e então o estado é alterado (removendo esta transação da lista) mas sem gerar uma nova `Action`, ou seja, 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.

De maneira geral você pode compreender a intercomunicação desta aplicação como um modelo baseado em mensagens, em que os dados compartilhados entre as camadas são mensagens. Dado que estas mensagens foram, em algum momento processadas pelo Treplica, foram também persistidas em disco, como uma estratégia para garantir a consistência dos dados em caso de recuperação.

5.3 Modelos de Transações

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 (LAMPORT, 1985), conforme visto na Seção 2.3.2. A nível de isolamento, trata-se de uma transação serializada, ou seja, não permite nenhuma das violações (ELMASRI; NAVATHE, 2005), dando ao cliente a garantia de consistência dos dados. Por outro lado, o modelo regular segue 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.

A partir deste ponto, o termo transação será utilizado para designar o modelo de transações, em que uma transação atômica refere-se a uma transação no modelo atômico, e uma transação regular refere-se a uma transação no modelo regular. Vale destacar que nesta primeira versão do Treplica-Redis, cada transação é composta apenas por uma operação, que pode ser de leitura ou escrita. Futuramente, a implementação pode ser estendida para abarcar transações com mais operações.

Quanto aos modelos de transações, as mesmas são implementadas da seguinte maneira:

5.3.1 Atômicas

Nas transações atômicas, compostas tanto por operações 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, que dá um identificador para esta transação e adiciona este à uma tabela de transações pendentes. Então, a transação 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 explicado em 5.2. Este processo ocorre tanto para escrita quanto para leitura em modelo atômico. Isto ocorre pois a ordenação total tem como propriedade de níveis de isolamento de transação como uma leitura em escalonamento serial.

Nas transações atômicas, é verificado, antes de cada execução, se a transação a ser processada é uma transação pendente de retorno para o cliente (ou seja, se a transação tem o identificador na tabela de transações pendentes), e caso sim, atribui um valor de retorno a esta tabela. Então, quando o processo que recepciona as requisições identifica que existe um retorno nesta tabela, responde-o ao cliente e então remove o item da tabela.

5.3.2 Regulares

Já nas transações regulares há a consistência relaxada, a leitura ou escrita são abordadas de maneira diferentes. As transações compostas por operações de leituras apenas, são executadas imediatamente na memória primária do Redis, retornando o dado encontrado para o cliente sem consultar os demais nós, podendo trazer um dado que não seja o mais recente. Já as transações que possuem escritas apenas, as mesmas são inserida em uma fila, e antes mesmo de ser processada, retorna uma confirmação para o cliente. Esta confirmação indica que uma transação foi inserida em uma fila para ordenação, porém não ocorre o procedimento de espera descrito na transação atômica. Futuramente, está será ordenada e operada no Redis, porém, até que isso ocorra, violações de leitura confirmada podem ocorrer. Destaca-se que transações mistas operam como atômicas.

O Manager expõe uma *API* na qual os clientes da aplicação podem requisitar transações do tipo atômicas ou regulares por meio dos métodos apresentados a seguir.

5.4 *API*

Para comunicar-se com os clientes, a aplicação desenvolvida inicia um servidor web, expondo, em uma *API* do tipo *Restful*, um *endpoint* para cada um dos comandos Redis disponibilizados. As funções básicas do Redis que são abstraídas na *API*, são:

- `set(key as String, value as T)`: Cria uma nova tupla de dados, se e somente

se a chave indicada for única; caso a chave não seja única, edita (sobrescreve) o valor da mesma. Equivale à função Redis `SET KEY VALUE`.

- `get(key as String)`: Busca e retorna ao cliente o dado (valor) referente à chave solicitada. Equivale à função Redis `GET KEY`.
- `incr(key as String, <optional> number as Double)`: Incrementa para a chave solicitada com o `number` informado, ou incrementa `1.0` caso `number` não seja informado. Equivale à função Redis `INCR Key Value[Optional]`.
- `delete(key as String)`: Busca e remove a tupla referente à chave solicitada. Equivale à função Redis `DEL Key`.
- `rename(key as String, newKey as String)`: Renomeia a chave de uma tupla, se e somente se a nova chave indicada for única. Equivale à função Redis `RENAME key newkey`.
- `setExpire(key as String, time as Integer)`: Define o tempo de validade (do inglês *ttl, time to live*) de uma chave. Equivale à função Redis `EXPIREAT key unix-time-seconds`.
- `deleteExpire(key as String)`: Remove a validade (do inglês *ttl, time to live*) de uma chave. Equivale à função Redis `PERSIST key`.
- `getExpire(key as String)`: Busca o tempo de vida restante (*ttl - Time to live*) de uma chave em millisegundos. Equivale à função Redis `TTL key`.
- `getAllKeys()`: Lista todas as chaves armazenadas. Equivale à função Redis `KEYS *`.

Por padrão, a aplicação executa um servidor web na porta UDP 8080, podendo esta ser reconfigurada. Se considerarmos as definições padrão, os *endpoints* gerados terão como URL base *http://localhost:8080*, para executar uma transação é necessário inserir a URL base seguida do tipo de transação (*/atomic* ou */regular*), seguido do caminho de cada transação.

As diferenças entre os tipos de transações serão apresentadas na Tabela 1, mas ambos disponibilizam os mesmos comandos, portanto, é possível realizar as ações com as seguintes configurações:

5.5 Implementando os Tipos de Transações

Descreveremos abaixo alguns detalhes das implementações de transações do tipo atômicas e regulares descritas através de pseudo-códigos, partindo de uma requisição do usuário.

Tabela 1 – Tabela de operações

Operação	Método	Corpo JSON	Param. URL	Retorno JSON	Caminho
set	POST	"key":"foo" "value":1		-	/set
get	GET		key*	"key":"foo" "value":1	/get?key=foo
incr	PUT		key* number		/incr?key=foo& number=3.2
rename	PUT		key* newkey*		/rename?key=foo& newKey=bar
del	DELETE		key*		/del?key=foo
expire	PUT		key* time*		/expire?key=foo& time=10
expire	GET		key*	"key":"foo" "ttl":100	/expire?key=foo
expire	DELETE		key*		/expire?key=foo
allKeys	GET		-	"foo" "bar"	/allKeys

* indica um argumento obrigatório

Ao **recepcionar uma transação**, conforme descrito no Algoritmo 1, inicialmente verifica-se se o tipo da transação (atômica/regular) e se o método corresponde a uma leitura ou escrita. Caso seja uma leitura regular, executa-se a transação no Redis (`executaRedis`) e retorna o resultado ao usuário. Caso seja uma escrita regular, envia-se a transação ao Treplica (`executaTreplica`) e imediatamente retorna uma confirmação ao usuário que a transação foi enfileirada. Já para as transações atômicas, cria-se um identificador único para a requisição que será utilizado posteriormente para retornar a resposta ao usuário e adiciona-o na lista de transações pendentes. Com o identificador presente na lista de transações pendentes, o Manager envia a transação ao Treplica (`executaTreplica`) e fica ouvindo o retorno de cada ordenação realizada pelo Treplica por meio dos métodos `wait/notify` do Java (OAKS; WONG, 1999). Então, quando a transação solicitada por este cliente for executada no Redis e ter seu valor adicionado na lista de transações pendentes, será então entregue ao cliente e terá seu registro apagado da lista de transações pendentes.

Observe que o evento de `executaTreplica` no Algoritmo 1 é uma chamada à *API* do Treplica, a qual recebe os parâmetros da transação a ser ordenada, por meio de seu algoritmo de consenso sobre a ordenação. A medida que a ordenação das `Actions` é definida e as mesmas replicadas, cada instância do Treplica inicia um processo de enviar as transações em ordem para o Manager, que é notificado por uma função interna do Treplica

Algoritmo 1 Recepcionar uma transação

```
1: metodo, tipo, operacao, parametros ← requisicao
2: retornoPendente ∨ ∅
3: id ← criarIdunico()
4: action ← {id, operacao, parametros}   ▷ Define uma Action que pode ser consumida
                                           pelo Treplica e/ou Redis
5: if tipo = "regular" ∧ metodo = "get" then
6:     resultado ← executaRedis(action.operacao, action.parametros)
7:     retorna resultado
8: else if tipo = "regular" then
9:     executaTreplica(action)                ▷ Esta é uma ação assíncrona
10:    retorna true
11: else
12:    retornoPendente.append(id, NULL)
13:    executaTreplica(action)                ▷ Esta é uma ação assíncrona
14:    while retornoPendente[id] = NULL do
15:        aguardarNotificacao : operacaoAplicada()   ▷ Pausa até que o Manager
                                                       retorne alguma transação aplicada no Redis
16:    end while
17:    resultado ← retornoPendente[id]
18:    retorna resultado
19: end if
20: if id ∈ retornoPendente then
21:    retornoPendente.delete(id)
22: end if
```

que iremos nomear de `emitirNotificacao:actionOrdenada`

O Algoritmo 2 representa as ações que serão realizadas ao receber uma notificação `emitirNotificacao:actionOrdenada`. Quando esta notificação é recebida já existe uma nova transação adicionada ao estado do Manager. Para cada nova transação no estado, o Manager irá executá-la no Redis, alterando assim o estado do SGBD. Para cada uma destas execuções espera-se um retorno, o qual deverá ser adicionado a lista de transações pendentes e então emitir uma notificação do tipo `emitirNotificacao:operacaoAplicada`.

Por meio destes algoritmos, a aplicação garante a consistência forte dos dados enquanto mantém todos os estados sincronizados. Podemos ilustrar a aplicação destes algoritmos em um exemplo a seguir no qual o usuário realiza duas ações:

- **Ação 1 (Figura 6)**: 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 7)**: Chave `A` já existe com o valor 15, como podemos ver, após receber uma transação com a operação `INCR` para a chave `A` no valor 3, é esperado que o valor desta chave seja atualizado para 18:

Algoritmo 2 Sincronizar

```
1: Redis ← {data ∨ ∅, version_stamp = 0}
2: retornoPendente ← retornoPendente
3: while true do
4:     Manager ← Treplica.getState()
5:     if Redis.version_stamp() ≥ Manager.version_stamp() then
6:         aguardarNotificacao : actionOrdenada()
7:     end if
8:     if Redis.version_stamp() < Manager.version_stamp() then
9:         for all action ∉ Manager do
10:            resposta = executaRedis(action.operacao, action.parametro) ▷ Esta ação
                                                executa a transação no servidor Redis
11:            Redis.version_stamp+ = 1
12:            if action.id ∈ retornoPendente then
13:                retornoPendente[action.id] ← resposta
14:            end if
15:            emitirNotificacao : operacaoAplicada()
16:        end for
17:    end if
18: end while
```

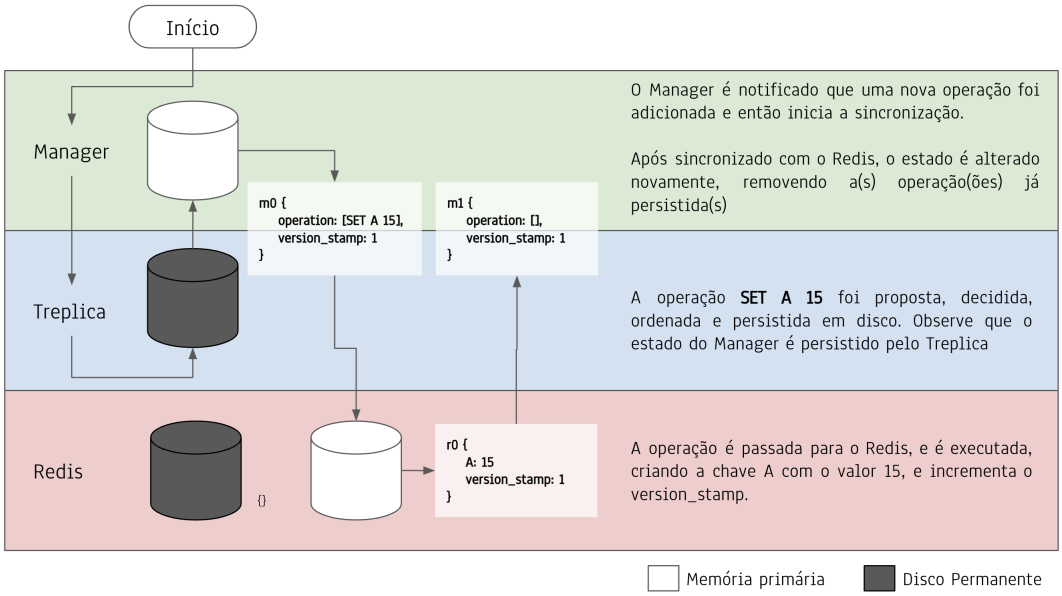

Figura 6 – Ação 1: Criando um registro

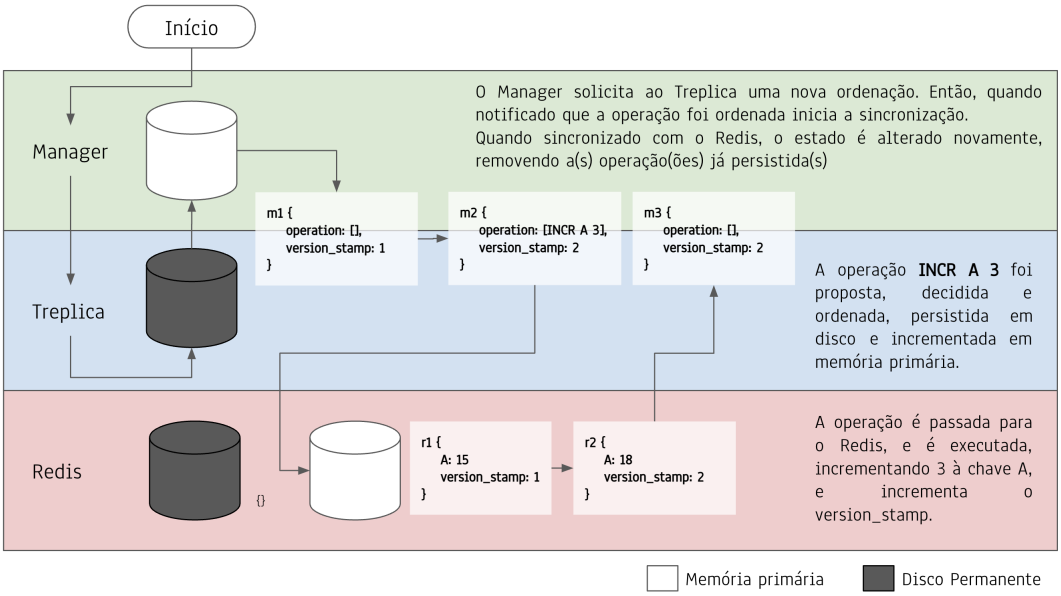

Figura 7 – Ação 2: Atualizando um registro

5.6 Persistência

Nesta aplicação, as gravações em disco são utilizadas prioritariamente como estratégia para garantir a consistência dos dados e recuperação de falhas. Cada camada possui seu próprio estado como descrito anteriormente, e aplica estratégias de uso do disco de maneira diferente.

Os recursos de uso e gerenciamento do disco pela aplicação são:

1. O componente `ledger` do Treplica armazena em disco todo registro de mensagens encaminhadas. A partir deste registro, é possível remontar os estado do Manager e por consequência do Redis, uma vez que, no processo de recuperação, o estado do Manager iniciará vazio e receberá todas as `Actions` em ordem.

2. Tanto no Redis quanto no Treplica é possível gerar instantâneos, que têm como objetivo tornar o reinício da aplicação mais rápido em caso de recuperação de falhas (destaca-se que não é um item essencial para o funcionamento da aplicação). Em cada nó da aplicação, define-se um número primo x distinto. Este número é utilizado como gatilho para disparar um comando para criar um instantâneo em ambas camadas. Este processo é ilustrado na Figura 8.

3. Dado a criação de um instantâneo no Treplica, um serviço interno de coleta de lixo apaga arquivos antigos substituídos pelo instantâneo.

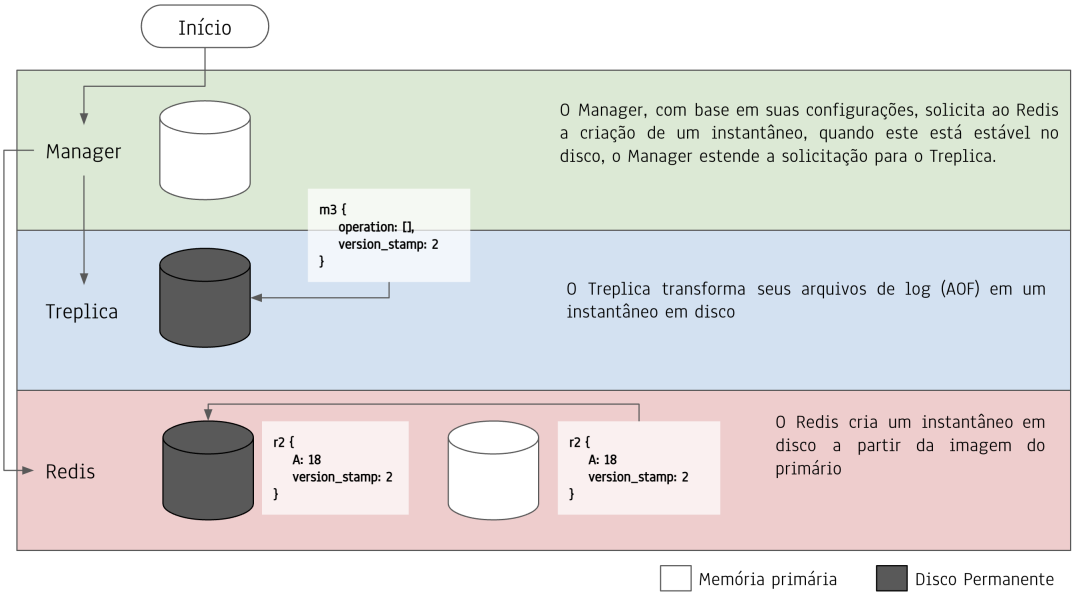

Figura 8 – Fluxo de Persistência e liberação de memória

5.7 Detecção e recuperação de falhas

Nesta aplicação pressupõe-se a ocorrência de falha do tipo **falha-e-recuperação** nos componentes no Manager e no Redis. Para falhas no Manager, o próprio contêiner que engloba a aplicação é capaz de verificar o estado da aplicação e, 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 (comando `PING`), tal que a não obtenção de respostas um período de tempo pré-definido é considerado como uma falha que reinicia toda a aplicação mantendo o disco.

Sendo detectado uma falha, inicia-se um procedimento de reinício completo da aplicação. O processo de inicialização do SGBD Redis busca o último instantâneo realizado e aplica-o como estado inicial. O Treplica recupera seu estado a partir dos dados armazenados no disco permanente. Primeiro realiza um reinício rápido, reaplicando o último instantâneo disponível, e então recupera os dados posteriores ao instantâneo a partir do seu registro de *logs*, que tem todas as informações decididas até o momento de sua falha. Ao término destas transações, o Treplica terá o mesmo estado que tinha antes da falha e já está pronto para integrar novas rodadas de decisão do algoritmo Paxos e, assim, obter os eventos que por ventura podem ter ocorridos durante o processo de recuperação. Por fim, o Manager tem seu estado recuperado pelo Treplica, recebendo deste todas as transações ordenadas, tal que para cada transação recepcionada o método de sincronização 5.2 é invocado. Assim, caso alguma das transações já tenham sido executadas no Redis, estas serão ignoradas, enquanto as demais serão operadas no Redis a fim de garantir a sincronização entre o Treplica e o Redis. Este fluxo é ilustrado nas Figuras 9 e 10:

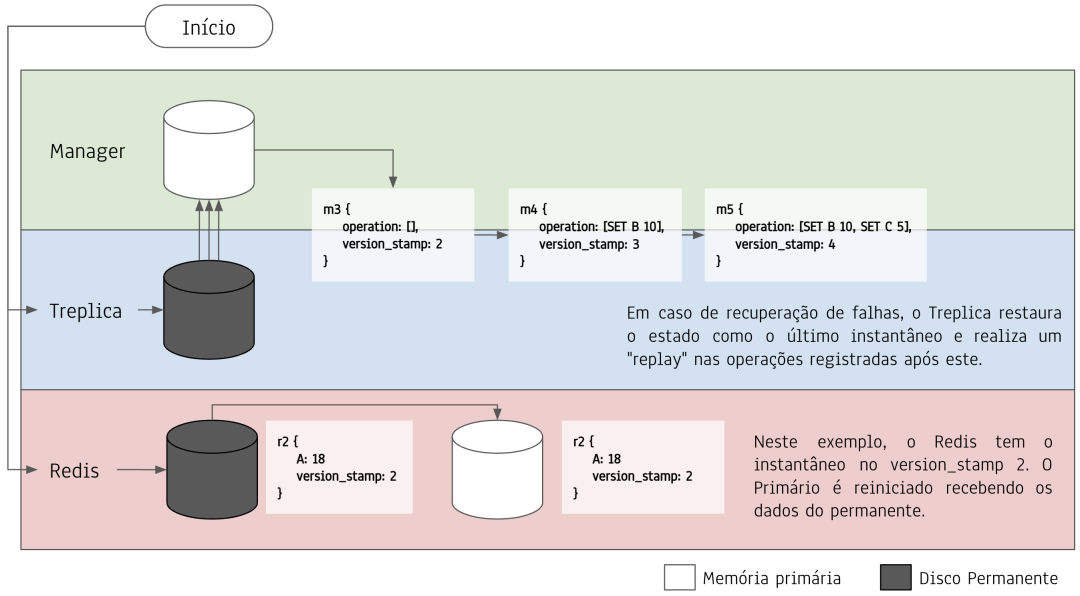

Figura 9 – Fluxo de recuperação de falhas - Parte 1

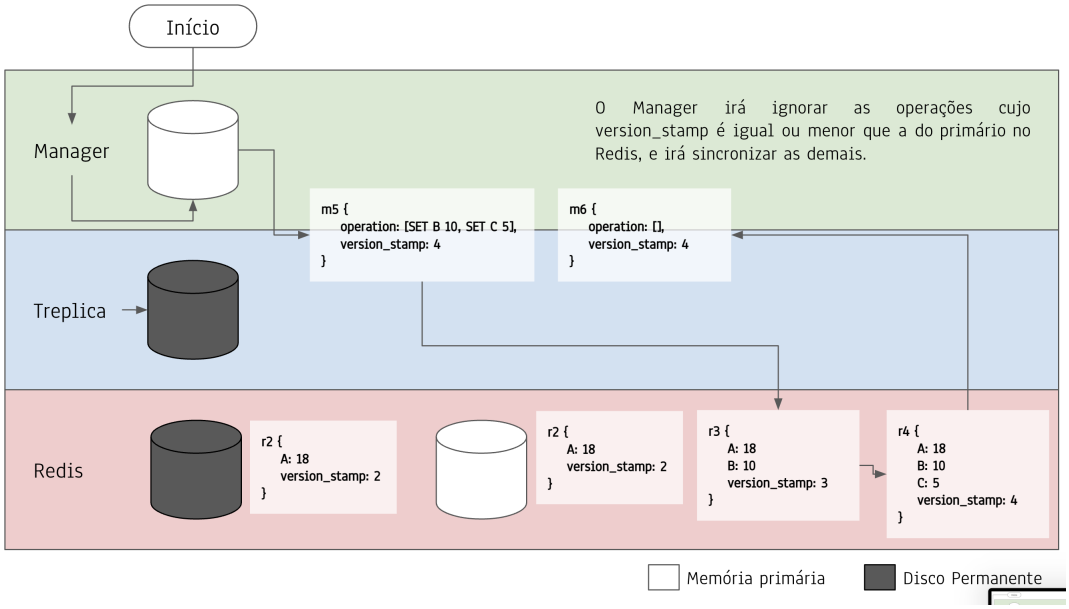

Figura 10 – Fluxo de recuperação de falhas - Parte 2

6 Experimentos e resultados

Como apresentado no Capítulo 3, uma das questões ainda em aberto do SGBD Redis é a consistência dos dados, a qual pode ser violada em alguns casos. Dentre estes, por exemplo, quando houver um *failover* antes que uma transação executada no primário seja replicada na réplica que assumirá o seu lugar. Também, em casos que há uma falha-e-recuperação e não existem instantâneos atualizados ou um método para reexecutar as alterações de estado. Ainda, em casos de partição na rede onde dois nós atuam simultaneamente como primário.

Por outro lado, o Treplica, abordado no Capítulo 4, por meio de suas propriedades de ordenação total e replicação descentralizadas com quórum de escrita, implementa um modelo que resolve as questões levantadas, provendo assim uma total consistência dos dados persistidos.

Assim, foi desenvolvido neste trabalho o *proxy* Treplica-Redis que utiliza um SGBD Redis, realizando leituras regulares com o máximo de desempenho oferecido pelo Redis, e ao mesmo tempo garante a consistência dos dados em escrita atômicas replicando e persistindo-os em disco por meio do *Treplica*, com o objetivo de permitir que tal sistema de armazenamento tenha uma consistência forte. Então, realizou-se um teste A/B a fim de compreender o *trade-off* entre consistência e desempenho para um mesmo SGBD. Isto indica que ao incrementar a consistência deste SGBD estamos, de alguma forma, sacrificando o seu desempenho. Logo o objetivo central do experimento executado é quantificar: o quanto do desempenho é sacrificado em busca da maximização da consistência?

6.1 Abordagem

A fim de medir o desempenho do *proxy* desenvolvido, optou-se por utilizar um teste de carga (SINGH; KHAN, 2017). Esse tipo de teste envolve a execução de um conjunto de cenários que representam a carga esperada em um ambiente de produção real, com o objetivo de identificar possíveis gargalos e pontos críticos no sistema. O objetivo é garantir que o sistema tenha uma capacidade suficiente para atender a demanda prevista e possa lidar com picos de tráfego sem interrupções ou degradação do serviço.

Assim, é possível avaliar a capacidade do sistema em lidar com essa carga, incluindo a capacidade de escalonamento, a capacidade de lidar com falhas e a capacidade de garantir a consistência dos dados. Além disso, os testes de carga podem ser usados para avaliar a eficiência de algoritmos de consenso, como o Paxos.

Para compreender o nível de consistência e desempenho fornecidos pela aplicação,

desenvolveu-se um *benchmarking* entre duas variantes, a primeira conforme descrita no Capítulo 5 e uma segunda que utiliza a mesma base porém desacopla-se do Treplica e utiliza a replicação embutida no Redis (Sentinel-Redis), que será abordada a seguir.

6.2 Aplicação de comparação Sentinel-Redis

A versão de comparação, nomeada Sentinel-Redis, foi desenvolvida por Souza (SOUZA, 2022) a partir do Treplica-Redis, utilizando a replicação nativa do SGBD e o Sentinel-Redis para replicação e recuperação a falhas, removendo portanto a camada do Treplica. Nesta aplicação o Redis continua a operar como a camada de armazenamento de dados e o Manager, que expõe o servidor *REST* acomoda as requisições dos clientes via *API*. A diferença entre as duas aplicações é que, enquanto a aplicação Treplica-Redis usa o Treplica para garantir consistência, a aplicação de Sentinel-Redis usa apenas o Redis, que possui uma consistência relaxada e aceita escritas apenas pelo nó primário (modelo P/S).

O autor também descreve a topologia da aplicação como um conjunto de contêineres Docker (DOCKER, 2020), com replicação P/S, no qual um destes é o nó primário e os demais secundários, de maneira que caso uma transação seja operada no nó secundário, esta é enviada ao nó primário e então executada no Redis e replicada. Enquanto na configuração da aplicação Treplica-Redis, a topologia de replicação é *P2P*, ou seja, qualquer nó pode receber requisições de escrita, enquanto as instâncias Redis agem individualmente como nós independentes e mantem o estado replicado graças às propriedades garantidas pelo Treplica.

A fim de garantir um ambiente de uso similar entre as variantes, com uma consistência mais forte, a aplicação Sentinel-Redis precisa aplicar algumas configurações ao servidor Redis, como:

- `appendonly yes`: Habilita o modo *AOF*. Isso significa que todas as transações com operações de escrita (que incluem `SET`, `DEL`, etc) são registradas em um arquivo especial chamado `appendonly.aof`. Essas transações são escritas no arquivo em ordem, permitindo que o banco de dados recrie o estado atual a partir do arquivo em caso de falha.

- `appendfsync always`: Indica que o sistema de arquivos deve sincronizar os dados com o disco a cada vez que uma transação de escrita é registrada no arquivo *AOF*. Isso garante que as transações estejam salvas no disco antes que a transação seja considerada completa e que o estado do banco de dados seja recuperado caso haja uma falha no sistema.

- `min-replicas-to-write 1`: Utilizada quando o banco de dados está configurado como um cluster Redis (Redis Cluster). Especifica o número mínimo de réplicas que

devem estar disponíveis para que as escritas sejam permitidas.

- `min-replicas-max-lag 5`: Utilizada quando o banco de dados está configurado como um cluster Redis (Redis Cluster). Especifica o número máximo de segundos de atraso que é permitido entre o nó principal e as réplicas antes que as escritas sejam bloqueadas.
- `replica-serve-stale-data no`: Especifica se as réplicas devem servir dados desatualizados quando não estiverem completamente sincronizadas com o nó principal. Se definida como `no`, as réplicas não poderão servir dados desatualizados.
- `sentinel down-after-milliseconds mymaster 5000`: Especifica o tempo (em milissegundos) que o primário pode ficar sem responder antes de ser considerado como em estado de falha e iniciar o processo de *failover*.
- `sentinel failover-timeout mymaster 30000`: Indica o tempo que o Sentinel aguarda o primário retornando, após iniciado o processo de *failover*. Caso o primário não retorne, uma réplica secundária é promovida a primário.

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

Vale destacar que apesar da 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, conforme evidenciado por Kyle Kingsbury (KINGSBURY, 2013). Portanto, ofertam uma consistência relaxada, enquanto a aplicação Treplica-Redis oferta uma consistência forte.

6.3 Elaboração dos testes

É esperado neste teste que o Treplica-Redis, por ser um *proxy* que garante uma consistência forte, apresente um desempenho menor que a versão de comparação Sentinel-Redis, que apresenta consistência relaxada (LIU, 2016), e portanto deverá apresentar melhor desempenho. Veja que além de ser uma evidência do teorema *CAP*, este experimento também trás como resultado o desempenho de cada uma das variáveis.

Inicialmente, tentou-se utilizar o software JMeter (APACHE. . . , 1998). Este é um software de teste de desempenho de código aberto desenvolvido pela Apache. Ele é usado para testar o desempenho de aplicativos web, sistemas de gerenciamento de banco de dados, servidores de aplicativos e outros sistemas. O teste é realizado enviando solicitações *HTTP*

e auferindo a resposta, ajudando assim a identificar gargalos e problemas de desempenho, bem como a consistência dos dados, por meio de validação na resposta. A aplicação suporta vários tipos de testes, incluindo testes de carga, e, além disso, possui uma interface gráfica intuitiva, bem como uma *API* para automatizar testes. No teste desenvolvido este envia diversas transações (agrupadas em *threads*) a cada um dos nós de um aplicação distribuída.

Percebeu-se entretanto, que o servidor *REST* operado pelo Springboot conta com uma limitação do número de chamadas *HTTP* que podem ser recepcionadas e operadas por segundo, criando então uma longa fila de requisições a serem atendidas assim que o processo de execução tenha disponibilidade. Esta limitação é na ordem de 200 chamadas por segundo. Com isso, mesmo que o sistema em si tenha capacidade de operar um número maior de requisições por segundo, o ponto de entrada cria um gargalo, não sendo possível observar portanto a real desempenho das amostras.

Assim, optou-se pelo uso de um outro software de teste de carga, desenvolvido como parte do Treplica, sendo integrado diretamente com o *framework* por meio de um método exposto na API do Manager que dispara o teste. Este software de teste utiliza o relógio universal para disparar simultaneamente requisições em várias réplicas, cada requisição ocorre em uma *thread* do sistema, de tal forma que várias requisições são iniciadas a cada segundo em cada réplica, e de maneira simultânea em todas as réplicas. Ao termino do teste os resultados de cada replica são agregados a fim de medir a capacidade do sistema distribuído e do algoritmo de consenso em lidar com um grande número de requisições simultâneas.

Deve-se informar o número alvo de transações por segundo, assim o teste tenta realizar uma estratégia de aquecimento, adicionando novas *threads* e transações por segundo em cada instância da aplicação alvo. Se por um lado o pensamento indutivo pode nos levar a crer que um maior número de paralelismo proporciona uma melhor otimização do sistema, devemos atentar que cada *thread* requer uma certa quantidade de recursos do sistema, como memória e *CPU*. Portanto, se este número cresce demasiado, isso pode levar a problemas de desempenho, como maior necessidade de recurso para o gerenciamento e aumento de latência, principalmente em uma longa fila de processamento.

Todos os cenários foram testados na mesma configuração de ambiente, sendo que cada aplicação é replicada em três nós, cada um executado em uma máquina 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.

6.4 Cenários de teste

Para realiza a comparação entre as variantes do sistema, foi escrito um único teste a ser executado por ambas versões. Este teste executa uma sucessão de transações compostas

por operações do tipo `SET`, cada operação (`set(key,value)`) cria uma chave aleatória e persiste como valor um arquivo JSON de variados tamanhos [1].

Logo, este teste é executado quatro vezes em cada aplicação (Treplica-Redis e Sentinel-Redis) com três réplicas. Cada teste tem a duração de 120 segundos, e neste deve ser informado, além da duração, o *JSON* e o número de transações por segundo a ser realizado em cada nó (*rate*). O teste ocorre com uma chamada a um endereço disponibilizado em cada instância da aplicação, logo o teste deve ser iniciado simultaneamente em todas as réplicas, por meio de uma chamada *HTTP* do tipo `POST` para o endereço `<replica>/test`, no qual deve-se enviar o corpo conforme modelo da Figura 11.

```
{
    "rate": <number>,
    "duration": 120,
    "payload": <JSON>,
    "mode": "SET",
    "operationType": "ATOMIC"
}
```

Figura 11 – Modelo de corpo para requisição de teste

O teste envia requisições com o objetivo de alcançar o a taxa de transações por segundo informado, portanto, um *rate* muito alto irá criar uma longa fila de espera, que poderá consumir muitos recursos da aplicação. Assim, objetivou-se encontrar valores que não prejudicassem o desempenho da aplicação.

Para a definição do *rate* apropriado, utilizou-se o modelo que deverá trazer o melhor desempenho (*JSON* de 4 bytes), e diversos testes foram efetuados a fim de encontrar o maior *rate* que não prejudicasse o desempenho da aplicação. Por exemplo, dado que a maior média de transações por segundo (t/s) obtida foi de 350 com o *rate* de 133 ($133\ x\ 3 = 400$, sendo 3 o número de réplicas no experimento), testa-se de maneira empírica incrementar o *rate* enquanto o número de t/s se mantiver em 350 com variância máxima de até 10%. Assim, para a aplicação Treplica-Redis o *rate* definido foi de 167 ($167\ x\ 3 = 501$) e para a aplicação Sentinel-Redis foi de 1100 ($1100\ x\ 3 = 3300$).

Ainda como parte do ETL, dos dados brutos são removidos o segundo inicial e final, visto que este tempo é utilizado para o aquecimento e desaquecimento da aplicação, nos quais as *threads* são iniciadas e encerradas. Ao executar o teste, cada instância gera

[1] Os tamanhos dos arquivos JSON podem ser de [i] 4 bytes, [ii] 350 bytes, [iii] 1,4 kilobytes e [iv] 3,5 kilobytes, disponíveis nos anexos Seção C

um arquivo de texto, onde cada linha representa uma transação e possui o tempo inicial e final de cada uma das transações executadas, conforme Figura 12.

```
1  Tempo inicial Tempo final
2  1666393187240 1666393187515
3  1666393187252 1666393188157
4  1666393187255 1666393188161
5  1666393187258 1666393188162
6  1666393187243 1666393188854
7  1666393187246 1666393188856
8  1666393187249 1666393188857
9  1666393187261 1666393188858
10 1666393187264 1666393188859
11 1666393187267 1666393188860
12 1666393187270 1666393189061
```

Figura 12 – Exemplo de histograma

A partir destes, é possível gerar um histograma [2] que consolida a execução de todas as réplicas, por meio do algoritmo `analyze-single-run.py` disponível na biblioteca do *software* de testes. O histograma é exibido com as informações de cada transação consolidadas conforme o exemplo apresentado na Figura 13 e pode gerar um gráfico utilizando recursos GnuPlot (WILLIAMS et al., 2004). Este histograma contém o segundo da execução, a quantidade de transações iniciadas, quantidade de transações finalizadas, e o tempo médio de resposta, respectivamente.

```
1 0 381.0 6.0 352.6666666666667
2 1 52.0 52.0 1234.75
3 2 54.0 54.0 2145.0925925925926
4 3 60.0 60.0 3119.15
```

Figura 13 – Exemplo de histograma

Também, é gerado o arquivo `overview.txt`, conforme exemplo na Figura 14, que contem informações consolidadas do histograma, que formarão a tabela 2 e serão discutidos a diante. Os resultados dos testes, com o arquivo de resumo e uma breve explicação sobre os mesmos estão disponíveis no Apêndice B.

[2] Os histogramas de cada teste estão disponíveis nos Anexos na Seção D

```
Time span (s): 50
Total operations: 100
Operations started (op/s):
   Average: 10.0 Stdev: 30.0   Median: 25.0
Operations completed (op/s):
   Average: 10.0 Stdev: 10.0   Median: 9.4
Response time (ms):
   Average: 120.0 Stdev: 12.0   Median: 100.0
```

Figura 14 – Exemplo de arquivo de resumo

6.5 Resultados consolidados e comparação

Após avaliar os resultados, foi possível observar que quanto maior o arquivo utilizado no teste, maior é o tempo de resposta médio da aplicação e menor é o número de transações que o sistema consegue processar. Este comportamento já era esperado e ocorre por alguns fatores, dentre eles destaca-se a sobrecarga na rede, uma vez que mensagens contendo o arquivo precisam ser trocadas entre as réplicas para o processo de replicação, e uma maior necessidade de recursos para persistir este dado em disco bem como armazená-lo em memória primária.

Os resultados consolidados da aplicação Treplica-Redis são apresentados na Tabela 2, seguidos do resultado da aplicação de comparação Sentinel-Redis, desenvolvida e com demais resultados no referido na Tabela 3 por Souza (SOUZA, 2022).

Tabela 2 – Tabela de resultados do Treplica-Redis

JSON	t/s (Média)	Desvio Padrão	Tempo de Resposta (ms) (Média)	Desvio Padrão	Transações Total
4 B	213,92	63,30	1.750	723,38	25.885
350 B	100,96	33,51	3.704	1.258,55	12.217
1,4 kB	79,70	21,43	4.526	1.719,99	9.644
3,5 kB	32,82	10,26	11.109	3.722,11	3.972

Percebe-se que os resultados das aplicações são bem distintos entre si. Enquanto o Treplica-Redis apresenta maior estabilidade no que tange às transações por segundo, tendo um desvio padrão de $16,5\%$, contra 25% em comparação com versão Sentinel-Redis, todos os outros parâmetros indicam uma redução no desempenho, que vai de 8,4 a 12,4 vezes. Destaca-se que esta comparação não tem como objetivo definir qual variante é

Tabela 3 – Aplicação de Comparação

JSON	t/s (Média)	Desvio Padrão	Tempo de Resposta (ms) (Média)	Desvio Padrão	Transações Total
4 B	1.796,09	629,57	243	117,78	217.327
350 B	1.251,83	488,31	371	165,41	151.472
1,4 kB	869,40	436,98	563	264,53	103.329
3,5 kB	333,33	141,65	1.318	431,70	40.334

significativamente melhor que outra, mas sim compreender o *trade-off* entre consistência e desempenho.

Por fim, vale destacar que grande parte dos servidores web encontrados não possuem suporte para número de requisições por segundos que o Redis pode oferecer, como é o exemplo do Springboot que utiliza um servidor Tomcat e tem desempenho máximo entre 1 e 2 centenas de chamadas por segundo (Apache Software Foundation, 2018). Neste contexto, o baixo desempenho da aplicação Treplica-Redis em comparação ao Sentinel-Redis pode nem sequer ser percebida pelos clientes e desenvolvedores da aplicação, caso esta utilize uma configuração de servidor web similar à utilizada nestes testes.

7 Conclusão

Neste trabalho objetivou-se construir um *proxy* Redis que garante consistência forte sob o modelo de replicação *P2P*. Apesar do SGBD possuir configurações capazes de ampliar a consistência dos dados, ainda assim é classificada como uma consistência relaxada, sobretudo durante falhas e processos de recuperação. Dentre os *proxies* avaliados, nenhum entrega um modelo de consistência forte com a garantia quanto à resolução de conflitos.

Com o desenvolvimento do Treplica-Redis, foi possível implementar um modelo de replicação de dados *P2P* que garante a consistência forte, obtida através do *framework* Treplica. Além disso, testes de *benchmarking* realizados com uma aplicação de comparação (Sentinel-Redis) mostraram que o desempenho foi reduzido em 8,4 a 12,4 vezes para garantir a consistência dos dados.

Não apenas trata-se de uma contribuição para as áreas de estudo de Bancos de Dados e Sistemas Distribuídos, na qual são apresentados resultados sobre a troca entre consistência e desempenho, como é possível expandir o uso do Redis para aplicações que necessitam de dados duráveis, e desejam obter alto desempenho, sem a necessidade de manter um segundo SGDB. Dentre as possibilidades de uso destaca-se:

- Casos nos quais a escrita dos dados deve ser atômica, porém como uma alta carga de leitura, onde aceita-se leituras regulares para maior desempenho. Por exemplo em redes-sociais, nas quais os dados armazenados precisam ser consistentes, existe uma alta carga de leitura (por milhões de usuários simultâneos) mas estes podem ler dados com atraso.

- Casos nos quais a escrita não precisa ser confirmada, porém a leitura do dado precisa ser consistente. Como por exemplo em aplicações de *IoT* ao coletar dados de ambiente (dados meteorológicos, dados geoespaciais, etc) nas quais a quantidade de dados escritos é muito grande, e a eventual perda de um destes não é relevante, mas as leituras precisam ser precisas pois guiam a tomada de decisão.

- Casos nos quais há diversos nós geo-localizados, e tanto a leitura quanto a escrita devem ser atômicas, optando por uma consistência forte frente ao desempenho. Como por exemplo em aplicações bancárias.

7.1 Trabalhos futuros

Este trabalho expande o uso do *framework* Treplica, dando ao Redis propriedades de replicação *P2P* e uma forte consistência de dados. Estas propriedades porém possuem um custo de desempenho. Este custo de desempenho porém não deve ser igual para todos os SGBDs, e propõe-se trabalhos futuros que avaliem o desempenho de outros SGBDs *NoSQL*. Assim, será possível identificar cenários em que o *trade-off* entre estas propriedades tenha uma relação adequada de custo-benefício.

Atrelado à sugestão de trabalhos futuros, testes mais complexos poderiam ser feitos, buscando simular tráfego real de aplicações, incluindo na carga do teste transações com operações de leitura, bem como realizando uma mescla entre transações atômicas e regulares.

Por fim, sugere-se validar a ocorrência de inconsistência dos dados, por meio da injeção falhas e observação da diferença entre o estado final dos bancos de dados. Assim, quantificando a ocorrência de inconsistências, será possível de permitir ao usuário do Treplica-Redis não só compreender o quanto de desempenho se está sacrificando, mas também o quanto de consistência se está adquirindo.

Referências

ANDERSON, E. et al. What consistency does your key-value store actually provide? In: *Proceedings of the Sixth International Conference on Hot Topics in System Dependability*. Vancouver, BC, Canada: USENIX Association, 2010. p. 1–16. Citado 2 vezes nas páginas 21 e 31.

APACHE JMeter - Apache JMeter™. 1998. Accessed on 21 Jan. 2023. Disponível em: <https://jmeter.apache.org/>. Citado na página 63.

Apache Software Foundation. *Performance tuning Apache Tomcat*. 2018. <https://tomcat.apache.org/articles/performance.pdf>. [Online; accessed 9-April-2023]. Citado na página 68.

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: Association for Computing Machinery, 2012. p. 53–64. Citado na página 21.

BAGUI, S.; NGUYEN, L. T. Database sharding: to provide fault tolerance and scalability of big data on the cloud. *International Journal of Cloud Applications and Computing*, IGI Global, p. 36–52, 2015. Citado na página 29.

BANGERT, B. *Redis Tools*. 2011. Github.com/bbangert/retools. Accessed in Jun-14-2021. Citado na página 37.

BONIFATI, A. et al. Distributed databases and peer-to-peer databases: past and present. *ACM SIGMOD Record*, p. 5–11, 2008. Citado 4 vezes nas páginas 22, 23, 25 e 29.

BOTROS, P. *River*. 2021. Github.com/pbotros/river. Accessed in Jun-13-2021. Citado na página 36.

BOULTON, C. *PHP-Resque*. 2017. Github.com/chrisboulton/php-resque. Accessed in Jun-17-2021. Citado na página 37.

BREWER, E. Towards robust distributed systems. In: *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing*. New York, NY, USA: Association for Computing Machinery, 2000. p. 7. Citado 3 vezes nas páginas 25, 26 e 39.

BUZATO, L. E.; VIEIRA, G. M. D.; ZWAENEPOEL, W. Dynamic content web applications: Crash, failover, and recovery analysis. In: *2009 IEEE/IFIP International Conference on Dependable Systems & Networks*. [S.l.: s.n.], 2009. p. 229–238. Citado na página 43.

CARLSON, J. *Redis in action*. [S.l.]: Simon and Schuster, 2013. Citado 3 vezes nas páginas 24, 35 e 39.

CARLSON, J. *RPQueue*. 2015. Github.com/josiahcarlson/rpqueue. Accessed in Jun-14-2021. Citado na página 37.

CELERY. *Celery*. 2009. Github.com/celery/celery. Accessed in Jun-12-2021. Citado na página 37.

CHEN, S. et al. Towards scalable and reliable in-memory storage system: A case study with redis. In: IEEE. *2016 IEEE Trustcom/BigDataSE/ISPA*. [S.l.], 2016. p. 1660–1667. Citado 2 vezes nas páginas 40 e 41.

CHEPRASOV, A. *php-redis-lock*. 2016. Github.com/cheprasov/php-redis-lock. Accessed in Jun-17-2021. Citado na página 36.

CODISLABS. *codis*. 2020. Github.com/CodisLabs/codis. Accessed in Jun-13-2021. Citado na página 36.

COMMUNITY, R. *Rq*. 2012. Github.com/rq/rq. Accessed in Jun-11-2021. Citado na página 37.

DOCKER, I. Docker. *línea].[Junio de 2017]. Disponible en: https://www. docker. com/what-docker*, 2020. Citado na página 62.

ELMASRI, R.; NAVATHE, S. B. *Sistemas de banco de dados*. São Paulo, SP, Brasil: Pearson Addison Wesley, 2005. Citado 4 vezes nas páginas 22, 30, 32 e 52.

GEORGE, M. *Pyres*. 2009. Github.com/binarymatt/pyres. Accessed in Jun-15-2021. Citado na página 37.

GRAY, J.; REUTER, A. *Transaction processing: Concepts and techniques*. [S.l.]: Morgan Kaufmann, 1993. Citado na página 32.

GRIBBLE, S. et al. What can database do for peer-to-peer? In: *Proceedings of Web and Databases*. [S.l.: s.n.], 2001. p. 31–36. Citado 2 vezes nas páginas 22 e 29.

GUERREIRO, C. *rdb-parser*. 2012. Github.com/pconstr/rdb-parser. Accessed in Jun-13-2021. Citado na página 36.

GUERREIRO, C. *redis-sync*. 2012. Github.com/pconstr/redis-sync. Accessed in Jun-11-2021. Citado na página 37.

HAN, J. et al. Survey on nosql database. In: IEEE. *Proceedings of the 2011 6th International Conference on Pervasive Computing and Applications*. [S.l.], 2011. p. 363–366. Citado na página 26.

HUAWEI. *Proxy Cluster Redis*. 2020. Support.huaweicloud.com/intl/en-us/productdesc-dcs/CacheCluster.html. Accessed in Jun-13-2021. Citado na página 36.

IENAGA, T. *RedisPlugin*. 2016. Github.com/ienaga/RedisPlugin. Accessed in Jun-11-2021. Citado na página 36.

KABAKUS, A. T.; KARA, R. A performance evaluation of in-memory databases. *Journal of King Saud University - Computer and Information Sciences*, p. 520–525, 2016. Citado 2 vezes nas páginas 21 e 34.

KAMARUZZAMAN. *Top 10 Databases to Use in 2021*. 2021. Towardsdatascience.com/top-10-databases-to-use-in-2021-d7e6a85402ba. Accessed in Ago-17-2021. Citado 2 vezes nas páginas 21 e 23.

KEWANG. *Hedis*. 2010. Github.com/hedisdb/hedis. Accessed in Jun-17-2021. Citado na página 36.

KINGSBURY, K. *Jepsen: Redis*. 2013. Accessed on 21 Jan. 2023. Disponível em: <https://aphyr.com/posts/313-jepsen-redis>. Citado na página 63.

KLEPPMANN, M. *Designing Data-Intensive Applications*. [S.l.]: O'Reilly Media, Inc., 2017. Citado na página 38.

KUMAR, P.; PABREJA, K. Nosql databases: a step to database scalability in web environment. *International Journal of Advanced Research in Computer Science*, IJARCS, v. 4, n. 5, p. 121–126, 2013. Citado na página 27.

LABS, R. *Multiple Active*. 2018. Docs.redislabs.com/latest/rs/administering/designing-production/networking/multiple-active-proxy/. Accessed in Jun-15-2021. Citado na página 36.

LABS, R. *Active Active Geo-Distribution*. 2019. Redislabs.com/redis-enterprise/technology/active-active-geo-distribution/. Accessed in Jun-17-2021. Citado na página 36.

LABS, R. *Redis Cluster Proxy*. 2020. Github.com/RedisLabs/redis-cluster-proxy. Accessed in Jun-13-2021. Citado na página 36.

LAMPORT, L. On interprocess communication-part i: Basic formalism, part ii: Algorithms. *Distributed Computing. Also appeared as SRC Research Report 8.*, p. 77–101, 1985. Citado 3 vezes nas páginas 25, 31 e 52.

LAMPORT, L. The part-time parliament. *ACM Transactions on Computer Systems (TOCS)*, ACM, v. 16, n. 2, p. 133–169, 1998. Citado na página 33.

LAMPORT, L. The part-time parliament. *ACM Transactions on Computer Systems*, Association for Computing Machinery, New York, NY, USA, p. 133–169, 1998. Citado 2 vezes nas páginas 33 e 45.

LAVIN, J. *Redis: A Practical Guide*. [S.l.]: O'Reilly Media, Inc., 2012. Citado na página 35.

LEIFER, C. *huey*. 2011. Github.com/coleifer/huey. Accessed in Jun-16-2021. Citado 2 vezes nas páginas 36 e 37.

LEIFER, C. *Walrus*. 2018. Github.com/coleifer/walrus. Accessed in Jun-13-2021. Citado na página 36.

LI, Y.; MANOHARAN, S. A performance comparison of SQL and NoSQL databases. In: *Proceedings of IEEE Pacific RIM Conference on Communications, Computers, and Signal Processing*. [S.l.: s.n.], 2013. p. 15–19. Citado na página 21.

LIBRELIST.COM. *Resque*. 2009. Github.com/resque/resque. Accessed in Jun-14-2021. Citado na página 37.

LIMITED, J. I. *Predixy*. 2018. Github.com/joyieldInc/predixy. Accessed in Jun-15-2021. Citado na página 36.

LIU, Y. Consistency and failover in redis sentinel. *Journal of Computer Science and Technology*, Springer, v. 31, n. 3, p. 426–437, 2016. Citado na página 63.

LYNCH, N. *Distributed algorithms*. San Francisco, CA, USA: Morgan Kaufmann, 1996. Citado 2 vezes nas páginas 44 e 46.

MA, K.; YANG, B. Stream-based live data replication approach of in-memory cache. *Concurrency and Computation: Practice and Experience*, 2016. Citado na página 22.

MACEDO, T.; CALHEIROS, D. *Redis Cookbook*. [S.l.]: Packt Publishing Ltd., 2013. Citado 5 vezes nas páginas 35, 37, 38, 40 e 41.

MARTENS, M. *Ost*. 2011. Github.com/soveran/ost. Accessed in Jun-13-2021. Citado na página 37.

MEHTA, A. *Proxy Service*. 2020. Github.com/green-branch/proxy-redis. Accessed in Jun-13-2021. Citado 2 vezes nas páginas 23 e 36.

MORRIS, R. Scatter storage techniques. *Communications of the ACM*, New York, NY, USA, p. 38–44, 1968. Citado na página 33.

OAKS, S.; WONG, H. *Java threads*. [S.l.]: O'Reilly Media, Inc., 1999. Citado na página 55.

ONGARO, D.; OUSTERHOUT, J. In search of an understandable consensus algorithm. *ACM Transactions on Computer Systems*, ACM, v. 32, n. 4, p. 1–48, 2014. Citado na página 33.

OZSU, M. T.; VALDURIEZ, P. *Principles of distributed database systems*. 3rd. ed. [S.l.]: Springer, 2011. Citado 2 vezes nas páginas 27 e 32.

PAKSULA, M. *ActiveRedis*. 2010. Github.com/matti/activeredis. Accessed in Jun-17-2021. Citado na página 36.

PAKSULA, M. *Persisting objects in redis key-value database*. [S.l.], 2010. Available in cs.helsinki.fi/u/paksula/misc/redis.pdf. Accessed in Mar-13-2021. Citado 4 vezes nas páginas 21, 25, 33 e 38.

PIERCE, E.; ALVISI, L. *A recipe for atomic semantics for byzantine quorum systems*. [S.l.], 2000. Available in cs.utexas.edu/users/lorenzo/papers/byzatomic.ps. Accessed in Mar-13-2021. Citado 2 vezes nas páginas 25 e 31.

PRITCHETT, D. Base: An acid alternative. *ACM Queue*, ACM, v. 6, n. 3, p. 48–55, 2008. Citado 2 vezes nas páginas 27 e 31.

PROJECT, E. *Envoy*. 2016. Envoy-proxy.io/docs/envoy/latest/intro/arch_overview/other_protocols/redis. Accessed in Jun-15-2021. Citado na página 36.

REDIS. *Higher level libraries and tools*. 2020. Redis.io/clients/. Accessed in Jun-13-2021. Citado 2 vezes nas páginas 35 e 36.

REDIS. *Redis Persistence*. 2020. Redis.io/topics/persistence. Accessed in Mar-13-2021. Citado na página 21.

REDIS. *Terminology in Redis Enterprise Software*. 2020. Docs.redislabs.com/latest/rs/concepts/terminology/. Accessed in Jun-13-2021. Citado 2 vezes nas páginas 23 e 36.

SADALAGE, P.; FOWLER, M. *NoSQL distilled: a brief guide to the emerging world of polyglot persistence*. [S.l.]: Pearson Education, 2013. Citado 9 vezes nas páginas 11, 21, 25, 28, 29, 30, 33, 34 e 39.

SCHNEIDER, F. B. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys (CSUR)*, ACM, v. 22, n. 4, p. 299–319, 1990. Citado 3 vezes nas páginas 22, 25 e 33.

SHARMA, S. *Mastering microservices with java: Build enterprise microservices with Spring Boot 2.0, Spring Cloud, and Angular*. [S.l.]: Packt Publishing Ltd, 2019. Citado na página 50.

SINGH, R. P.; KHAN, A. N. A comprehensive study of load testing tools. In: IEEE. *2017 3rd International Conference on Computational Intelligence & Communication Technology (CICT)*. [S.l.], 2017. p. 1–5. Citado na página 61.

SOUZA, W. P. d. C. *Desempenho e Consistência de Banco de Dados Chave-Valor Persistentes*. Sorocaba, SP: [s.n.], 2022. Trabalho de Conclusão de Curso. Orientado pelo Prof. Me. Fernando Cesar Miranda e coorientado pelo Prof. Dr. Gustavo Maciel Dias Vieira. Citado 2 vezes nas páginas 62 e 67.

STONEBRAKER, M. et al. Scidb: A database management system for applications with complex analytics. *ACM SIGMOD Record*, ACM, v. 39, n. 4, p. 1–12, 2010. Citado na página 25.

TAN, K.-L. et al. In-memory databases: Challenges and opportunities from software and hardware perspectives. *ACM SIGMOD Record*, New York, NY, USA, p. 35–40, 2015. Citado 3 vezes nas páginas 21, 25 e 34.

TANENBAUM, A. S.; STEEN, M. V. *Distributed Systems: Principles and Paradigms*. Upper Saddle River, NJ, USA: Prentice Hall, 2006. Citado na página 46.

VIEIRA, G.; BUZATO, L. The performance of paxos and fast paxos. 08 2013. Citado na página 43.

VIEIRA, G. M.; BUZATO, L. E. Treplica: ubiquitous replication. In: CITESEER. *SBRC'08: Proc. of the 26th Brazilian Symposium on Computer Networks and Distributed Systems*. [S.l.], 2008. p. 25. Citado na página 43.

VIEIRA, G. M.; BUZATO, L. E. Implementation of an object-oriented specification for active replication using consensus. *Technical Report IC-10–26*, 2010. Citado 4 vezes nas páginas 23, 25, 33 e 43.

VOGELS, W. Eventually consistent. *Communications of the ACM*, ACM, v. 52, n. 1, p. 40–44, 2008. Citado na página 27.

WESTCOTT, M. *Runnel*. 2020. Github.com/mjwestcott/runnel. Accessed in Jun-14-2021. Citado 2 vezes nas páginas 36 e 37.

WILLIAMS, T. et al. 1 gnuplot. 2004. Citado na página 66.

WUJUNWEI, A. *redlock*. 2020. Github.com/wujunwei/redlock. Accessed in Jun-16-2021. Citado na página 36.

A Documentação

O código-fonte da aplicação é disponibilizado a partir do repositório privado disponível em *https://bitbucket.org/danilopereira10/treplica-redis*.

A.1 Dependências

A aplicação Java tem as seguintes dependências:

Tabela 4 – Tabela de dependências

Dependência	**Versão**	**Fonte**
Java	1.8	https://www.java.com/pt-BR/download/
Maven	3.8.6	https://maven.apache.org/download.cgi
Treplica	2.0.0	https://bitbucket.org/gdvieira/treplica/
Springboot	1.5.9	https://mvnrepository.com/artifact/org.springframework .boot/spring-boot-starter-parent/1.5.9.RELEASE
Swagger	2.9.2	https://mvnrepository.com/artifact/io.springfox/ springfox-swagger-ui/2.9.2
Maven Jar Plugin	3.2.0	https://mvnrepository.com/artifact/org.apache.maven .plugins/maven-jar-plugin/3.2.0
Lombok	1.18.20	https://mvnrepository.com/artifact/org.projectlombok /lombok/1.18.20
JAXB API	2.4.0	https://mvnrepository.com/artifact/javax.xml.bind /jaxb-api/2.4.0-b180830.0359

Além das dependências listadas, é necessário também adicionar as demais dependências requeridas pelo Springboot e Treplica.

A.2 Uso

Após ter todas as bibliotecas instaladas, é possível compilar a aplicação por meio do código *mvn clean install*. Isto irá gerar um arquivo executável .JAR.

As configurações da aplicação são definidas por meio de variáveis de ambientes enviadas à JVM, sendo:

Tabela 5 – Tabela de variáveis de ambiente

Argumento	**Padrão**	**Descrição**
DISK		[**Requerido**] Caminho de uma pasta vazia à qual tem acesso a instância do aplicativo a ser executado. Esse caminho deve ser exclusivo para cada instância
SS_TRIGGER	1000	Define o número de transações para criar um instantâneo
REDIS_HOST	127.0.0.1	Endereço IP de uma instância do servidor Redis
REDIS_PORT	6379	Porta da instância do servidor Redis
REDIS_PASS		Senha da instância do servidor Redis.
REDIS_EMBED	true	Um booleano que define se deve ser usada uma instância do Redis Server incorporada (embed) ou instância independente
CLEAN_INIT	false	Limpa o diretório inicial na inicialização, inclui instantâneos, alterações e arquivo .rdb
SERVER_PORT	8080	Define a porta local que o servidor REST do Treplica-Redis deve ser executado
WATCHER	true	Este booleano define se o sistema de monitoramento Treplica-Redis (chamado Watcher) deve ser usado ou não. Este sistema detecta falhas e é responsável por sua recuperação
WATCHER_TIME	5000	Intervalo em milissegundos no qual o Watcher executa uma verificação
WATCHER_FAIL_BEHAVIOR	DESTROY	Comportamento esperado quando o Watcher detecta uma falha. **DESTROY**: destrói e tenta reiniciar o processo. **RESTART**: tenta reiniciar o servidor Redis usando o servidor Redis incorporado
TREPLICA_PROCESSES	1	Número de nós Treplica-Redis
TREPLICA_FAST_PAXOS	false	Este booleano define o algoritmo a ser usado, sendo FastPaxos (**true**) ou Paxos (**false**)
TREPLICA_ROUND_TRIP	200	Tempo limite sobre qualquer tentativa de consenso Paxos
LOG	ERROR	Define o nível de login do SpringBoot, deve ser **INFO** ou **ERROR**

Tendo as variáveis definidas, basta executar o comando *java -jar <PATH_TO_FILE>*, tal que *PATH_TO_FILE* é o caminho para o arquivo .JAR gerado pelo Maven. Com isto,

a aplicação será executada e será possível enviar requisições HTTP ao servidor lançado.

B Resultados dos testes

Os testes são divididos da seguinte maneira: O teste 1 compreende a escrita do arquivo *JSON* de 4 bytes. O teste 2 refere-se a escrita do arquivo *JSON* de 350 bytes. O teste 3 é referente a escrita do arquivo *JSON* de 1,4 kilobytes. E, por fim, o teste 4 referente ao arquivo *JSON* de 3,5 kilobytes.

A seguir serão apresentados testes referentes à aplicação Treplica-Redis. Observe que as Figuras 16, 19, 22 e 25 representam as transações iniciadas e finalizadas por segundo, em cada um dos testes. Já as figuras 17, 20, 23 e 26 representam o tempo médio de resposta em cada segundo. Em todos os testes é possível perceber que tempo de resposta na aplicação Treplica-Redis é marcado por picos de latência. Isto ocorre principalmente pois uma das estratégias do Treplica é agrupar um conjunto de transações como um lote e ordenar todo este lote, podendo criar variações no tempo de resposta em prol de ampliar o desempenho do algoritmo de ordenação.

```
1 Time span (s): 121
2 Total operations: 25885
3 Operations started (op/s):
4     Average: 213.92561983471074 Stdev: 70.371770534457    Median:
        152.0
5 Operations completed (op/s):
6     Average: 213.92561983471074 Stdev: 63.301943794438614
        Median: 152.0
7 Response time (ms):
8     Average: 1750.0887769471626 Stdev: 723.3842591210818    Median
        : 2255.2105263157896
```

Figura 15 – Treplica-Redis: Resumo do teste 1

Na sequência serão apresentados os resultados consolidados da aplicação Sentinel-Redis, com o tempo médio das respostas sobreposto às transações iniciadas e finalizadas, como por ser visto nas Figuras 27, 28, 29 e 30

Figura 16 – Treplica-Redis: Transações iniciadas e finalizadas por segundo - teste 1

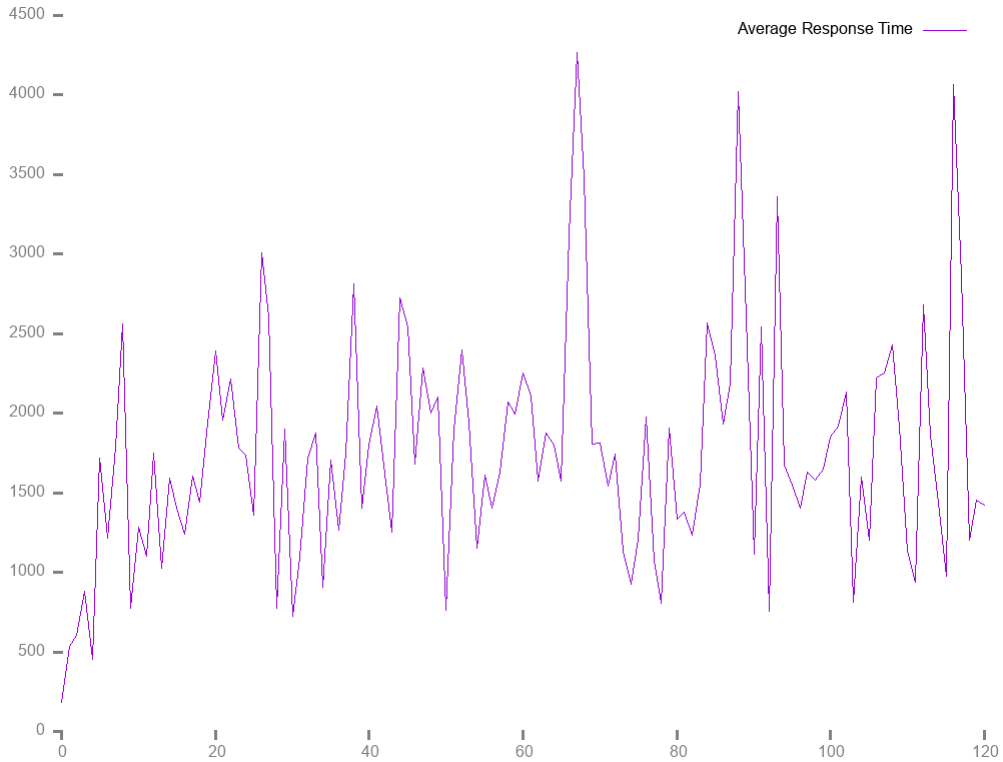

Figura 17 – Treplica-Redis: Tempo médio de resposta por segundo - teste 1

```
1 Time span (s): 121
2 Total operations: 12217
3 Operations started (op/s):
4     Average: 100.96694214876032 Stdev: 41.63130510491226    Median
          : 81.0
5 Operations completed (op/s):
6     Average: 100.96694214876032 Stdev: 33.51192173050697    Median
          : 81.0
7 Response time (ms):
8     Average: 3704.3274403667615 Stdev: 1258.5541473027224
          Median: 4400.555555555556
```

Figura 18 – Treplica-Redis: Resumo do teste 2

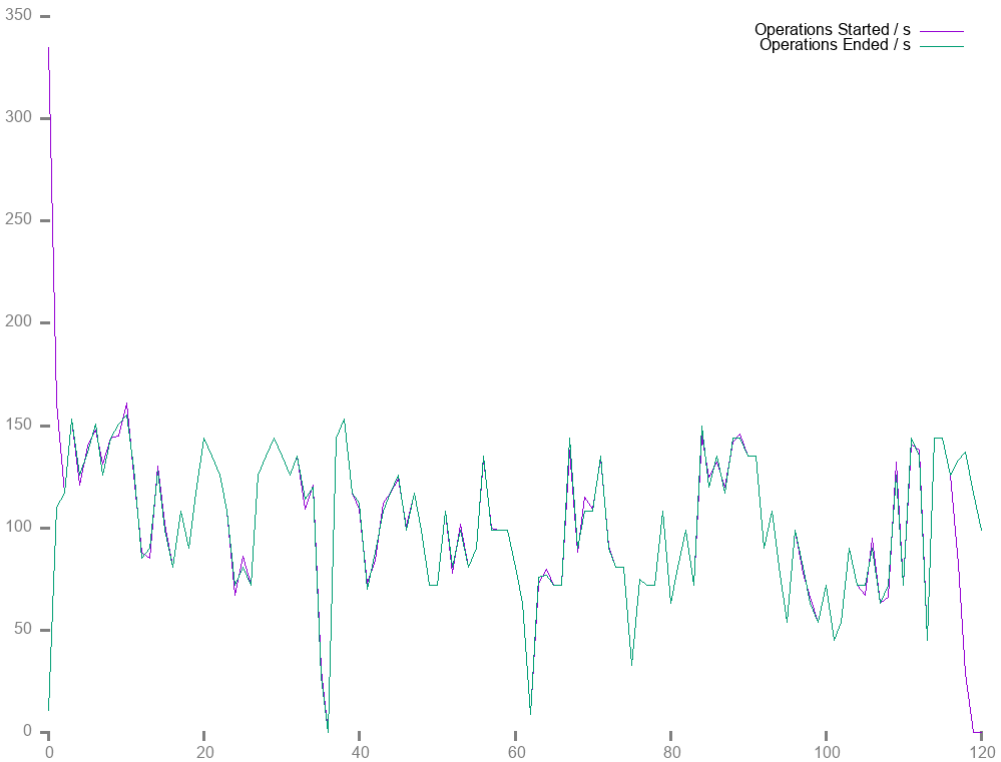

Figura 19 – Treplica-Redis: Transações iniciadas e finalizadas por segundo - teste 2

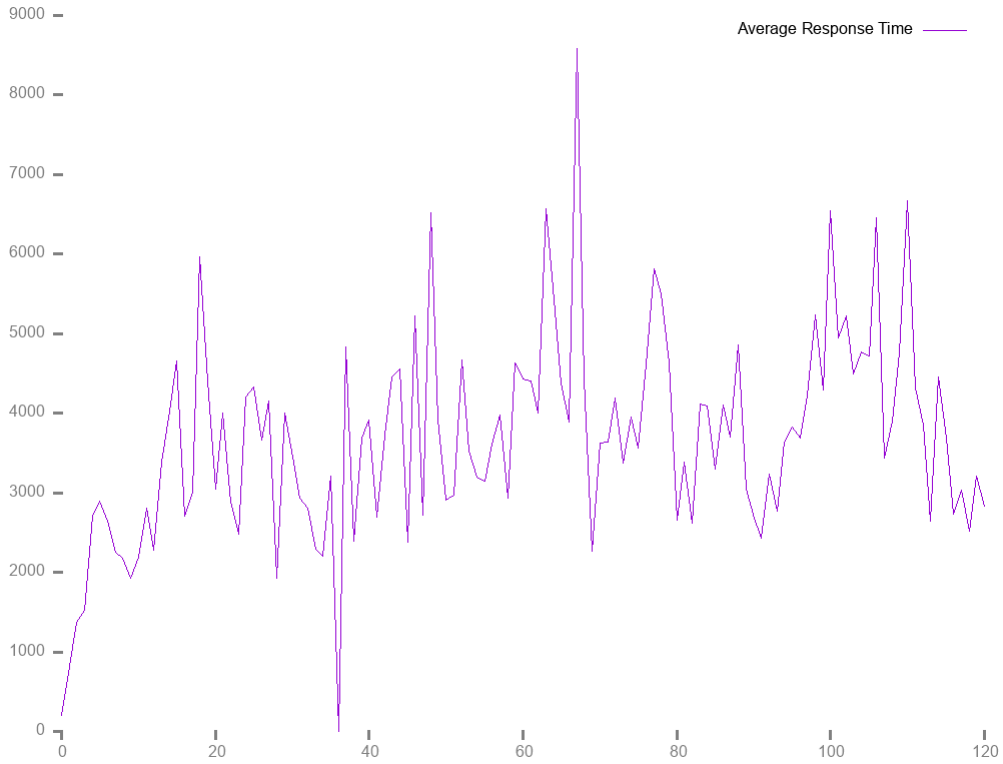

Figura 20 – Treplica-Redis: Tempo médio de resposta por segundo - teste 2

```
1 Time span (s): 121
2 Total operations: 9644
3 Operations started (op/s):
4     Average: 79.70247933884298 Stdev: 39.47628921857172    Median:
          60.0
5 Operations completed (op/s):
6     Average: 79.70247933884298 Stdev: 21.438145375358065    Median
          : 56.0
7 Response time (ms):
8     Average: 4526.409244733284 Stdev: 1719.9927620187636    Median
          : 4711.625
```

Figura 21 – Treplica-Redis: Resumo do teste 3

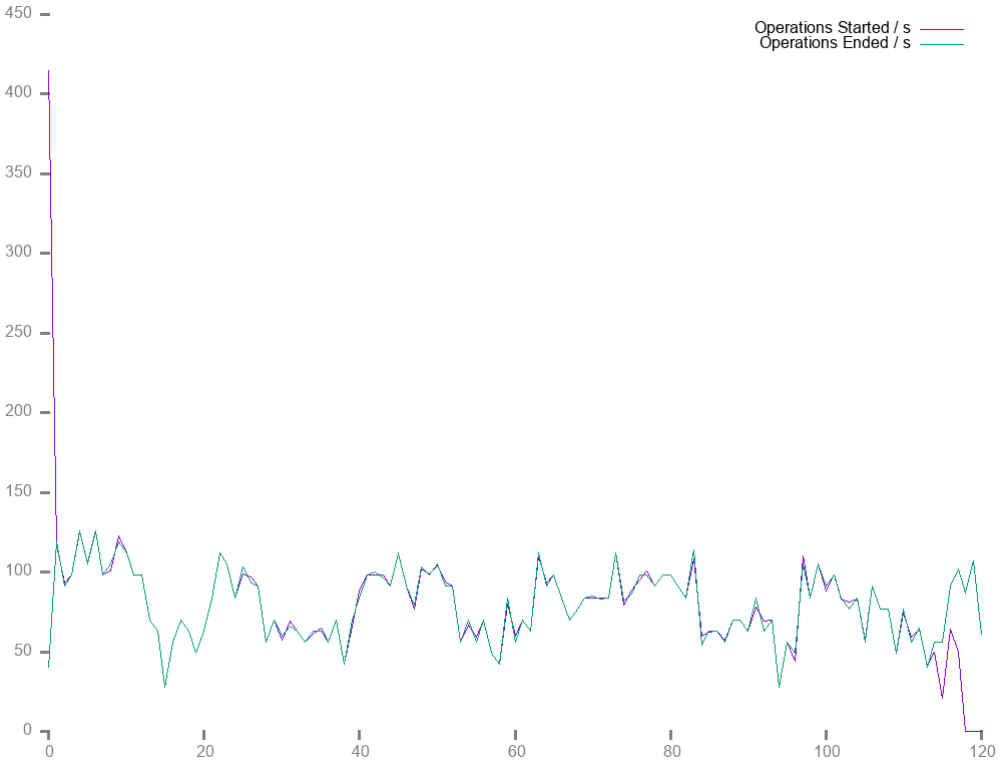

Figura 22 – Treplica-Redis: Transações iniciadas e finalizadas por segundo - teste 3

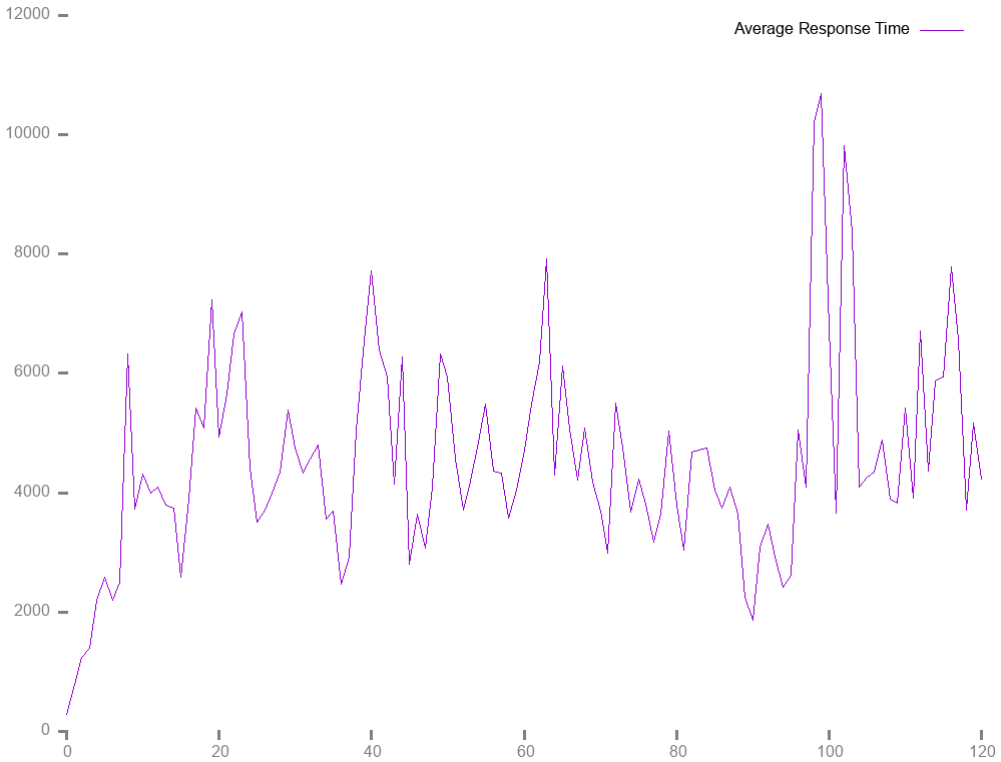

Figura 23 – Treplica-Redis: Tempo médio de resposta por segundo - teste 3

```
1 Time span (s): 121
2 Total operations: 3972
3 Operations started (op/s):
4     Average: 32.82644628099174 Stdev: 34.5754435223301    Median:
        35.0
5 Operations completed (op/s):
6     Average: 32.82644628099174 Stdev: 10.266188586772287    Median
        : 36.0
7 Response time (ms):
8     Average: 11109.181203022723 Stdev: 3722.114754346462    Median
        : 9887.472222222223
```

Figura 24 – Treplica-Redis: Resumo do teste 4

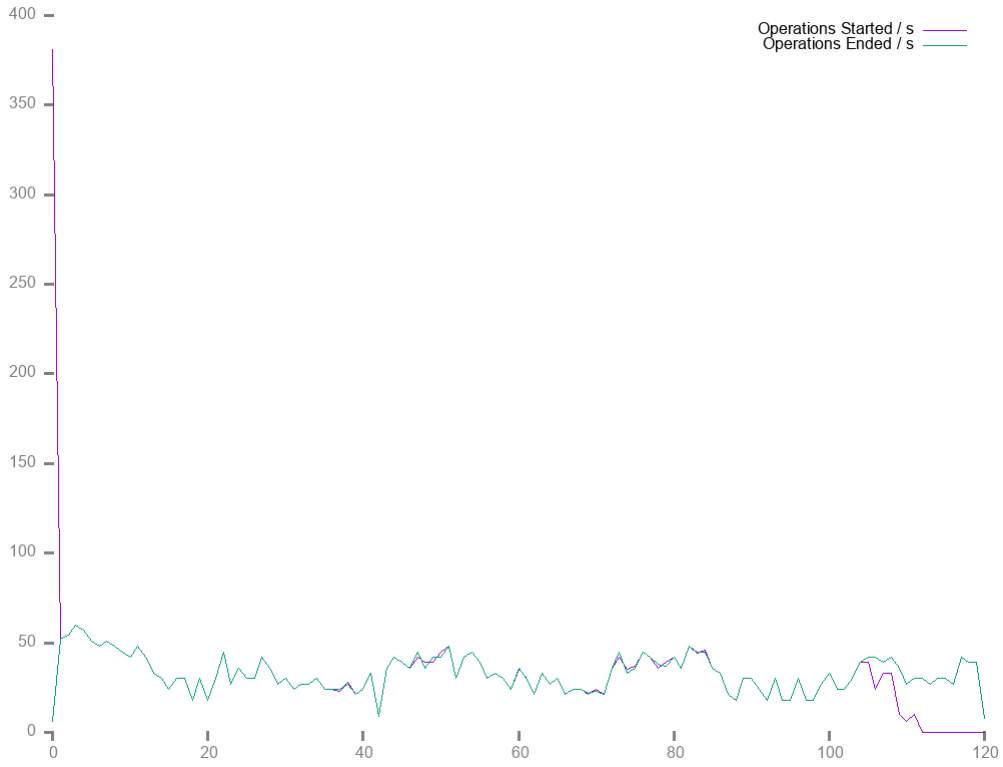

Figura 25 – Treplica-Redis: Transações iniciadas e finalizadas por segundo - teste 4

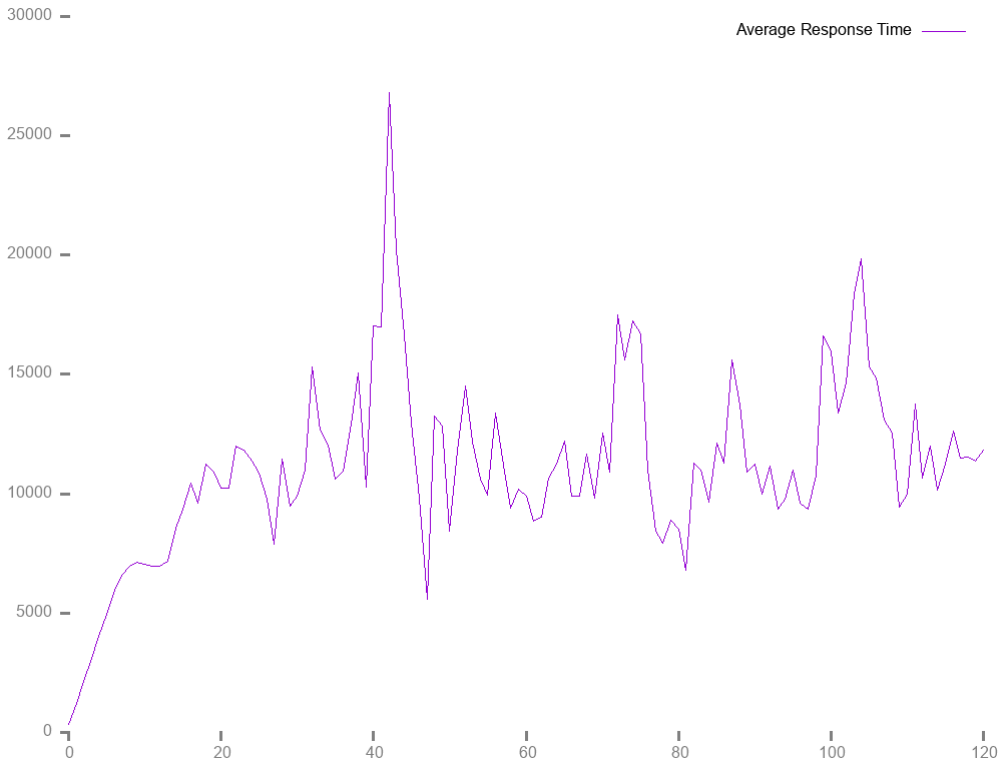

Figura 26 – Treplica-Redis: Tempo médio de resposta por segundo - teste 4

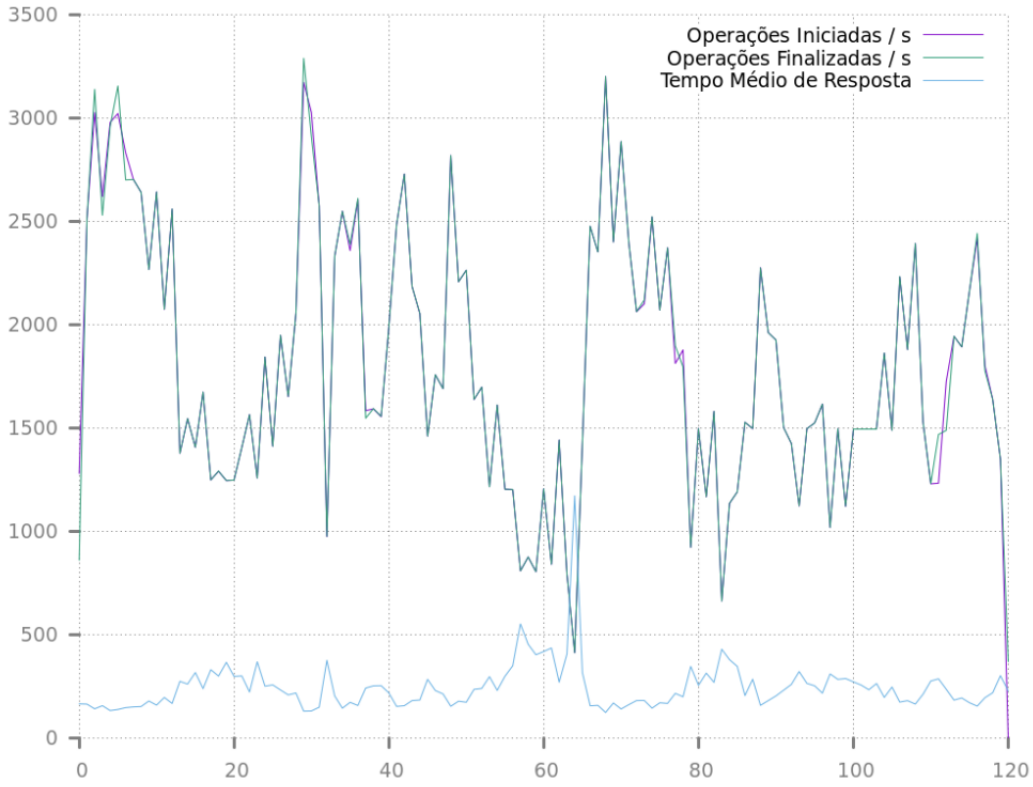

Figura 27 – Sentinel-Redis: Transações iniciadas, transações finalizadas e tempo médio de respostas - teste 1

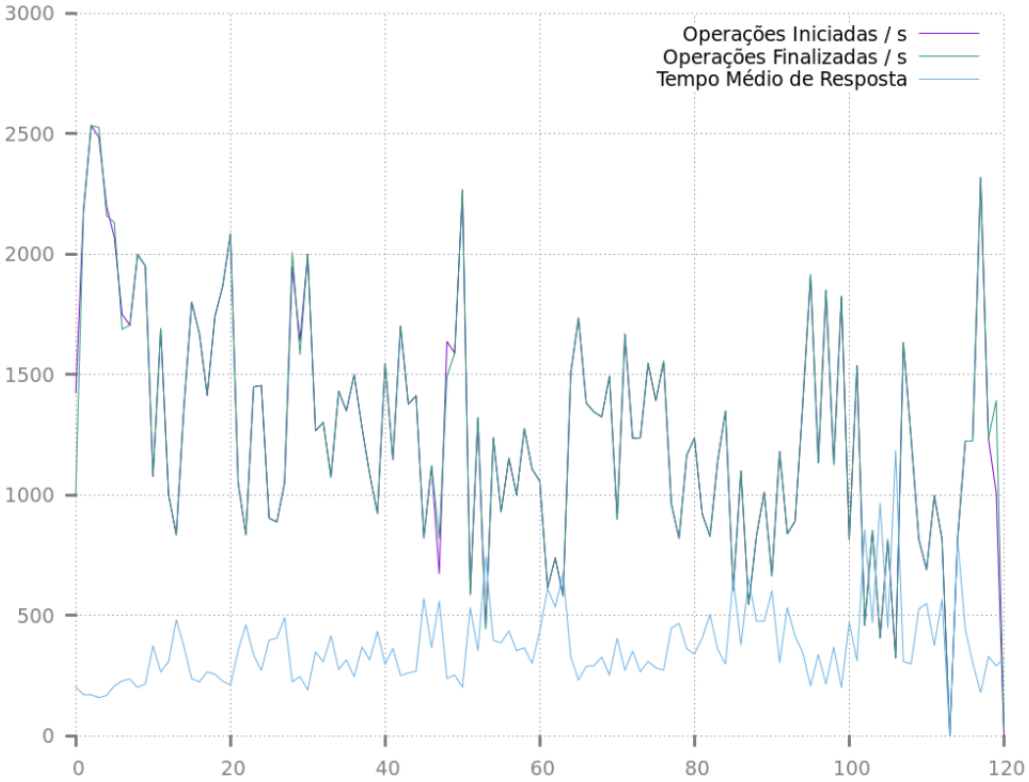

Figura 28 – Sentinel-Redis: Transações iniciadas, transações finalizadas e tempo médio de respostas - teste 2

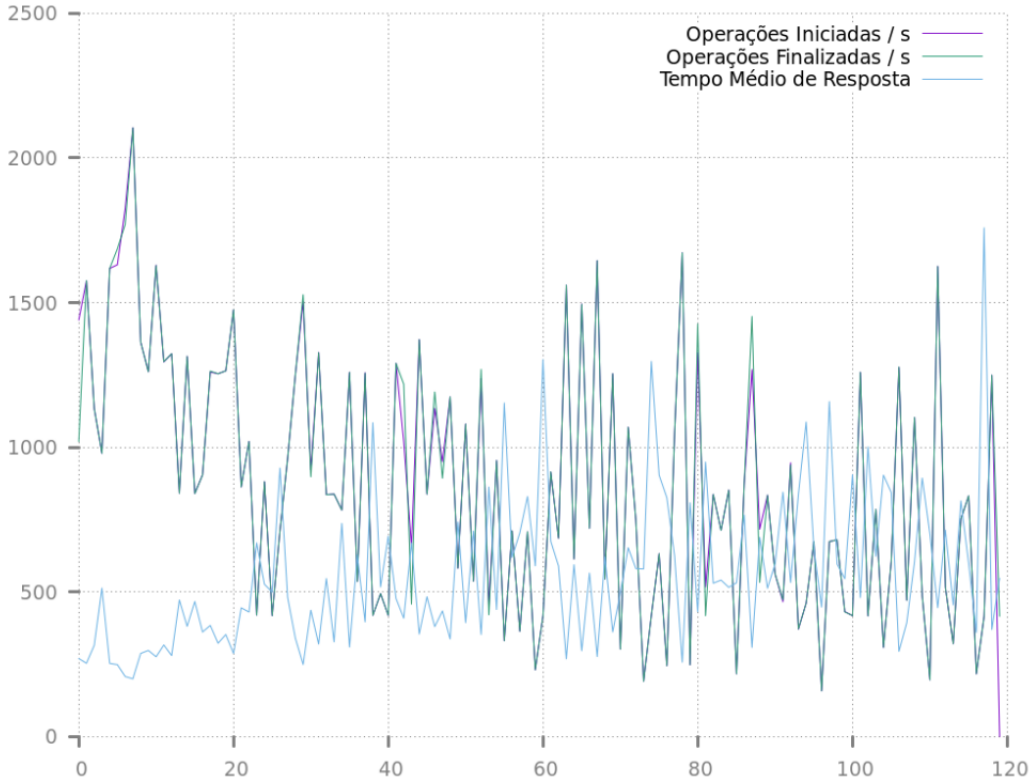

Figura 29 – Sentinel-Redis: Transações iniciadas, transações finalizadas e tempo médio de respostas - teste 3

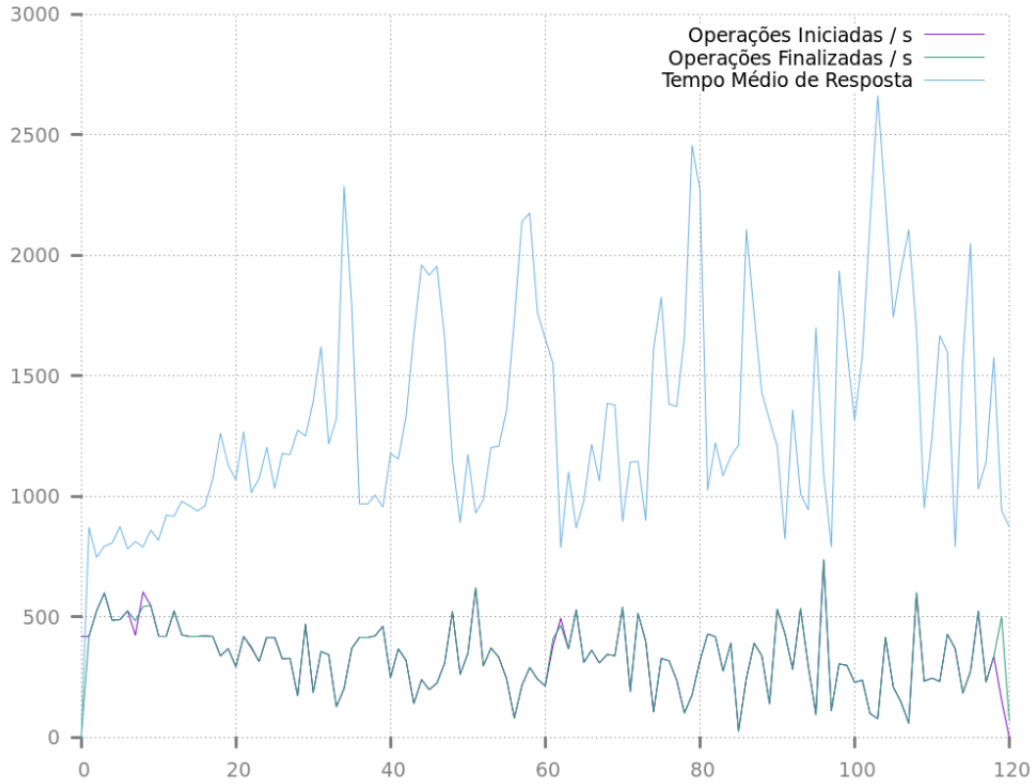

Figura 30 – Sentinel-Redis: Transações iniciadas, transações finalizadas e tempo médio de respostas - teste 4

C JSONs utilizados no teste

A seguir são apresentados os arquivos que compõe os testes apresentados no Capítulo 6. São quatro arquivos de diferentes tamanhos, que são persistidos a cada execução do teste.

Arquivo 1 - Tamanho 4 bytes:

```
{}
```

Arquivo 2 - Tamanho 350 bytes:

```
{
    "sclr": 0,
    "str": "b",
    "sub_doc": {
        "sclr": 10,
        "str": "c",
        "arr": [
            1,
            2,
            3,
            {
                "sclr": 20,
                "str": "d"
            }
        ]
    },
    "array_of_docs": [
        -1,
        {
            "sclr": 11,
            "str": "e",
            "arr": [
                4,
                5,
                6,
                {
                    "sclr": 21,
                    "str": "f"
```

```
                    }
                ]
            },
            {
                "sclr": 12,
                "str": "g",
                "arr": [
                    7,
                    8,
                    9,
                    {
                        "sclr": 22,
                        "str": "h"
                    }
                ]
            },
            -2
        ]
}
```

Arquivo 3 - Tamanho 1,4 kilobytes:

```
    {
        "object with members": [
            "array with this element",
            10,
            22,
            9.112,
            null
        ],
        "JSON Test Pattern pass1": {
            "integer": 1234567890,
            "real": -9876.543210,
            "e": 1.23456789,
            "E": 1.234567890,
            ".": 2.3456789012,
            "zero": 0,
            "one": 1,
            "underscore": "_",
            "data1": "Hey, a data here",
```

```
            "data2": "Hey, another data here",
            "data3": 1,
            "data4": 9.8,
            "data5": true,
            "alpha": "abcdefghijklmnopqrstuvwyz",
            "ALPHA": "ABCDEFGHIJKLMNOPQRSTUVWYZ",
            "digit": "0123456789",
            "0123456789": "digit",
            "special": "1~!@#$%^&*()+-={:[,]}|;.</>?",
            "true": true,
            "false": false,
            "null": null,
            "array": [],
            "object": {},
            "address": "50 St. James Street",
            "url": "http://www.JSON.org/",
            "list": [
                1,
                2,
                3,
                4,
                5,
                6,
                7
            ],
            "compact": [
                1,
                2,
                3,
                4,
                5,
                6,
                7
            ]
        }
    }
```

Arquivo 4 - Tamanho 3,5 kilobytes:

```
{
```

```
{
    "web-app": {
        "servlet": [
            {
                "servlet-name": "cofaxCDS",
                "servlet-class": "org.cofax.cds.CDSServlet",
                "init-param": {
                    "configGlossary:installationAt": "
                      Philadelphia, PA",
                    "configGlossary:adminEmail": "ksm@pobox.com
                      ",
                    "configGlossary:poweredBy": "Cofax",
                    "configGlossary:poweredByIcon": "/images/
                      cofax.gif",
                    "configGlossary:staticPath": "/content/
                      static",
                    "templateProcessorClass": "org.cofax.
                      WysiwygTemplate",
                    "templateLoaderClass": "org.cofax.
                      FilesTemplateLoader",
                    "templatePath": "templates",
                    "templateOverridePath": "",
                    "defaultListTemplate": "listTemplate.htm",
                    "defaultFileTemplate": "articleTemplate.htm
                      ",
                    "useJSP": false,
                    "jspListTemplate": "listTemplate.jsp",
                    "jspFileTemplate": "articleTemplate.jsp",
                    "cachePackageTagsTrack": 200,
                    "cachePackageTagsStore": 200,
                    "cachePackageTagsRefresh": 60,
                    "cacheTemplatesTrack": 100,
                    "cacheTemplatesStore": 50,
                    "cacheTemplatesRefresh": 15,
                    "cachePagesTrack": 200,
                    "cachePagesStore": 100,
                    "cachePagesRefresh": 10,
                    "cachePagesDirtyRead": 10,
                    "searchEngineListTemplate": "
                      forSearchEnginesList.htm",
```

```
33                  "searchEngineFileTemplate": "
                      forSearchEngines.htm",
34                  "searchEngineRobotsDb": "WEB-INF/robots.db",
35                  "useDataStore": true,
36                  "dataStoreClass": "org.cofax.SqlDataStore",
37                  "redirectionClass": "org.cofax.
                      SqlRedirection",
38                  "dataStoreName": "cofax",
39                  "dataStoreDriver": "com.microsoft.jdbc.
                      sqlserver.SQLServerDriver",
40                  "dataStoreUrl": "jdbc:microsoft:sqlserver://
                      LOCALHOST:1433;DatabaseName=goon",
41                  "dataStoreUser": "sa",
42                  "dataStorePassword": "dataStoreTestQuery",
43                  "dataStoreTestQuery": "SET NOCOUNT ON;select
                       test='test';",
44                  "dataStoreLogFile": "/usr/local/tomcat/logs/
                      datastore.log",
45                  "dataStoreInitConns": 10,
46                  "dataStoreMaxConns": 100,
47                  "dataStoreConnUsageLimit": 100,
48                  "dataStoreLogLevel": "debug",
49                  "maxUrlLength": 500
50              }
51          },
52          {
53              "servlet-name": "cofaxEmail",
54              "servlet-class": "org.cofax.cds.EmailServlet",
55              "init-param": {
56                  "mailHost": "mail1",
57                  "mailHostOverride": "mail2"
58              }
59          },
60          {
61              "servlet-name": "cofaxAdmin",
62              "servlet-class": "org.cofax.cds.AdminServlet"
63          },
64          {
65              "servlet-name": "fileServlet",
```

```
66                  "servlet-class": "org.cofax.cds.FileServlet"
67              },
68              {
69                  "servlet-name": "cofaxTools",
70                  "servlet-class": "org.cofax.cms.
                      CofaxToolsServlet",
71                  "init-param": {
72                      "templatePath": "toolstemplates/",
73                      "log": 1,
74                      "logLocation": "/usr/local/tomcat/logs/
                          CofaxTools.log",
75                      "logMaxSize": "",
76                      "dataLog": 1,
77                      "dataLogLocation": "/usr/local/tomcat/logs/
                          dataLog.log",
78                      "dataLogMaxSize": "",
79                      "removePageCache": "/content/admin/remove?
                          cache=pages&id=",
80                      "removeTemplateCache": "/content/admin/
                          remove?cache=templates&id=",
81                      "fileTransferFolder": "/usr/local/tomcat/
                          webapps/content/fileTransferFolder",
82                      "lookInContext": 1,
83                      "adminGroupID": 4,
84                      "betaServer": true
85                  }
86              }
87          ],
88          "servlet-mapping": {
89              "cofaxCDS": "/",
90              "cofaxEmail": "/cofaxutil/aemail/*",
91              "cofaxAdmin": "/admin/*",
92              "fileServlet": "/static/*",
93              "cofaxTools": "/tools/*"
94          },
95          "taglib": {
96              "taglib-uri": "cofax.tld",
97              "taglib-location": "/WEB-INF/tlds/cofax.tld"
98          }
```

```
99          }
100 }
```

D Resultados dos histogramas

A seguir são apresentados os arquivos do histograma gerado para cada um dos arquivos de teste, apresentado em cada linha: Segundo, Transações iniciadas, Finalizadas e a Latência A seguir são apresentados os arquivos do histograma gerado para cada um dos arquivos de teste, apresentado em cada linha: Segundo, Transações iniciadas, Finalizadas e a Latência

Arquivo 1 - tamanho 4 bytes:

```
0 142.0 21.0 184.61904761904762
1 513.0 269.0 532.3791821561339
2 335.0 325.0 601.2338461538461
3 289.0 304.0 884.1282894736842
4 262.0 247.0 450.7368421052632
5 110.0 114.0 1716.0701754385964
6 346.0 342.0 1214.312865497076
7 315.0 342.0 1784.8245614035088
8 296.0 279.0 2562.8064516129034
9 339.0 329.0 772.8966565349544
10 304.0 304.0 1282.1348684210527
11 253.0 264.0 1104.1174242424242
12 298.0 287.0 1750.2857142857142
13 264.0 266.0 1027.7067669172932
14 293.0 304.0 1591.0131578947369
15 211.0 209.0 1402.358851674641
16 258.0 247.0 1242.3846153846155
17 228.0 228.0 1603.377192982456
18 190.0 190.0 1442.8315789473684
19 152.0 152.0 1938.2631578947369
20 181.0 190.0 2390.452631578947
21 174.0 171.0 1957.888888888889
22 173.0 171.0 2217.4853801169593
23 175.0 171.0 1782.8538011695907
24 152.0 152.0 1737.4144736842106
25 183.0 190.0 1356.321052631579
26 166.0 177.0 3009.25988700565
27 267.0 260.0 2607.503846153846
28 220.0 209.0 771.066985645933
```

```
29 232.0 247.0 1900.8097165991903
30 318.0 304.0 723.0953947368421
31 261.0 266.0 1095.6165413533834
32 234.0 228.0 1719.25
33 294.0 304.0 1872.8026315789473
34 249.0 247.0 904.5668016194332
35 103.0 95.0 1703.4526315789474
36 197.0 209.0 1263.5741626794259
37 314.0 304.0 1747.421052631579
38 268.0 266.0 2810.785714285714
39 141.0 152.0 1401.3618421052631
40 201.0 190.0 1799.3
41 190.0 190.0 2045.6315789473683
42 181.0 190.0 1618.9894736842105
43 180.0 171.0 1254.7076023391812
44 152.0 171.0 2725.5789473684213
45 177.0 171.0 2543.1052631578946
46 165.0 152.0 1679.1513157894738
47 247.0 247.0 2283.425101214575
48 171.0 171.0 1999.842105263158
49 190.0 190.0 2104.3684210526317
50 217.0 228.0 762.311403508772
51 239.0 228.0 1903.0657894736842
52 175.0 190.0 2398.436842105263
53 319.0 304.0 1938.7763157894738
54 216.0 247.0 1148.6153846153845
55 297.0 266.0 1613.8233082706768
56 209.0 209.0 1403.3875598086124
57 171.0 171.0 1630.654970760234
58 209.0 209.0 2067.9521531100477
59 133.0 133.0 1997.8345864661653
60 152.0 152.0 2255.2105263157896
61 152.0 152.0 2108.0789473684213
62 155.0 165.0 1570.6
63 206.0 196.0 1874.9336734693877
64 168.0 171.0 1798.06432748538
65 149.0 152.0 1573.9276315789473
66 63.0 57.0 3120.3859649122805
67 55.0 76.0 4266.118421052632
```

```
68 284.0 266.0 3475.2556390977443
69 250.0 247.0 1803.7206477732793
70 304.0 304.0 1814.1184210526317
71 190.0 190.0 1542.878947368421
72 265.0 266.0 1742.4661654135339
73 269.0 285.0 1134.6947368421052
74 246.0 231.0 923.2164502164502
75 244.0 263.0 1209.2395437262358
76 268.0 248.0 1978.899193548387
77 285.0 284.0 1070.2640845070423
78 277.0 286.0 807.8846153846154
79 298.0 303.0 1907.5445544554455
80 238.0 228.0 1336.938596491228
81 174.0 190.0 1380.7894736842106
82 153.0 133.0 1235.2706766917292
83 180.0 190.0 1554.2947368421053
84 139.0 129.0 2568.201550387597
85 80.0 80.0 2359.3125
86 171.0 171.0 1932.6608187134502
87 180.0 190.0 2182.221052631579
88 86.0 76.0 4022.842105263158
89 152.0 152.0 2600.5855263157896
90 178.0 190.0 1114.278947368421
91 308.0 304.0 2539.7434210526317
92 244.0 247.0 758.0121457489879
93 296.0 285.0 3362.7754385964913
94 286.0 302.0 1673.9238410596026
95 215.0 211.0 1549.4028436018957
96 278.0 266.0 1406.4661654135339
97 214.0 228.0 1628.4254385964912
98 242.0 228.0 1580.298245614035
99 179.0 179.0 1649.3016759776535
100 131.0 155.0 1850.6451612903227
101 223.0 203.0 1920.6945812807883
102 246.0 242.0 2131.9214876033056
103 228.0 228.0 814.9561403508771
104 238.0 266.0 1596.1842105263158
105 280.0 266.0 1202.5112781954888
106 184.0 171.0 2221.46783625731
```

```
107 134.0 133.0 2252.3684210526317
108 209.0 209.0 2427.6124401913876
109 190.0 190.0 1911.3684210526317
110 190.0 190.0 1134.3631578947368
111 152.0 152.0 940.0526315789474
112 152.0 152.0 2683.3486842105262
113 154.0 160.0 1863.44375
114 196.0 201.0 1434.0597014925372
115 168.0 171.0 976.8421052631579
116 188.0 190.0 4064.436842105263
117 233.0 247.0 2859.9635627530365
118 179.0 228.0 1202.0263157894738
119 198.0 285.0 1453.378947368421
120 0.0 209.0 1425.4258373205741
```

Arquivo 2 - tamanho 350 bytes:

```
0 335.0 11.0 202.0909090909091
1 161.0 110.0 823.8727272727273
2 117.0 117.0 1371.5213675213674
3 153.0 153.0 1519.8235294117646
4 121.0 126.0 2709.3174603174602
5 140.0 137.0 2895.6642335766423
6 148.0 151.0 2646.456953642384
7 131.0 126.0 2252.1825396825398
8 144.0 144.0 2171.847222222222
9 145.0 151.0 1923.9337748344371
10 161.0 155.0 2185.483870967742
11 126.0 129.0 2806.8217054263564
12 88.0 85.0 2272.6
13 85.0 90.0 3386.311111111111
14 130.0 128.0 4029.421875
15 102.0 99.0 4654.323232323232
16 81.0 81.0 2712.3456790123455
17 108.0 108.0 3009.6018518518517
18 90.0 90.0 5972.711111111111
19 117.0 117.0 4409.982905982906
20 144.0 144.0 3047.7291666666665
21 135.0 135.0 3998.8962962962964
22 126.0 126.0 2887.904761904762
```

```
23 108.0 108.0 2484.259259259259
24 67.0 72.0 4206.819444444444
25 86.0 81.0 4326.641975308642
26 72.0 72.0 3657.5
27 126.0 126.0 4158.277777777777
28 135.0 135.0 1927.325925925926
29 144.0 144.0 4001.5555555555557
30 135.0 135.0 3453.0666666666666
31 126.0 126.0 2935.5079365079364
32 135.0 135.0 2807.1925925925925
33 109.0 114.0 2294.1315789473683
34 121.0 120.0 2206.5
35 31.0 27.0 3209.5185185185187
36 0.0 0.0 0
37 144.0 144.0 4830.652777777777
38 153.0 153.0 2387.3071895424837
39 117.0 117.0 3694.2820512820513
40 109.0 112.0 3912.6071428571427
41 73.0 70.0 2690.3714285714286
42 84.0 88.0 3722.2954545454545
43 112.0 108.0 4454.185185185185
44 117.0 117.0 4550.555555555556
45 124.0 126.0 2381.8253968253966
46 101.0 99.0 5226.69696969697
47 117.0 117.0 2720.1794871794873
48 99.0 99.0 6520.050505050505
49 72.0 72.0 3909.2083333333335
50 72.0 72.0 2910.4583333333335
51 108.0 108.0 2965.685185185185
52 78.0 81.0 4667.876543209876
53 102.0 99.0 3520.3535353535353
54 81.0 81.0 3201.320987654321
55 90.0 90.0 3147.411111111111
56 134.0 135.0 3618.4074074074074
57 100.0 99.0 3975.848484848485
58 99.0 99.0 2932.5656565656564
59 99.0 99.0 4631.69696969697
60 81.0 81.0 4430.259259259259
61 63.0 63.0 4400.555555555556
```

```
62 9.0 9.0 4004.4444444444443
63 73.0 76.0 6576.25
64 80.0 77.0 5423.454545454545
65 72.0 72.0 4375.5
66 72.0 72.0 3894.375
67 138.0 144.0 8578.791666666666
68 88.0 90.0 4432.877777777778
69 115.0 108.0 2270.953703703704
70 109.0 108.0 3623.6296296296296
71 134.0 135.0 3641.562962962963
72 91.0 90.0 4195.811111111111
73 81.0 81.0 3369.4444444444443
74 81.0 81.0 3950.037037037037
75 33.0 33.0 3558.6060606060605
76 75.0 75.0 4613.506666666667
77 72.0 72.0 5818.388888888889
78 72.0 72.0 5504.125
79 108.0 108.0 4635.018518518518
80 63.0 63.0 2652.1428571428573
81 81.0 81.0 3382.962962962963
82 99.0 99.0 2623.060606060606
83 72.0 72.0 4120.638888888889
84 145.0 150.0 4085.04
85 125.0 120.0 3297.266666666667
86 132.0 135.0 4106.903703703703
87 120.0 117.0 3701.803418803419
88 142.0 144.0 4858.902777777777
89 146.0 144.0 3040.8819444444443
90 135.0 135.0 2683.340740740741
91 135.0 135.0 2430.4148148148147
92 90.0 90.0 3235.711111111111
93 108.0 108.0 2770.972222222222
94 81.0 81.0 3640.962962962963
95 54.0 54.0 3820.3888888888887
96 99.0 99.0 3694.2626262626263
97 78.0 81.0 4221.234567901234
98 66.0 63.0 5235.793650793651
99 54.0 54.0 4297.314814814815
100 72.0 72.0 6544.361111111111
```

```
101 45.0 45.0 4944.2444444444445
102 54.0 54.0 5218.851851851852
103 90.0 90.0 4496.766666666666
104 72.0 72.0 4764.513888888889
105 67.0 72.0 4717.208333333333
106 95.0 90.0 6452.2555555555555
107 63.0 63.0 3436.285714285714
108 66.0 72.0 3903.9861111111113
109 132.0 126.0 4753.214285714285
110 72.0 72.0 6669.458333333333
111 141.0 144.0 4318.868055555556
112 138.0 135.0 3850.607407407407
113 45.0 45.0 2646.6666666666665
114 144.0 144.0 4453.888888888889
115 144.0 144.0 3692.7986111111113
116 126.0 126.0 2731.9126984126983
117 82.0 133.0 3037.684210526316
118 29.0 137.0 2512.605839416058
119 0.0 117.0 3210.982905982906
120 0.0 99.0 2833.5555555555557
```

Arquivo 3 - tamanho 1,4 kilobytes:

```
0 415.0 40.0 280.45
1 117.0 119.0 795.5546218487395
2 93.0 91.0 1247.868131868132
3 98.0 98.0 1396.765306122449
4 125.0 126.0 2205.190476190476
5 106.0 105.0 2588.4857142857145
6 126.0 126.0 2197.9603174603176
7 98.0 98.0 2493.887755102041
8 101.0 105.0 6319.704761904762
9 123.0 119.0 3728.3361344537816
10 112.0 112.0 4320.241071428572
11 98.0 98.0 3987.9591836734694
12 98.0 98.0 4088.908163265306
13 70.0 70.0 3790.942857142857
14 63.0 63.0 3747.7301587301586
15 28.0 28.0 2589.9285714285716
16 56.0 56.0 3886.7678571428573
```

```
17 70.0 70.0 5418.4
18 63.0 63.0 5078.984126984127
19 49.0 49.0 7239.306122448979
20 63.0 63.0 4939.603174603175
21 84.0 84.0 5662.238095238095
22 112.0 112.0 6655.696428571428
23 105.0 105.0 7025.857142857143
24 84.0 84.0 4417.333333333333
25 99.0 103.0 3508.1941747572814
26 97.0 93.0 3702.6666666666665
27 91.0 91.0 3989.2747252747254
28 56.0 56.0 4339.589285714285
29 70.0 70.0 5392.314285714286
30 57.0 60.0 4754.833333333333
31 69.0 66.0 4326.621212121212
32 63.0 63.0 4560.222222222223
33 56.0 56.0 4805.035714285715
34 61.0 63.0 3549.936507936508
35 65.0 63.0 3699.2380952380954
36 56.0 56.0 2469.839285714286
37 70.0 70.0 2921.5714285714284
38 42.0 42.0 5059.261904761905
39 66.0 70.0 6540.671428571429
40 88.0 84.0 7728.464285714285
41 98.0 98.0 6397.2959183673465
42 98.0 100.0 5941.31
43 98.0 96.0 4142.864583333333
44 91.0 91.0 6269.054945054945
45 112.0 112.0 2809.75
46 91.0 91.0 3638.3076923076924
47 77.0 79.0 3078.9240506329115
48 102.0 103.0 4102.038834951456
49 99.0 98.0 6319.673469387755
50 104.0 105.0 5935.647619047619
51 94.0 91.0 4567.010989010989
52 91.0 91.0 3706.153846153846
53 56.0 56.0 4202.642857142857
54 67.0 70.0 4847.1
55 59.0 56.0 5496.0
```

```
56 70.0 70.0 4366.4
57 49.0 49.0 4324.918367346939
58 42.0 42.0 3574.6190476190477
59 80.0 84.0 4046.4880952380954
60 60.0 56.0 4711.625
61 70.0 70.0 5486.4857142857145
62 63.0 63.0 6187.777777777777
63 110.0 112.0 7925.446428571428
64 93.0 91.0 4299.7692307692305
65 98.0 98.0 6132.938775510204
66 84.0 84.0 5062.785714285715
67 70.0 70.0 4215.714285714285
68 77.0 77.0 5078.61038961039
69 84.0 84.0 4185.321428571428
70 84.0 85.0 3691.0941176470587
71 84.0 83.0 2979.987951807229
72 84.0 84.0 5498.5952380952385
73 112.0 112.0 4746.892857142857
74 79.0 82.0 3670.6341463414633
75 89.0 86.0 4229.046511627907
76 95.0 98.0 3814.4489795918366
77 101.0 98.0 3173.591836734694
78 91.0 91.0 3636.6043956043954
79 98.0 98.0 5041.816326530612
80 98.0 98.0 3791.969387755102
81 91.0 91.0 3041.6373626373625
82 84.0 84.0 4683.523809523809
83 108.0 114.0 4715.263157894737
84 60.0 54.0 4754.407407407408
85 62.0 63.0 4042.5238095238096
86 63.0 63.0 3745.8253968253966
87 57.0 56.0 4090.125
88 70.0 70.0 3638.842857142857
89 70.0 70.0 2247.2285714285713
90 63.0 63.0 1864.904761904762
91 78.0 84.0 3092.4523809523807
92 69.0 63.0 3476.9841269841268
93 70.0 70.0 2850.5
94 28.0 28.0 2412.3214285714284
```

```
95 56.0 56.0 2611.9285714285716
96 44.0 49.0 5043.3877551020405
97 110.0 105.0 4095.9714285714285
98 84.0 84.0 10210.345238095239
99 105.0 105.0 10694.295238095237
100 88.0 91.0 6766.384615384615
101 98.0 98.0 3654.9591836734694
102 83.0 84.0 9822.130952380952
103 81.0 77.0 8372.35064935065
104 83.0 84.0 4092.4523809523807
105 57.0 56.0 4255.160714285715
106 91.0 91.0 4344.879120879121
107 77.0 77.0 4878.61038961039
108 77.0 77.0 3901.7012987012986
109 49.0 49.0 3828.5510204081634
110 74.0 77.0 5424.077922077922
111 59.0 56.0 3912.125
112 64.0 65.0 6705.384615384615
113 41.0 40.0 4356.175
114 50.0 56.0 5881.607142857143
115 21.0 56.0 5939.946428571428
116 64.0 91.0 7788.395604395604
117 50.0 102.0 6559.588235294118
118 0.0 87.0 3717.402298850575
119 0.0 107.0 5170.878504672897
120 0.0 61.0 4231.098360655737
```

Arquivo 4 - tamanho 3,5 kilobytes:

```
0 381.0 6.0 352.6666666666667
1 52.0 52.0 1234.75
2 54.0 54.0 2145.0925925925926
3 60.0 60.0 3119.15
4 57.0 57.0 4016.0
5 51.0 51.0 4960.431372549019
6 48.0 48.0 5946.0625
7 51.0 51.0 6572.392156862745
8 48.0 48.0 6946.3125
9 45.0 45.0 7112.555555555556
10 42.0 42.0 7048.5
```

```
11 48.0 48.0 6953.354166666667
12 42.0 42.0 6983.428571428572
13 33.0 33.0 7194.727272727273
14 30.0 30.0 8561.566666666668
15 24.0 24.0 9365.166666666666
16 30.0 30.0 10461.666666666666
17 30.0 30.0 9623.466666666667
18 18.0 18.0 11243.611111111111
19 30.0 30.0 10901.133333333333
20 18.0 18.0 10228.5
21 30.0 30.0 10250.066666666668
22 45.0 45.0 12013.044444444444
23 27.0 27.0 11849.851851851852
24 36.0 36.0 11350.916666666666
25 30.0 30.0 10880.266666666666
26 30.0 30.0 9795.1
27 42.0 42.0 7890.452380952381
28 36.0 36.0 11452.833333333334
29 27.0 27.0 9471.666666666666
30 30.0 30.0 9910.4
31 24.0 24.0 10979.416666666666
32 27.0 27.0 15327.888888888889
33 27.0 27.0 12687.407407407407
34 30.0 30.0 12028.166666666666
35 24.0 24.0 10632.166666666666
36 24.0 24.0 10946.708333333334
37 23.0 24.0 12929.166666666666
38 28.0 27.0 15043.481481481482
39 21.0 21.0 10271.380952380952
40 24.0 24.0 17030.458333333332
41 33.0 33.0 17013.090909090908
42 9.0 9.0 26804.555555555555
43 36.0 36.0 20228.055555555555
44 42.0 42.0 16666.333333333332
45 39.0 39.0 12951.71794871795
46 36.0 36.0 9800.527777777777
47 42.0 45.0 5600.288888888889
48 39.0 36.0 13240.194444444445
49 39.0 42.0 12833.785714285714
```

```
50 45.0 42.0 8447.547619047618
51 48.0 48.0 11774.1875
52 30.0 30.0 14517.466666666667
53 42.0 42.0 12158.52380952381
54 45.0 45.0 10606.844444444445
55 39.0 39.0 9947.51282051282
56 30.0 30.0 13395.633333333333
57 33.0 33.0 11195.454545454546
58 30.0 30.0 9378.833333333334
59 24.0 24.0 10186.75
60 35.0 36.0 9887.472222222223
61 31.0 30.0 8870.666666666666
62 21.0 21.0 9027.619047619048
63 33.0 33.0 10653.454545454546
64 27.0 27.0 11240.333333333334
65 30.0 30.0 12220.466666666667
66 21.0 21.0 9893.333333333334
67 24.0 24.0 9899.291666666666
68 24.0 24.0 11681.25
69 21.0 22.0 9799.863636363636
70 24.0 23.0 12533.347826086956
71 21.0 21.0 10911.904761904761
72 36.0 36.0 17515.055555555555
73 42.0 45.0 15596.066666666668
74 35.0 33.0 17255.484848484848
75 37.0 36.0 16700.055555555555
76 45.0 45.0 10986.044444444444
77 42.0 42.0 8443.02380952381
78 36.0 38.0 7942.868421052632
79 39.0 37.0 8881.891891891892
80 42.0 42.0 8536.333333333334
81 36.0 36.0 6795.0
82 48.0 48.0 11279.1875
83 44.0 45.0 10983.888888888889
84 46.0 45.0 9664.133333333333
85 36.0 36.0 12134.972222222223
86 33.0 33.0 11278.030303030304
87 21.0 21.0 15602.47619047619
88 18.0 18.0 13713.666666666666
```

```
89 30.0 30.0 10922.4
90 30.0 30.0 11229.533333333333
91 24.0 24.0 9993.625
92 18.0 18.0 11161.5
93 30.0 30.0 9342.666666666666
94 18.0 18.0 9763.5
95 18.0 18.0 10986.722222222223
96 30.0 30.0 9609.8
97 18.0 18.0 9358.777777777777
98 18.0 18.0 10774.111111111111
99 27.0 27.0 16618.25925925926
100 33.0 33.0 15936.848484848484
101 24.0 24.0 13388.583333333334
102 24.0 24.0 14662.0
103 30.0 30.0 18378.8
104 39.0 39.0 19862.4358974359
105 39.0 42.0 15324.261904761905
106 24.0 42.0 14843.238095238095
107 33.0 39.0 13093.25641025641
108 33.0 42.0 12549.92857142857
109 10.0 36.0 9427.083333333334
110 6.0 27.0 9974.666666666666
111 10.0 30.0 13751.2
112 0.0 30.0 10641.066666666668
113 0.0 27.0 11993.25925925926
114 0.0 30.0 10133.466666666667
115 0.0 30.0 11242.233333333334
116 0.0 27.0 12629.333333333334
117 0.0 42.0 11500.095238095239
118 0.0 39.0 11525.76923076923
119 0.0 39.0 11380.641025641025
120 0.0 8.0 11826.0
```