Eduardo Thomaz dos Santos e Eduardo Niza Minosso

Banco de dados chave-valor distribuído

São Paulo, SP

2024

Eduardo Thomaz dos Santos e Eduardo Niza Minosso

Banco de dados chave-valor distribuído

Trabalho de conclusão de curso apresentado ao Departamento de Engenharia de Computação e Sistemas Digitais da Escola Politécnica da Universidade de São Paulo para obtenção do Título de Engenheiro.

Universidade de São Paulo – USP
Escola Politécnica
Departamento de Engenharia de Computação e Sistemas Digitais (PCS)

Orientador: Prof. Dr. Jorge Rady

São Paulo, SP
2024

Gerar a ficha catalográfica em https://www.poli.usp.br/bibliotecas/servicos/catalogacao-na-publicacao
Salvar o pdf e incluir na monografia

Este trabalho é dedicado às crianças adultas que,
quando pequenas, sonharam em se tornar cientistas.

Agradecimentos

Os agradecimentos principais são direcionados a todos que nos ajudaram direta, ou indiretamente para a conclusão do curso. As nossas famílias, por todo o apoio e ajuda por toda a nossa gradução. Aos nossos amigos que fizemos durante esse curso, que estiveram sempre ao nosso lado e nos incentivaram nos momentos difíceis. E aos professores que nos ensinaram muito, corrigindo e guiando para que nós pudessemos nos formar como Engenheiros de Computação.

Resumo

O relatório descreve o desenvolvimento do Sistema de Banco de Dados Chave-valor Distribuído, que tem como intuito o desenvolvimento de um sistema de banco de dados distribuído baseado no modelo chave-valor e, dessa forma, contribuir para a comunidade científica com meu conhecimento de aplicações de banco de dados e sistema distribuídos.

Palavras-chave: Sistema distribuído. Banco de Dados.

Abstract

The report describes the development of a Distributed Key-Value Database System, which aims to develop a distributed database system based on the key-value model and, in this way, contribute to the scientific community with my knowledge of distributed database and system applications.

Keywords: Distributed System. Database Management System.

Lista de ilustrações

Figura 1 – Requisição considerando um nó único. 13
Figura 2 – Requisição considerando três nós que precisam chegar a um consenso. . 14
Figura 3 – Infraestrutura considerando três *pods* durante comunicações externas. . 16
Figura 4 – Sistema considerando três *pods* durante comunicações internas. 16
Figura 5 – Fluxo das requisições no sistema. 24
Figura 6 – Exemplo de inicialização com três instâncias iniciais e duas dinâmicas. 25
Figura 7 – Arquivo salvo no sistema. 42
Figura 8 – Arquivo salvo e obtido do sistema. 42
Figura 9 – Líder voltou após diversas rodadas de eleição e passou a seguir novo líder. 42

Sumário

1 INTRODUÇÃO 10
1.1 Motivação 10
1.2 Objetivos 11
1.3 Justificativa 11
1.4 Organização do Trabalho 11

2 ASPECTOS CONCEITUAIS 12
2.1 Consenso entre nós 12
2.2 Arquitetura 12
2.3 Observabilidade 14
2.4 Infraestrutura 15
2.5 Banco de Dados Chave-Valor 16
2.5.1 Aplicações que Utilizam Bancos de Dados Chave-Valor 17
2.5.2 Justificativa da Escolha no Projeto 17

3 MÉTODO DO TRABALHO 19

4 ESPECIFICAÇÃO DE REQUISITOS 20

5 DESENVOLVIMENTO DO TRABALHO 21
5.1 Tecnologias Utilizadas 21
5.1.1 Linguagem de Programação Go 21
5.1.2 Protocol Buffers (Protobuf) 21
5.1.3 gRPC 22
5.1.4 Gerenciamento de Configuração com YAML 22
5.1.5 Gerenciamento de Estado e Persistência 22
5.1.6 Controle de Concorrência com `sync` e Goroutines 22
5.1.7 Logger para Monitoramento 23
5.1.8 Outras Ferramentas 23
5.2 Aplicações Desenvolvidas 24
5.2.1 CLI - Aplicações Core 24
5.2.1.1 Orquestrador da Aplicação 26
5.2.1.2 API REST e Balanceador de Carga 26
5.2.1.3 Servidores gRPC 26
5.2.2 Aplicação front-end 26
5.2.2.1 Funcionalidades e Tecnologias Empregadas 27

5.2.2.2 Recuperação de PDFs Armazenados 27
5.2.2.3 Funcionamento Detalhado 28
5.3 Implementação e Algoritmos 29
5.3.1 Implementação do Algoritmo Raft 29
5.3.1.1 Estrutura Geral do Nó Raft 29
5.3.2 Implementação do Servidor e Inicialização do Cluster 30
5.3.2.1 Servidor gRPC Integrado ao Raft 30
5.3.2.2 Inicialização e Configuração do Cluster 31
5.3.2.2.1 Inicialização Estática 31
5.3.2.2.2 Inicialização Dinâmica via Arquivo de Configuração 31
5.3.2.3 Execução e Gerenciamento do Cluster 32
5.3.2.4 Integração com o Algoritmo Raft e o Sistema 32
5.3.2.5 Benefícios para o Projeto 33
5.3.2.6 Estados do Nó 34
5.3.2.7 Persistência de Estado 34
5.3.2.8 Mecanismo de Eleições 34
5.3.2.9 Replicação de Logs e *Heartbeats* 35
5.3.2.10 Aplicação de Entradas de Log 36
5.3.2.11 Benefícios da Implementação 37
5.3.3 Definição das Interfaces com Protocol Buffers 37
5.3.3.0.1 Arquivo `raft.proto` 37
5.3.3.1 Geração de Código com *protoc* 38
5.3.3.2 Integração com o Sistema 39
5.3.3.2.1 Implementação do Servidor 39
5.3.3.2.2 Implementação do Cliente 39
5.3.3.3 Benefícios da Utilização do Protobuf 39
5.3.4 Módulo *grpcutil* 40
5.3.4.1 Cliente gRPC 40
5.3.4.2 Configuração do Cliente 40
5.3.4.3 Cliente Mock 40
5.3.5 Implementação do Módulo `grpcutil` e Considerações Finais 41
5.4 Testes e Avaliação 41

6 CONSIDERAÇÕES FINAIS 44
6.1 Conclusões do Projeto de Formatura 44
6.2 Contribuições 44
6.3 Perspectivas de Continuidade 44

REFERÊNCIAS 45

1 Introdução

1.1 Motivação

Nas últimas décadas a internet surgiu e seu uso cresceu muito. Com isso, a necessidade de lidar com grandes volumes de dados aumentou a demanda por recursos computacionais exponencialmente e sistemas tradicionais, centralizados e baseados em uma única máquina não são suficientes para muitas tarefas. A computação em nuvem também surgiu e possibilitou a flexibilização do uso de máquinas conforme a necessidade de carga, reduzindo custos e investimentos em infraestrutura. Diante da necessidade de trabalhar com aplicações que fazem uso intenso de dados e a maior facilidade em ter mais máquinas, a pesquisa sobre sistemas distribuídos têm se tornado cada vez mais importante.

Sistemas distribuídos funcionam através de uma rede de aplicações interconectadas que trabalham em busca de um mesmo objetivo. Nestes sistemas, o trabalho é dividido entre as diversas máquinas que o compõem, resultando em melhor uso dos recursos computacionais e performance. Além da performance, também há a vantagem em casos de falha, pois quando um computador falha, a carga pode ser transferida para outro sistema. Diante disso, surge o conceito de escalabilidade horizontal, onde ao invés de aumentar recursos da máquina em que a aplicação roda, como CPU e memória, busca-se aumentar o número de réplicas da aplicação.

Sistemas de gerenciamento de bancos de dados representam um tipo de aplicação extremamente beneficiado por sistemas distribuídos e as grandes aplicações deste tipo, como Postgres, MySQL, MongoDB, Redis, etc. possuem suporte para escalabilidade horizontal, seja através do aumento do número de réplicas, permitindo mais acessos simultâneos, ou então através do particionamento dos dados (divisão do dado entre diferentes máquinas), que permite separar uma tarefa em partes menores dividí-las entre diferentes máquinas.

Além dos ganhos de performance, outro benefício é a tolerância a erros. Se tratando de sistemas de armazenamento, que muitas vezes usam discos rígidos mecânicos devido ao baixo custo, as falhas podem ser catastróficas e resultar em uma perda de dados importantes. No entanto, através da replicação de dados, caso ocorra uma falha em um dispositivo, é possível usar outros dispositivos como backup e recuperar os dados sem comprometer o funcionamento do sistema.

Um dos primeiros exemplos reais onde a computação distribuída teve um grande impacto comercialmente foi na criação do Google File System (GFS) [1]. No GFS, além de ganhos significativos em performance, o seu sistema de replicação também possibilitou se recuperar de falhas em sistemas de armazenamento, através da replicação dos dados entre

diferentes máquinas. Assim, uma falha em um HD não impactava na perda dos dados.

1.2 Objetivos

O objetivo deste projeto é desenvolver um sistema sistema de banco de dados distribuído baseado no modelo chave-valor, que seja capaz de lidar com um grande volume de requisições, se recuperar de falhas nos computadores e que possa ser escalado horizontalmente conforme a necessidade. O

Para isto, o sistema será composto por múltiplas replicações, onde cada uma armazenará cópias dos dados e será responsável pela replicação em outros nós. Para garantir a consistência dos dados entre esses múltiplos nós, será utilizado um algoritmo de consenso que permitirá a resolução de conflitos através da eleição de nós como líderes.

1.3 Justificativa

Aplicações que utilizam sistemas distribuídos têm cada vez mais sido utilizados em diversas áreas da sociedade, como redes sociais, sistemas bancários, sistemas de busca, etc. Esses sistemas funcionam através de um conjunto de computadores conectados entre si para realizar uma tarefa. Uma tarefa comum em sistemas distribuídos é o armazenamento e recuperação de dados. Neste contexto, uma aplicação de banco de dados no modelo chave-valor possui um ganho de escalabilidade, performance e tolerância a falha.

Esse trabalho trás uma contribuição significativa para a área de sistemas distribuídos e de sistemas de bancos de dados, pois apresenta a construção de um sistema de banco de dados com escalabilidade horizontal e tolerante a particionamentos da rede mantendo a sua consistência. Portanto, ele é relevante para a sociedade ao aplicar conhecimentos sobre tecnologias inovadoras e de alto impacto na engenharia de software.

1.4 Organização do Trabalho

Este documento apresentará os aspectos conceituais releventes para o desenvolvimento de sistemas de banco de dados distribuídos no Capítulo 2. Em seguida, no Capítulo 3, apresentará a metodologia utilizada para o desenvolvimento do trabalho. No Capítulo 4 apresentará a especificação de requisitos do projeto.

2 Aspectos Conceituais

2.1 Consenso entre nós

Em 1999, Eric Brewer e Armando Fox publicaram um artigo (BREWER; FOX, 1999) onde introduziam o teorema CAP que define que é um possível que um sistema possua três caracterísiticas simultaneas: consistência forte, alta disponibilidade e tolerância a particionamentos.

Este sistema de banco de dados considera desejável se aproximar de características ACID (Atomicidade, Consistência, Isolamento e Durabilidade). Por isso a consistência dos dados é prioritária. Para fins de escalabilidade e performance neste projeto, a consistência forte e tolerância a particionamentos será priorizada em detrimento de alta disponibilidade. Em casos de particionamento na rede, alguns nós podem se tornar temporariamente indisponíveis.

Diversos algoritmos proporcionam a possibilidade de ter forte consistência com tolerância a particionamento. Os principais exemplos são Paxos e Raft (ONGARO; OUSTERHOUT, 2014) e Zab (utilizado na ferramenta Apache Zookeeper) (HUNT et al., 2010).

2.2 Arquitetura

Internamente, a arquitetura do sistema deve buscar reduzir a duplicação e acoplamento do código, para que o processo de desenvolvimento e manutenção do sistema seja facilitado. Por isso, buscarei a separação de diferentes responsabilidades em camadas bem definidas e hierarquizadas, sendo elas:

1. **Camada de API**: será responsável por promover uma interface única para que usuários possam interagir com o sistema, através de operações como leitura e escrita de dados. Esta interface deverá ser capaz de receber conexões através da internet, ter confiabilidade das entregas, capacidade para lidar com diversos formatos de dados (inclusive binários), ser capaz de lidar com grandes volumes de dados e ter uma boa usabilidade para o usuário final.

2. **Camada de Consenso**: terá como responsabilidade a comunicação entre os diferentes nós da aplicação em busca do consenso para garantir a consistência dos dados. Esta camada será diretamente controlada pela camada de aplicação e deve decidir o que fazer conforme a necessidade de uma requisição. Para a comunicação

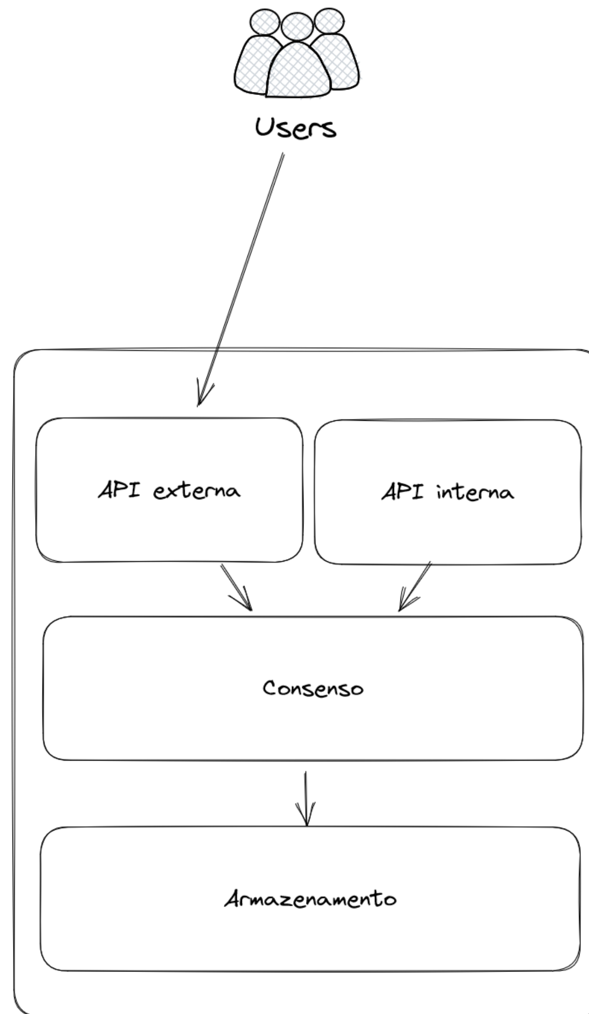

Figura 1 – Requisição considerando um nó único.

entre nós, além de garantir os requisitos anteriores, também é importante levar em consideração a necessidade de baixa latência.

3. **Camada de Armazenamento**: Esta camada será orquestrada pela camada de consenso e deverá ser responsável por realizar as operações de leitura e escrita no disco, garantindo cofiabilidade e performance da escrita.

Na figura, é possível visualizar como o fluxo de uma requisição irá ocorrer pelo sistema, considerando a operação de apenas um único nó.

Entretanto, conforme o sistema é escalado horizontalmente, a camada de consenso pode precisar se comunicar com outras instâncias da aplicação. Na figura 2, é possível observar que a camada de consenso se comunica com uma API interna própria que permitirá outras ações sob o sistema. Assim é possível separar operações que podem ser executadas por usuários comuns de operações internas.

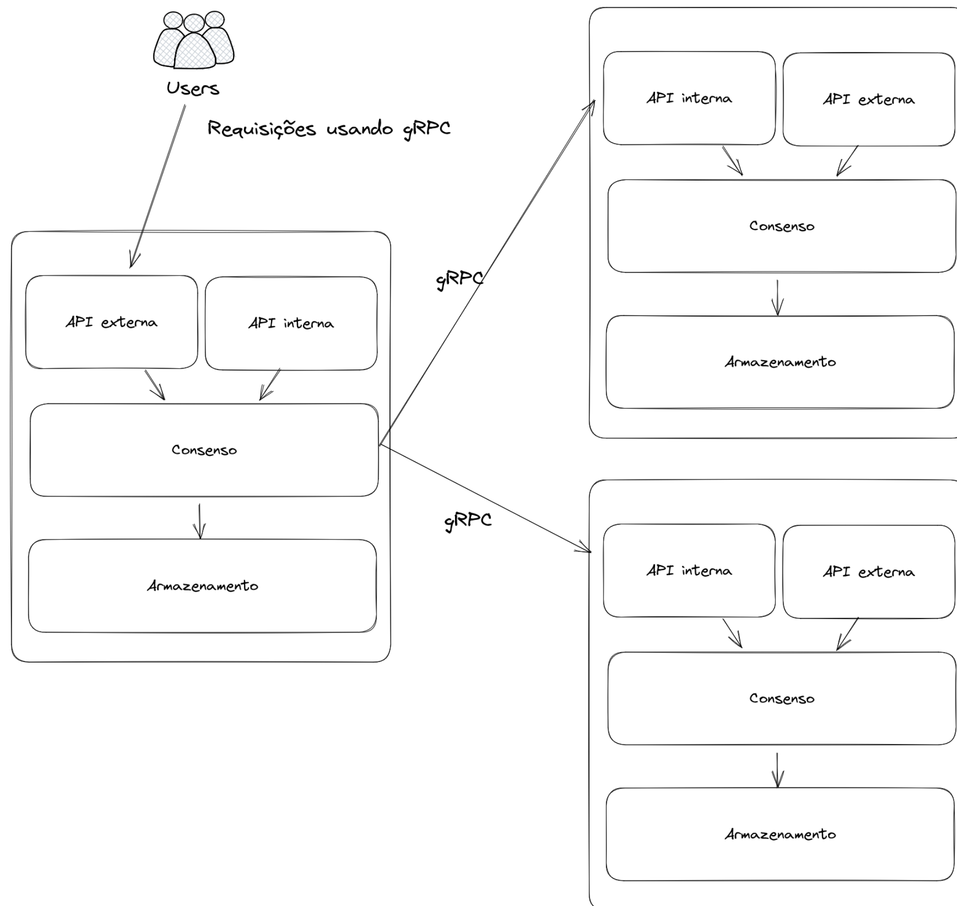

Figura 2 – Requisição considerando três nós que precisam chegar a um consenso.

2.3 Observabilidade

Conforme sistemas distribuídos escalam e se tornam mais complexos, entendê-los e mantê-los se torna mais difícil para as equipes de desenvolvimento e infraestrutura. Para solucionar este problema, uma solução adotada é a observabilidade para garantir a habilidade para monitorar os sistemas em tempo real.

A observabilidade é um conjunto de práticas que permitem entender o estado interno de um sistema através da coleta e análise de saídas. A coleta de dados pode ser dividida em três pilares: logs, métricas e traces.

- **Logs**: arquivos de texto que registram eventos e atividades de um sistema, permitindo a analise detalhada do que aconteceu com o sistema. Um log pode ser usado para informar que um erro aconteceu e, com essa informação, descobrir quando o erro ocorreu, o que estava sendo executado e quais as suas consequências.

- **Traces**: dados sobre como uma requisição atravessa o sistema e seus nós. Estes dados contém informações sobre a latência do sistema e quais partes estão sendo

mais ofensoras no tempo de resposta. Através do uso de tracing, é possível encontrar problemas na execução e gargalos para a performance da aplicação.

- **Métricas**: dados na forma de contagens e taxas sobre os componentes do sistema, como uso de CPU, memória e rede.

Através das coletas e analises destes dados, a analise de problemas se tornará mais fácil, diminuindo o tempo para correção de erros e permitindo melhorias de performance com maior facilidade. Além disso, também facilitará o processo de testagem manual da aplicação em situações de alta carga.

2.4 Infraestrutura

Nos últimos 20 anos, com o surgimento da internet, diversos soluções surgiram para facilitar a criação de aplicações baseadas em web. Uma das mais importantes advento da computação em nuvem, que eliminou a necessidade de altos investimentos e manutenção em hardware.

Através da computação em nuvem, é possível realizar a escalabilidade horizontal das aplicações, onde em um momento de alta carga, mais instâncias do sistema são criadas e depois destruídos conforme se tornam desnecessários (*autoscaling*). Ao lidar com grandes quantidades de nós em sistemas distribuídos, diversos problemas surgem, como escalar conforme a necessidade, lidar com redes, segurança, etc. Diante dessa necessidade, diversas ferramentas e técnicas surgiram, como a virtualização.

A virtualização permite que a aplicação não se preocupe com características do ambiente, como diferentes máquinas e sistemas operacionais. O Docker permite virtualizar o sistema operacional através de containers contendo sua aplicação e como vantagem, é possível garantir que o sistema se comportará da mesma forma em diferentes ambientes, como desenvolvimento e produção. Para facilitar o gerenciamento de containers em larga escala, outra ferramenta criada foi o Kubernetes, que permite diversas abstrações de rede e infraestrutura para aplicações baseadas em container. Através dele é possível operar um sistema distribuído através de diversas máquinas.

O kubernetes facilita a conexão entre diferentes componentes, entre eles o *ingress*, uma aplicação de proxy reverso que é usada para o balanceamento de carga, como o NGINX. Também existe o *service*, que permitirá o roteamento entre diferentes *pods* de uma mesma aplicação (dentro do contexto do kubernetes, é chamada de *deployment*). A figura 3 mostra como a infraestrutura lida com uma requisição externa.

Como uma das características mais importantes deste sistema é permitir a adição e remoção de nós conforme a necessidade, a aplicação precisa descobrir estes. Para isso, o

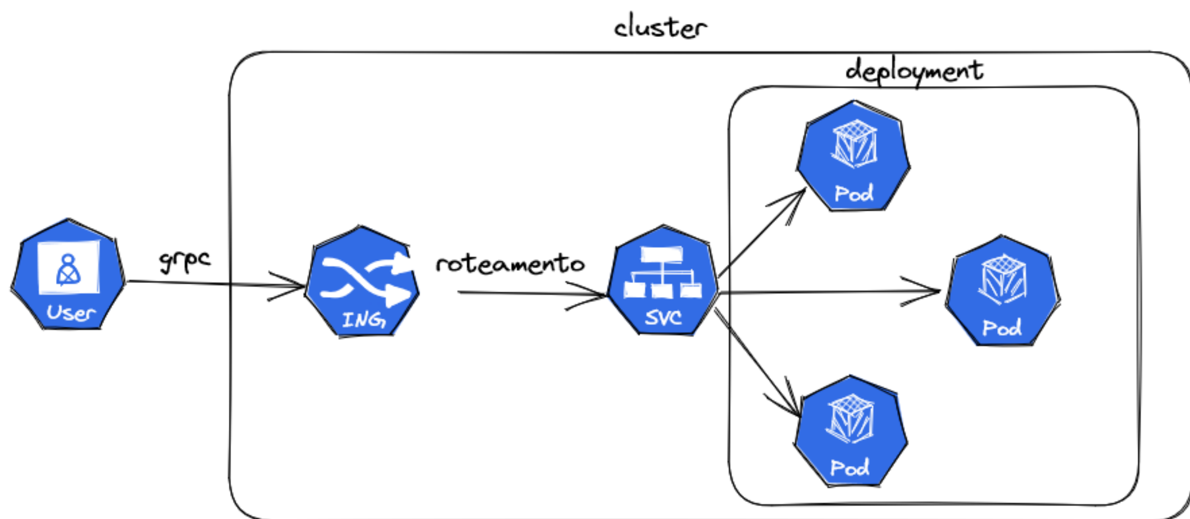

Figura 3 – Infraestrutura considerando três *pods* durante comunicações externas.

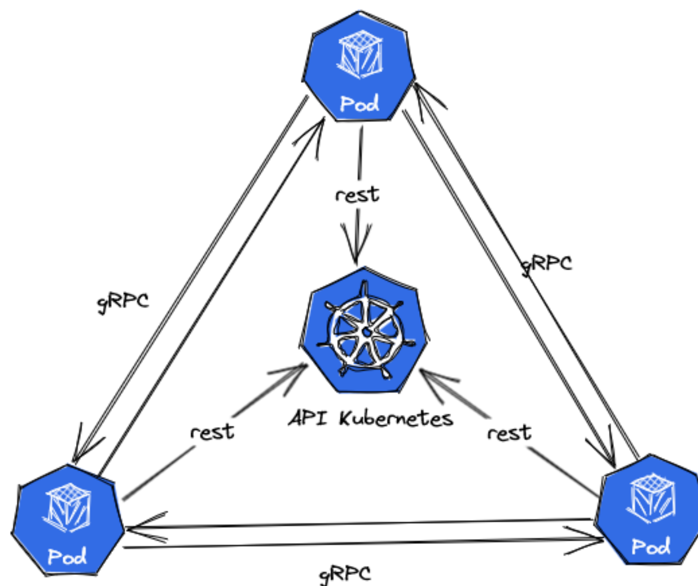

Figura 4 – Sistema considerando três *pods* durante comunicações internas.

kubernetes fornece uma API que permite listar as diversas instâncias da aplicação que estão em execução (chamadas de *pods*). A figura 4 mostra como a aplicação se porta para conseguir se comunicar internamente.

2.5 Banco de Dados Chave-Valor

Um banco de dados chave-valor é um tipo de banco de dados não relacional que utiliza um modelo simples de pares *chave-valor* para armazenar dados. Neste modelo, cada dado armazenado é associado a uma chave única, permitindo acesso rápido e eficiente aos valores correspondentes. Esse tipo de banco de dados é caracterizado por sua simplicidade,

desempenho e capacidade de escalabilidade horizontal, tornando-o ideal para aplicações que exigem alto desempenho e gerenciamento de grandes volumes de dados (Amazon Web Services, 2024; NAPOLEON, 2024; **??**).

Os bancos de dados chave-valor são amplamente utilizados em diversos cenários, especialmente em aplicações que requerem baixíssima latência e alta disponibilidade. Alguns exemplos de bancos de dados chave-valor populares incluem o Redis, Amazon DynamoDB e Riak (IMASTERS, 2024).

2.5.1 Aplicações que Utilizam Bancos de Dados Chave-Valor

Redes Sociais: Plataformas como o Twitter utilizam bancos de dados chave-valor, como Redis, para armazenar sessões de usuários, contagens de seguidores e outras informações que requerem acesso rápido (CARLSON, 2013).

E-commerce: Sites de comércio eletrônico usam esse modelo para gerenciar carrinhos de compras, preferências de usuários e histórico de navegação, proporcionando respostas rápidas e melhor experiência ao cliente.

Sistemas de Cache: O Redis usa chave-valor e é frequentemente usado como cache em aplicações web para acelerar o acesso a dados frequentemente consultados (IMASTERS, 2024).

2.5.2 Justificativa da Escolha no Projeto

A escolha de um modelo de banco de dados chave-valor para este projeto se baseia em diversos fatores alinhados aos objetivos estabelecidos:

- **Simplicidade e Desempenho**: A estrutura simples dos bancos de dados chave-valor permite operações de leitura e escrita extremamente rápidas, atendendo à necessidade de alta performance do sistema.
- **Escalabilidade Horizontal**: O modelo chave-valor se adapta facilmente a arquiteturas distribuídas, facilitando a adição de novos nós ao sistema sem comprometer a consistência dos dados.
- **Flexibilidade**: A capacidade de armazenar dados binários associados a chaves textuais atende diretamente aos requisitos funcionais do projeto, permitindo o armazenamento de diversos tipos de dados.
- **Tolerância a Falhas**: A simplicidade do modelo facilita a implementação de mecanismos de replicação e redundância, essenciais para garantir a disponibilidade e confiabilidade em um ambiente distribuído.

Ao adotar um banco de dados chave-valor, o projeto beneficia-se de uma arquitetura que favorece a escalabilidade e a performance, essenciais para aplicações modernas que lidam com grandes volumes de dados e requisições simultâneas. Além disso, a familiaridade e maturidade das tecnologias associadas a esse modelo facilitam o desenvolvimento e a manutenção do sistema.

3 Método do trabalho

O desenvolvimento deste trabalho está divido em quatro fases: especificação dos requisitos, planejamento do projeto, implantação e testes.

Na especificação dos requisitos, o objetivo é detalhar os objetivos do projeto, funcionalidades que deve atender e características não funcionais que deve possuir. O planejamento do projeto irá ser importante para decidir as tecnologias utilizadas para alcançar os objetivos e idealizar como a aplicação deverá ser testada e demonstrada. Na fase de implantação o sistema será construído e testado aos poucos. Por fim, na fase de testes serão realizados testes sobre o sistema todo garantindo seu funcionamento e características desejadas.

4 Especificação de Requisitos

Para este sistema de banco de dados baseado no modelo chave-valor os requisitos funcionais incluem a capacidade de armazenar dados binários com base em uma chave textual. Ele também deve ser capaz de recuperar os dados, alterá-los ou deletá-los.

Outro requisito funcional do projeto é que o sistema deve ser capaz de suportar um aumento no número de usuários e dados armazenados sem comprometer o desempenho. Para que isto seja possível, ele deve ser escalável horizontalmente, com adição de novas máquinas para se adequar a um aumento de carga.

Além de suas funcionalidades, o projeto deve atender a uma necessidade de alta consistência e tolerância a falhas, então precisa garantir que as informações presentes em vários nós estejam sempre sincronizadas e confiáveis. Também deve possuir como características o desempenho. Para isso, um algoritmo de consenso será utilizado, permitindo consistência dos dados mesmo em casos de particionamento da rede.

Por se tratar de um sistema distribuído que pode ser escalado para usar diversos containers simultaneamente, é difícil garantir seu funcionamento de forma confiável. Para isso, a observabilidade é crucial para o seu monitoramente e detecção de anomalias e ajuda a compreendê-lo em diferentes condições. Assim, é mais fácil detectar e corrigir problemas antes que usuários finais sejam impactados.

Em resumo, o sistema possui os seguintes requisitos funcionais:

- Salvar dados binários com base em uma chave
- Ler dados com base em uma chave
- Deletar dados com base em uma chave
- Atualizar dados com base em uma chave
- Escalabilidade Horizontal

E os seguintes requisitos não funcionais:

- Consistência dos dados
- Alta performance
- Tolerância a falhas
- Observabilidade (coleta e visualização de logs, traces e métricas)

5 Desenvolvimento do Trabalho

5.1 Tecnologias Utilizadas

O desenvolvimento deste trabalho envolveu a utilização de uma série de tecnologias e ferramentas escolhidas pela sua adequação aos requisitos do sistema, com foco em consistência forte, escalabilidade horizontal e tolerância a falhas. A seguir, apresentamos as principais tecnologias utilizadas e as razões que motivaram sua escolha.

5.1.1 Linguagem de Programação Go

A linguagem Go (*Golang*) foi escolhida como base para o desenvolvimento do sistema devido às seguintes características:

- **Alto Desempenho:** Go é uma linguagem compilada que gera código de máquina otimizado, garantindo baixa latência e alto desempenho, aspectos cruciais para sistemas distribuídos.
- **Paralelismo Simplificado:** A implementação de goroutines e canais permite a criação de aplicações altamente concorrentes de forma simples, ideal para gerenciar múltiplos nós de um cluster.
- **Bibliotecas Robustas:** Go oferece suporte nativo para criação de servidores HTTP e gRPC, além de pacotes eficientes para manipulação de arquivos, sincronização de threads e comunicação em rede.
- **Facilidade de Manutenção:** A sintaxe simples e legível de Go, aliada ao gerenciamento de dependências com `go modules`, reduz a complexidade no desenvolvimento e manutenção do código.

5.1.2 Protocol Buffers (Protobuf)

Protocol Buffers, ou Protobuf, é uma tecnologia de serialização de dados utilizada para definir os serviços e mensagens do sistema. Os motivos para sua adoção incluem:

- **Eficiência:** As mensagens geradas são compactas, reduzindo o consumo de banda e melhorando a performance em redes distribuídas.
- **Interoperabilidade:** O Protobuf permite que os serviços sejam definidos de forma neutra em relação à linguagem de programação, facilitando futuras integrações.

- **Automação:** A geração automática de código simplifica a implementação de servidores e clientes gRPC, garantindo consistência entre os nós do sistema.

5.1.3 gRPC

O protocolo gRPC foi utilizado como base para a comunicação entre os nós do cluster. Ele foi escolhido pelas seguintes razões:

- **Comunicação Bidirecional:** Suporte nativo a chamadas *streaming*, permitindo comunicação contínua e eficiente entre líder e seguidores.
- **Interoperabilidade:** Compatível com diversas linguagens de programação, o que pode facilitar a extensão do sistema no futuro.
- **Facilidade de Integração:** Integra-se nativamente com Protobuf, acelerando o desenvolvimento e garantindo consistência na troca de mensagens.

5.1.4 Gerenciamento de Configuração com YAML

O formato YAML foi utilizado para configurar os nós do cluster em implementações dinâmicas. As principais vantagens de sua utilização incluem:

- **Leitura Humana:** A sintaxe simples do YAML facilita a configuração e a compreensão das definições do cluster.
- **Flexibilidade:** Permite a definição de parâmetros, como portas e identificadores dos nós, sem a necessidade de recompilar o código.

5.1.5 Gerenciamento de Estado e Persistência

Para garantir tolerância a falhas, o sistema armazena informações críticas em disco utilizando as bibliotecas nativas de Go para manipulação de arquivos e serialização (`encoding/gob`). Essa abordagem foi escolhida devido à sua simplicidade e eficiência, permitindo que os nós recuperem seu estado rapidamente após falhas ou reinicializações.

5.1.6 Controle de Concorrência com `sync` e Goroutines

A biblioteca padrão de sincronização (`sync`) foi usada para implementar mecanismos de controle de concorrência, como mutexes e `sync.WaitGroup`. Além disso, o uso extensivo de goroutines permitiu que o sistema realizasse operações de forma paralela, aumentando o desempenho e a escalabilidade.

5.1.7 Logger para Monitoramento

Para a produção de logs do sistema, foi desenvolvido um pacote próprio localizado na pasta `pkg/logger`. Este pacote funciona como um *wrapper* para a biblioteca **Zap**, uma ferramenta de logs estruturados desenvolvida pela Uber Technologies, Inc. Uber Technologies, Inc. (2024).

A utilização da biblioteca Zap trouxe diversas vantagens:

- **Desempenho Elevado**: Otimizada para aplicações de alta performance, garantindo eficiência mesmo em sistemas com grande volume de dados.
- **Logs Estruturados**: Permite o registro de informações em formatos como JSON, facilitando a análise e o processamento automatizado.
- **Configuração Flexível**: Suporta diferentes níveis de log (INFO, DEBUG, ERROR) e formatos de saída, adaptando-se a variados cenários operacionais.

O pacote `pkg/logger` padroniza a geração de logs em todo o sistema, evitando repetições de código e garantindo consistência. Isso proporciona:

1. **Consistência**: Mantém o mesmo formato e estrutura de logs em todas as partes do sistema.
2. **Monitoramento Eficiente**: Facilita a identificação de falhas e o rastreamento de problemas em sistemas distribuídos.
3. **Manutenção Simplificada**: Permite ajustes na lógica de logs diretamente no pacote, refletindo-se em toda a aplicação.

Além disso, o logger foi configurado para registrar informações críticas, como mensagens de erro, estados internos dos nós do cluster e dados de depuração, contribuindo significativamente para a observabilidade e a manutenção do sistema distribuído.

5.1.8 Outras Ferramentas

- **Git e GitHub:** Utilizados para versionamento e colaboração no desenvolvimento.
- **Makefiles:** Automatizaram a execução de comandos comuns, como geração de código Protobuf e compilação do projeto.

5.2 Aplicações Desenvolvidas

Por se tratar de um banco de dados, construído para ser utilizado principalmente por outros sistemas de software, o projeto foi desenvolvido com um grande enfoque em recursos técnicos e lógicos, como o algoritmo de replicação e armazenamento e pouco em intercaes gráficas.

É um sistema com poucos recursos para o usuário final, como interface gráfica ou protocolos fáceis de serem utilizados. O protocolo gRPC, por exemplo, apesar de ser bem comum para comunicação entre sistemas distribuídos, não é tão adotado quanto APIs REST e possui uma curva de aprendizado maior.

O sistema foi categorizado em partes que possuem dependência entre si. A primeira delas, nomeada de `Aplicações Core`, contém o que usualmente chama-se de back-end, isto é, que não são comumente utilizadas por um usuário final, mas por outras aplicações.

Além disso, também foi construído um código de *front-end* simples, visando principalmente o auxílio na condução de testes para demonstração do funcionamento correto do sistema como um todo.

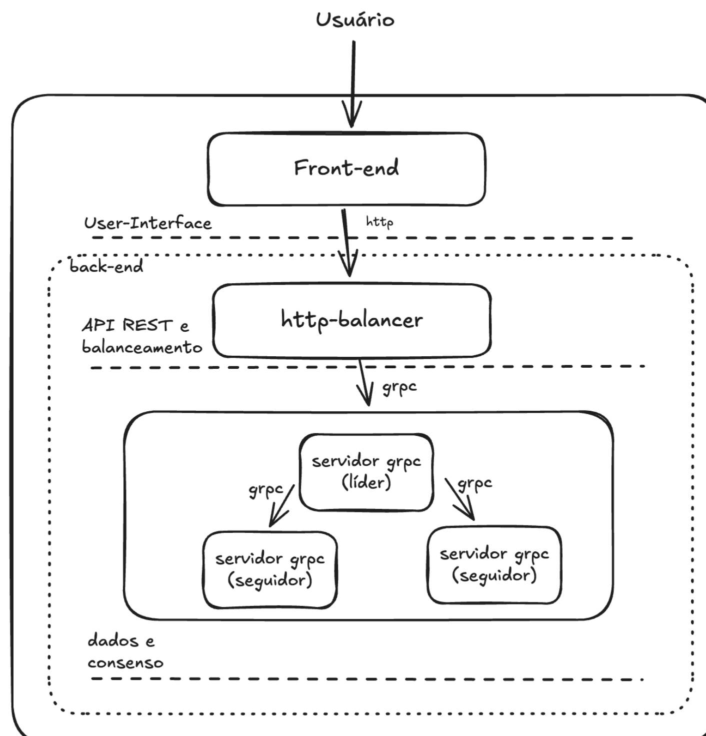

Figura 5 – Fluxo das requisições no sistema.

5.2.1 CLI - Aplicações Core

A primeira parte do sistema é a *CLI (Command-Line Interface)*. Ela é composta por três aplicações contidas em um mesmo arquivo binário e que compartilham a mesma

base de código.

Foi utilizada uma estrutura de comandos que permite facilmente adicionar novas aplicações, se necessário. Ela é compilada através de um arquivo `Makefile` e foi criada de forma a produzir instruções para o usuário, conforme o exemplo a seguir.

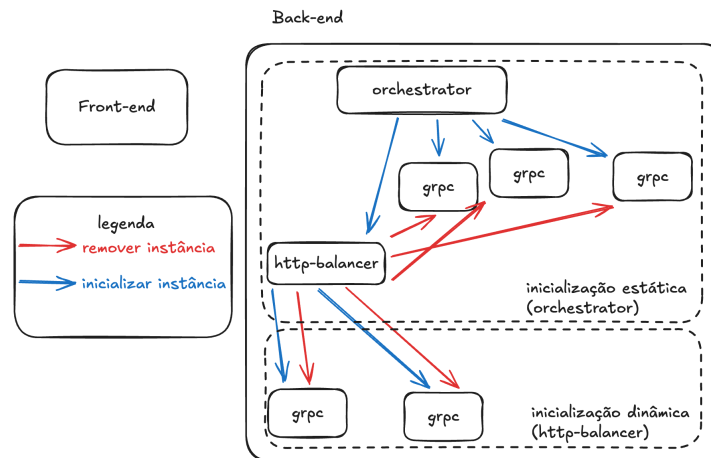

Figura 6 – Exemplo de inicialização com três instâncias iniciais e duas dinâmicas.

```
$./bin/cli help
Single binary with multiple commands

Usage:
  cli [command]

Available Commands:
  completion     Generate the autocompletion script for the specified shell
  grpc           Start grpc server
  help           Help about any command
  http-balancer Start rest http api
  orchestrator   Orchestrates raft nodes

Flags:
  -h, --help   help for cli

Use "cli [command] --help" for more information about a command.
```

Neste resultado do console ao executar o comando de ajuda da CLI, é possível ver a presença de um comando para ajuda, um gerador de scrips de *autocomplete* (para terminal *bash* e *zsh*) e outras três aplicações: (i) grpc (Orquestrador da Aplicação) - que cria servidores gRPC; (ii) `http-balancer` (API REST e Balanceador de Carga) - que cria

um servidor REST HTTP que irá balancear a carga dos sistemas; e (iii) `orchestrator` (Orquestrador da Aplicação) - uma aplicação que é responsável por coordenar as outras.

5.2.1.1 Orquestrador da Aplicação

O orquestrador da aplicação é configurado através de um arquivo `yaml` - tipo de arquivo amplamente usado em configurações de aplicações, como o Docker. Ele é o componente principal das Aplicações Core e, além de carregar as configurações, também inicializa as instâncias das outras aplicações, como os servidores gRPC e o balanceador de carga.

5.2.1.2 API REST e Balanceador de Carga

Esta aplicação possui duas responsabilidades: (i) produzir uma interface mais simples para usuários finais - uma API REST, que é mais fácil de ser utilizada; e (ii) balancear a carga entre os diferentes servidores gRPC.

O balanceador de carga é um intermediador responsável por realizar requisições às instâncias finais do banco de dados, de forma a distribuir a carga igualmente entre as mesmas.

A API REST realiza uma simplificação das interfaces de `protobuf` através do uso de APIs Rest. Ela é organizada em dois grupos de rotas: *db* e *admin*. A primeira é responsável por executar os comandos de salvar, ler e deletar dados do banco de dados. Para isso, ela realiza requisições através da camada de balancemento.

Além de ações sobre o armazenamento do banco de dados, esta também é responsável por adicionar ou remover instâncias (inicialização dinâmica de instâncias) do banco de dados para além da inicialização estática.

5.2.1.3 Servidores gRPC

Por fim, o sistema também possui uma aplicação responsável por realizar as manipulações nos dados (leitura, gravação e deleção de dados).

Estes se comunicam entre-si exclusivamente através do gRPC e também possuem uma interface utilizando o mesmo protocolo. Estas aplicações também são responsáveis por garantir o consenso entre si.

5.2.2 Aplicação front-end

Além da implementação do backend distribuído, foi desenvolvido um front-end para demonstrar, de forma prática, o uso do sistema de banco de dados chave-valor. Esse front-end, construído com o *framework* Angular (TypeScript), ilustra como um usuário

pode interagir facilmente com o sistema, inserindo, consultando e recuperando arquivos PDF a partir da base de dados distribuída.

5.2.2.1 Funcionalidades e Tecnologias Empregadas

- **Interface de Upload de Arquivo:** Uma página em Angular exibe um campo para seleção de um arquivo PDF e outro para a inserção de um nome (chave) associado ao documento. Após o usuário escolher o arquivo, o front-end lê o conteúdo binário e o converte em Base64, garantindo uma transmissão padronizada via HTTP.
- **Conversão para Base64:** O componente TypeScript responsável pelo upload utiliza a `FileReader API` do navegador para ler o PDF e convertê-lo em uma string Base64. Essa conversão torna o PDF adequado para ser armazenado no banco de dados chave-valor, que exige dados textuais para chave e valor.
- **Integração com a API REST do Banco de Dados:** Após a conversão, o usuário insere o nome desejado para o arquivo. Ao clicar em "Registrar", o front-end realiza uma requisição HTTP POST à API do banco de dados distribuído, enviando a chave (nome do arquivo) e o valor (string Base64 do PDF).
- **Armazenamento no Banco de Dados Distribuído:** A API registra a chave e o valor no cluster distribuído, garantindo consistência, disponibilidade e escalabilidade. Assim, o PDF fica armazenado de forma segura e tolerante a falhas, podendo ser recuperado futuramente.

5.2.2.2 Recuperação de PDFs Armazenados

Além do registro de novos PDFs, o front-end implementa um módulo em TypeScript para recuperar arquivos já armazenados. O usuário fornece a chave (nome do PDF), e o componente realiza uma requisição HTTP GET para a API do banco de dados, obtendo a string Base64 correspondente. A seguir, esse valor é decodificado e convertido novamente em PDF, permitindo que o usuário baixe ou visualize o documento.

Esse processo é demonstrado no trecho de código a seguir, que apresenta a função `generatePDF()`:

```
generatePDF(): void {
    if (this.inputBase64) {
        try {
            // Remove o prefixo "data:application/pdf;base64," se existir
            const base64Data = this.inputBase64.startsWith('data:application/pdf;ba
                ? this.inputBase64.split(',')[1]
```

```
                : this.inputBase64;

            // Decodifica a string Base64 em binário
            const byteCharacters = atob(base64Data);
            const byteNumbers = Array.from(byteCharacters).map(char => char.charCod
            const byteArray = new Uint8Array(byteNumbers);
            const blob = new Blob([byteArray], { type: 'application/pdf' });

            // Cria um link para download do PDF
            const url = window.URL.createObjectURL(blob);
            const a = document.createElement('a');
            a.href = url;
            a.download = 'generated.pdf';
            document.body.appendChild(a);
            const event = new MouseEvent('click', { bubbles: true, cancelable: true
            a.dispatchEvent(event);

            document.body.removeChild(a);
            setTimeout(() => {
                window.URL.revokeObjectURL(url);
            }, 1000);
            window.open(url, '_blank');
        } catch (e) {
            this.error = 'Erro ao gerar o PDF. Verifique a chave inserida.';
        }
    } else {
        this.error = 'Por favor, insira uma chave válida.';
    }
}
```

5.2.2.3 Funcionamento Detalhado

- **Requisição HTTP**: O front-end solicita ao banco de dados a string Base64 associada à chave fornecida.
- **Decodificação Base64**: A string é convertida em binário por meio da função `atob()`.
- **Criação do PDF**: Os dados binários são encapsulados em um `Blob` do tipo PDF, permitindo sua manipulação no navegador.

- **Download/Visualização**: É criado dinamicamente um elemento `a` para acionar o download do arquivo, que também pode ser aberto em uma nova aba.
- **Tratamento de Erros**: Em caso de falhas, o usuário é notificado por meio de mensagens de erro.

Portanto, embora este front-end em Angular (TypeScript) seja apenas um exemplo de utilização, ele demonstra a facilidade e a flexibilidade de integrar sistemas distribuídos de armazenamento chave-valor a aplicações web modernas, evidenciando o potencial da solução proposta neste trabalho.

5.3 Implementação e Algoritmos

Este trabalho propõe o desenvolvimento de um sistema distribuído para um banco de dados chave-valor, com foco em consistência forte, escalabilidade horizontal e tolerância a falhas. Para atingir esses objetivos, foi projetado um conjunto de componentes interconectados, integrando o algoritmo de consenso Raft com serviços de comunicação eficientes e persistência de dados. O Raft foi escolhido como o núcleo do sistema, pois oferece simplicidade e robustez no gerenciamento de replicação de logs e na coordenação entre os nós do cluster distribuído.

Além do algoritmo Raft, o sistema utiliza gRPC para comunicação entre os nós e Protocol Buffers (Protobuf) para definição e serialização das mensagens. A arquitetura modular e flexível do sistema permite a inicialização dinâmica do cluster, garantindo adaptabilidade em diferentes cenários. Nesta seção, apresentamos a arquitetura do sistema, detalhamos o funcionamento do Raft, sua integração com os serviços de comunicação e as decisões técnicas que sustentam o desenvolvimento do projeto.

5.3.1 Implementação do Algoritmo Raft

Para garantir a consistência e a tolerância a falhas no banco de dados distribuído, foi implementado o algoritmo de consenso *Raft*. O *Raft* é amplamente utilizado em sistemas distribuídos por sua simplicidade e clareza em comparação com outros algoritmos de consenso, como o Paxos. A escolha pelo *Raft* se alinha aos objetivos do projeto de fornecer um sistema robusto, escalável e confiável.

5.3.1.1 Estrutura Geral do Nó Raft

Cada nó do cluster não apenas implementa uma instância do algoritmo *Raft*, mas também incorpora um servidor gRPC que expõe os serviços necessários para a comunicação entre os nós e com clientes externos. A estrutura principal do nó é definida pela combinação

da estrutura `Raft` e do `Server`, integrando os componentes de comunicação e lógica de consenso.

A estrutura `Server` é responsável por:

- **Serviços gRPC**: Implementa os métodos definidos nos arquivos `.proto`, como `Heartbeat`, `RequestVote`, `Set`, `Delete` e `Get`.
- **Interação com o Raft**: Cada método gRPC delega as operações à instância `Raft` correspondente, garantindo que a lógica de consenso seja aplicada a todas as operações.
- **Sincronização**: Utiliza mutexes para garantir a segurança em concorrência durante o acesso e modificação do estado interno.

A inicialização do servidor é realizada pela função `Start`, que configura o servidor gRPC, registra os serviços e inicia a escuta na porta especificada. Além disso, o servidor monitora sinais do sistema operacional para realizar um desligamento gracioso, garantindo a persistência do estado e a integridade do sistema.

5.3.2 Implementação do Servidor e Inicialização do Cluster

A implementação do servidor e a inicialização do cluster são componentes essenciais para o funcionamento do sistema distribuído. Eles permitem que múltiplos nós operem em conjunto, comunicando-se via gRPC e coordenando-se através do algoritmo Raft.

5.3.2.1 Servidor gRPC Integrado ao Raft

O servidor é implementado pela estrutura `Server`, que combina a funcionalidade do gRPC com a lógica do Raft:

```
type Server struct {
    mu     sync.Mutex
    raft   *raft.Raft
    logger logger.Logger

    grpc *grpc.Server

    pb.UnimplementedRaftServer
    pb.UnimplementedDatabaseServer
}
```

Os principais pontos de integração incluem:

- **Métodos Raft**: Implementação dos métodos `Heartbeat` e `RequestVote`, que são chamados por outros nós do cluster.
- **Operações do Banco de Dados**: Implementação dos métodos `Set`, `Delete` e `Get`, permitindo interações com o armazenamento chave-valor.
- **Comunicação Segura**: Uso de mutexes para sincronizar o acesso ao estado do servidor, prevenindo condições de corrida.

5.3.2.2 Inicialização e Configuração do Cluster

A inicialização do cluster é realizada em duas abordagens distintas, demonstrando a flexibilidade do sistema:

5.3.2.2.1 Inicialização Estática

No primeiro cenário, os nós do cluster são definidos estaticamente no código:

```
nodes := make(map[raft.ID]*raft.Node)
nodes["A"] = raft.NewNode("[::]:8080")
nodes["B"] = raft.NewNode("[::]:8081")
nodes["C"] = raft.NewNode("[::]:8082")
```

Cada nó é iniciado em uma goroutine separada, permitindo a execução concorrente dos servidores:

```
go func() {
    server := server.CreateServer(
        raft.MakeRaft("A", nodes),
    )
    server.Start(":8080")
}()
```

5.3.2.2.2 Inicialização Dinâmica via Arquivo de Configuração

No segundo cenário, a configuração dos nós é carregada a partir de um arquivo YAML, permitindo maior flexibilidade e escalabilidade:

```
file, err := os.ReadFile("raft.yaml")
var config Config
yaml.Unmarshal(file, &config)
```

Os nós são criados dinamicamente com base nas informações do arquivo:

```
for _, s := range config.RaftCluster.Servers {
    nodes[s.ID] = raft.NewNode(fmt.Sprintf("[::]%s", s.Port))
}
```

5.3.2.3 Execução e Gerenciamento do Cluster

Em ambos os casos, os servidores são iniciados e gerenciados utilizando `sync.WaitGroup` para sincronizar a execução das goroutines:

```
var wg sync.WaitGroup
for _, s := range config.RaftCluster.Servers {
    wg.Add(1)
    go func(srv Server) {
        defer wg.Done()
        server := server.CreateServer(
            raft.MakeRaft(srv.ID, nodes),
        )
        server.Start(srv.Port)
    }(s)
}
wg.Wait()
```

Essa abordagem assegura que todos os nós do cluster sejam iniciados corretamente e permaneçam em execução durante o funcionamento do sistema.

5.3.2.4 Integração com o Algoritmo Raft e o Sistema

A implementação do servidor e a inicialização do cluster são cruciais para a integração completa do algoritmo Raft ao sistema. Elas permitem:

- **Comunicação entre Nós**: Utilizando gRPC, os nós podem trocar mensagens de forma eficiente e padronizada.
- **Execução Concorrente**: A utilização de goroutines e sincronização permite que o sistema aproveite o paralelismo, melhorando a performance.
- **Flexibilidade de Configuração**: A possibilidade de definir os nós via arquivo de configuração facilita a adaptação do sistema a diferentes ambientes e necessidades.

5.3.2.5 Benefícios para o Projeto

A implementação detalhada do servidor e do processo de inicialização do cluster trouxe os seguintes benefícios ao projeto:

- **Escalabilidade Horizontal**: Facilita a adição ou remoção de nós no cluster sem a necessidade de alterações significativas no código.
- **Robustez e Confiabilidade**: O tratamento adequado de sinais do sistema operacional e a sincronização de acesso garantem uma operação estável.
- **Adaptabilidade**: A configuração dinâmica permite que o sistema seja facilmente adaptado a diferentes cenários de uso e requisitos.

Cada nó do cluster implementa uma instância do algoritmo *Raft*, mantendo estados críticos e participando do processo de consenso. A estrutura principal do nó é definida pela estrutura `Raft`, que contém os seguintes componentes:

- **Identificador do Nó (`me`)**: Representa o ID único do nó dentro do cluster.
- **Estados Persistentes**:
  - `currentTerm`: O termo atual do nó, utilizado para identificar a validade de mensagens.
  - `votedFor`: ID do candidato que recebeu o voto do nó no termo atual.
  - `logEntry`: Registro das entradas de log que precisam ser replicadas.
- **Estados Voláteis**:
  - `commitIndex`: Índice da maior entrada de log conhecida como comprometida.
  - `lastApplied`: Índice da última entrada de log aplicada ao estado do nó.
- **Estados Voláteis em Líderes**:
  - `nextIndex`: Para cada nó, o índice da próxima entrada de log que o líder enviará para replicação.
  - `matchIndex`: Para cada nó, o índice da maior entrada de log conhecida replicada naquele nó.

5.3.2.6 Estados do Nó

O algoritmo *Raft* define três estados para os nós: *Follower* (seguidor), *Candidate* (candidato) e *Leader* (líder). A transição entre esses estados é gerenciada pela enumeração `State`, como mostrado abaixo:

```
type State int

const (
    Follower State = iota + 1
    Candidate
    Leader
)
```

5.3.2.7 Persistência de Estado

A persistência de estado é uma característica fundamental para garantir a tolerância a falhas no sistema. Informações críticas, como o termo atual, o nó votado, o índice de commit, e as entradas de log, são armazenadas em disco para permitir que os nós recuperem seu estado após falhas ou reinicializações.

A persistência é gerenciada por funções específicas que serializam o estado do nó em um arquivo local. Esses arquivos são organizados em uma estrutura que identifica cada nó pelo seu ID exclusivo. A serialização é realizada utilizando o pacote `encoding/gob`, que converte as estruturas de dados do Go em um formato adequado para armazenamento.

Por exemplo, a função responsável por salvar o estado coleta todos os dados relevantes do nó em uma estrutura e os serializa para um arquivo específico. Essa abordagem garante que o estado do nó possa ser carregado novamente de maneira confiável, mantendo a integridade do sistema mesmo em cenários de falhas inesperadas.

Essa estratégia contribui significativamente para a robustez do sistema distribuído, reduzindo os impactos de falhas e otimizando a recuperação de nós.

5.3.2.8 Mecanismo de Eleições

Quando um nó não recebe *heartbeats* do líder dentro de um intervalo de tempo aleatório, ele se torna um *Candidate* e inicia uma eleição. O processo de eleição é realizado pelas funções `broadcastRequestVote` e `sendRequestVote`, que solicitam votos dos outros nós:

```
func (r *Raft) broadcastRequestVote() {
    args := dto.VoteArgs{
```

```
        Term:        r.currentTerm,
        CandidateID: r.me,
    }

    for i := range r.nodes {
        go func(i ID) {
            r.sendRequestVote(i, args)
        }(i)
    }
}
```

Se o candidato receber votos da maioria dos nós, ele se torna o líder e começa a enviar *heartbeats* para os seguidores.

5.3.2.9 Replicação de Logs e *Heartbeats*

O líder é responsável por manter os seguidores atualizados, replicando as entradas de log através de *heartbeats*. A função `broadcastHeartbeat` envia periodicamente mensagens aos seguidores contendo novas entradas de log e informações de commit:

```
func (r *Raft) broadcastHeartbeat() {
    for i := range r.nodes {
        var args dto.HeartbeatArgs

        args.Term = r.currentTerm
        args.LeaderID = r.me
        args.LeaderCommit = r.commitIndex

        prevLogIndex := max(r.nextIndex[i]-1, 0)
        if r.getLastIndex() > prevLogIndex {
            args.PrevLogIndex = prevLogIndex
            args.PrevLogTerm = r.logEntry[prevLogIndex].Term
            args.Entries = r.logEntry[prevLogIndex:]
        }

        go func(i ID, args dto.HeartbeatArgs) {
            r.sendHeartbeat(i, args)
        }(i, args)
    }
}
```

Os seguidores, ao receberem um *heartbeat*, atualizam seus logs e confirmam o recebimento, permitindo que o líder avance o `commitIndex` quando a maioria dos nós tiver replicado as entradas.

5.3.2.10 Aplicação de Entradas de Log

As entradas de log no algoritmo *Raft* representam comandos que precisam ser aplicados ao estado do sistema, garantindo a consistência entre os nós do cluster. Esses comandos incluem operações básicas de armazenamento de chave-valor, como `SET`, para adicionar ou atualizar um valor associado a uma chave, e `DELETE`, para remover uma chave específica.

A aplicação desses comandos é gerenciada pela função `apply`, que verifica as entradas do log já comprometidas e as executa sequencialmente. Durante essa execução, o sistema realiza as seguintes etapas:

- **Validação do Índice de Commit:** Antes de processar as entradas, a função garante que o índice de commit (`commitIndex`) é maior ou igual ao índice da última entrada aplicada (`lastApplied`). Essa validação evita inconsistências no estado.
- **Execução de Comandos:** Cada entrada do log é avaliada com base no tipo de operação:
  - Para comandos `SET`, o sistema salva a chave e o valor no armazenamento.
  - Para comandos `DELETE`, a chave correspondente é removida.
- **Atualização do Estado:** Após a execução bem-sucedida de uma entrada, o índice `lastApplied` é atualizado para refletir a última entrada processada.

Essa abordagem assegura que todas as operações comprometidas sejam aplicadas de forma ordenada e consistente, mesmo em cenários de falhas.

Por exemplo, a função avalia o tipo de comando com um `switch` simples, que diferencia as operações `SET` e `DELETE`, garantindo a aplicação correta e apropriada ao armazenamento de dados. Erros encontrados durante o processamento, como falhas no acesso ao armazenamento, são capturados e reportados, reforçando a confiabilidade do sistema.

Essa lógica de aplicação de comandos é essencial para manter a integridade do estado global do sistema e sustenta a confiabilidade do algoritmo *Raft* em garantir consistência forte entre os nós.

5.3.2.11 Benefícios da Implementação

A utilização do algoritmo *Raft* trouxe os seguintes benefícios ao projeto:

- **Consistência Forte**: Garantia de que todos os nós possuem o mesmo estado após a aplicação das entradas de log.
- **Tolerância a Falhas**: Capacidade de continuar operando mesmo com a falha de alguns nós, devido ao mecanismo de eleição e replicação.
- **Escalabilidade Horizontal**: Facilidade em adicionar novos nós ao cluster sem impactar a consistência ou disponibilidade.
- **Alta Disponibilidade**: Possibilidade de atender operações de leitura em qualquer nó e rápida recuperação de falhas.

5.3.3 Definição das Interfaces com Protocol Buffers

Para facilitar a comunicação entre os componentes do sistema distribuído e garantir interoperabilidade, utilizamos o *Protocol Buffers* (protobuf), uma linguagem neutra e independente de plataforma para serializar dados estruturados. O protobuf permite definir interfaces de serviços e estruturas de dados de forma concisa, gerando código automaticamente para várias linguagens de programação.

5.3.3.0.1 Arquivo `raft.proto`

O arquivo `raft.proto` define as interfaces e mensagens necessárias para implementar o serviço `Raft`, que é fundamental para a operação do protocolo de consenso Raft no sistema distribuído. Ele especifica os métodos que permitem a comunicação entre os nós do cluster para manter a consistência e coordenar ações críticas como eleições e replicação de logs.

Os métodos definidos incluem:

- `RequestVote`: Gerencia solicitações de voto durante o processo de eleição de um novo líder. Um nó candidato utiliza este método para solicitar votos de outros nós.
- `Heartbeat`: Realiza a sincronização periódica entre o líder e os seguidores, mantendo-os atualizados sobre o estado atual do sistema e assegurando que os nós seguidores não iniciem eleições desnecessárias.
- `SearchLog`: Permite a busca por entradas específicas no log replicado, facilitando a recuperação e verificação de informações armazenadas.

Cada um desses métodos é acompanhado por mensagens que encapsulam os parâmetros de entrada e as respostas necessárias para o funcionamento correto do protocolo. Por exemplo:

- **`RequestVoteArgs`** e **`RequestVoteReply`**: Estruturas que contêm informações sobre o termo atual, o candidato solicitante e se o voto foi concedido, respectivamente. Isso permite aos nós decidirem se devem votar em um candidato com base nos termos e no histórico de logs.

- **`HeartbeatArgs`** e **`HeartbeatReply`**: Mensagens que incluem informações sobre o termo, o identificador do líder, entradas de log a serem replicadas, e confirmam o sucesso da operação. Isso garante que os seguidores atualizem seus logs e termos conforme necessário.

- **`LogEntry`** e **`LogCMD`**: Estruturas que representam entradas individuais no log, incluindo o termo, índice e o comando associado (como operações de `SET` ou `DELETE`). Essas estruturas são fundamentais para a replicação consistente de logs entre os nós.

- **`SearchLogArgs`** e **`SearchLogReply`**: Mensagens que permitem a busca por entradas no log com base em critérios específicos, retornando o comando encontrado, se disponível, e informações sobre seu estado de commit.

A definição clara dessas mensagens e serviços no arquivo `raft.proto` é essencial para o funcionamento correto do protocolo Raft no sistema. Ela estabelece um contrato de comunicação entre os nós, garantindo que todos utilizem os mesmos formatos de mensagem e compreendam os mesmos parâmetros e respostas.

Além disso, ao utilizar o Protocol Buffers para definir essas interfaces, o sistema se beneficia de uma comunicação eficiente e serialização compacta, o que é crítico para desempenho em sistemas distribuídos. A geração automática de código a partir dos arquivos `.proto` também assegura consistência entre as implementações dos nós e facilita a manutenção e evolução do sistema.

5.3.3.1 Geração de Código com *protoc*

Utilizando o compilador `protoc`, os arquivos `.proto` são processados para gerar código em Go. As seguintes etapas foram realizadas:

1. Definição dos arquivos `database.proto` e `raft.proto`, especificando serviços, métodos e mensagens.

2. Execução do comando `protoc` para gerar os códigos:

```
protoc --go_out=. --go-grpc_out=. proto/database.proto
protoc --go_out=. --go-grpc_out=. proto/raft.proto
```

3. Inclusão dos pacotes gerados no projeto, permitindo que a aplicação utilize as interfaces definidas.

5.3.3.2 Integração com o Sistema

Os códigos gerados automaticamente fornecem as estruturas e interfaces necessárias para implementar os servidores e clientes gRPC. No lado do servidor, as interfaces são implementadas para atender às chamadas dos clientes, enquanto no lado do cliente, as funções geradas permitem realizar chamadas remotas aos métodos definidos.

5.3.3.2.1 Implementação do Servidor

No servidor, as interfaces geradas são implementadas conforme a lógica do sistema. Por exemplo, para o serviço `Database`, os métodos `Set`, `Delete` e `Get` são implementados para interagir com o armazenamento local e coordenar a replicação via Raft.

5.3.3.2.2 Implementação do Cliente

No cliente, as funções geradas são utilizadas para criar conexões e realizar chamadas aos métodos remotos. Isso permite que componentes do sistema ou aplicativos externos interajam com o banco de dados distribuído de forma transparente.

5.3.3.3 Benefícios da Utilização do Protobuf

A adoção do *Protocol Buffers* trouxe diversos benefícios para o projeto:

- **Eficiência**: As mensagens protobuf são compactas, reduzindo o consumo de banda e melhorando a performance.
- **Interoperabilidade**: As definições em `.proto` são independentes de linguagem, permitindo futuras extensões em outras linguagens se necessário.
- **Facilidade de Manutenção**: As mudanças nas interfaces podem ser gerenciadas centralmente nos arquivos `.proto`, e o código gerado reflete automaticamente essas mudanças.
- **Consistência**: Garante que todos os componentes do sistema utilizem as mesmas definições de mensagens e serviços, evitando incompatibilidades.

5.3.4 Módulo *grpcutil*

O pacote `grpcutil` é responsável por abstrair a comunicação entre os nós do sistema utilizando o protocolo gRPC. Esse módulo encapsula tanto a funcionalidade de clientes quanto de mocks, permitindo maior flexibilidade nos testes e na execução da aplicação. Abaixo, explicamos os principais componentes do pacote:

5.3.4.1 Cliente gRPC

A implementação do cliente gRPC, representada pela estrutura `grpcClient`, fornece métodos para interagir com outros nós no cluster, incluindo as chamadas `Heartbeat` e `RequestVote`. Esses métodos são essenciais para o funcionamento do protocolo Raft, pois possibilitam: - **Heartbeat**: A comunicação periódica do nó líder com os seguidores para manter a consistência e verificar se eles estão ativos. - **RequestVote**: O processo de eleição de um novo líder quando o atual está inativo ou ocorre um reinício.

O cliente é configurado com as seguintes características: - Utilização de credenciais inseguras (`insecure.NewCredentials()`), simplificando o desenvolvimento e os testes. - Manuseio automático da conexão com o servidor gRPC via `grpc.DialOption`.

Além disso, a função `makeGrpcClient` facilita a criação de instâncias do cliente com os parâmetros necessários para a comunicação.

5.3.4.2 Configuração do Cliente

O módulo também inclui uma estrutura `config` que permite ativar um cliente mock por meio de uma variável de ambiente (`MOCK_GRPC_CLIENT`). Esse mecanismo possibilita alternar entre o cliente real e um cliente simulado para cenários de testes, sem necessidade de alterar o código principal.

A função `MakeClient` atua como uma fábrica que decide qual implementação (real ou mock) será utilizada com base nessa configuração, garantindo flexibilidade no ambiente de execução.

5.3.4.3 Cliente Mock

O `mockClient` é uma implementação simulada do cliente gRPC, utilizada principalmente para testes. Ele é controlado por um `mockController`, que permite configurar as respostas esperadas para as chamadas `Heartbeat` e `RequestVote`. Isso é especialmente útil para validar a lógica de negócios do algoritmo Raft em situações controladas, como: - Simular falhas na comunicação. - Testar cenários em que o líder ou os seguidores retornam respostas específicas.

O uso de mocks reduz a dependência de um ambiente distribuído completo durante o desenvolvimento e os testes iniciais, acelerando o ciclo de desenvolvimento.

5.3.5 Implementação do Módulo `grpcutil` e Considerações Finais

O módulo `grpcutil` desempenha um papel central na solução do TCC ao fornecer uma camada eficiente e testável para a comunicação entre os nós do sistema distribuído. Ele utiliza o gRPC para implementar as chamadas essenciais do protocolo Raft, como `Heartbeat`, que mantém a consistência entre os nós e verifica sua atividade, e `RequestVote`, responsável pelo gerenciamento de eleições de liderança.

A configuração flexível do cliente, incluindo o suporte a mocks controlados por variáveis de ambiente, permitiu validar a lógica de negócios do sistema sob diferentes cenários, como falhas na comunicação e comportamento de nós específicos. Essa abordagem simplifica o desenvolvimento e os testes, garantindo uma implementação confiável e robusta.

O uso de arquivos `.proto` e a geração automática de código foram fundamentais para abstrair a complexidade da comunicação entre os nós, permitindo que o foco permanecesse na implementação da lógica do protocolo de consenso Raft. A separação clara entre as interfaces e a implementação facilita futuras extensões e manutenção do sistema.

A aplicação do Raft foi essencial para garantir um sistema distribuído eficiente, tolerante a falhas e escalável. Durante os testes, o algoritmo demonstrou sua capacidade de lidar com falhas de nós, inconsistências de logs e alta carga de operações, assegurando confiabilidade mesmo em cenários complexos.

5.4 Testes e Avaliação

Para além da criação do código-fonte, é de extrema importância a execução de testes que de fato comprovem o funcionamento do sistema como um todo. Para isto, houve um enfoque principalmente nos testes das funcionalidades de *back-end*, com o *front-end* sendo usado apenas para auxiliar nos testes e demonstrações.

Os testes de bancos de funcionalidade do banco de dados foram executados e apresentaram os resultados esperados. Na imagem abaixo, podemos ver que foi possível salvar um arquivo pdf de tamanho razoável (1 MB) em pouquíssimos segundos, garantindo sua distribuição pelo sistema.

Também foram realizados com sucesso os testes da eleição em casos simples, como complexos. Ao parar um líder por um longo período de tempo e iniciar seus dados após diversas eleições. Conforme o esperado, ele passou a respeitar o novo líder, estabelecido em um eleição posterior.

Por fim, também foram realizados diversos testes para garantir a consistência dos nós, capacidade de lidar com particionamento e erros de rede. Além de garantir a possibilidade de se recuperar de falhas

Figura 7 – Arquivo salvo no sistema.

Figura 8 – Arquivo salvo e obtido do sistema.

Figura 9 – Líder voltou após diversas rodadas de eleição e passou a seguir novo líder.

Funcionalidades do Banco de Dados

1. Verificar se o `SET` está funcionando
2. Verificar se o `GET` está funcionando

    2.1. Obter um PDF

   2.2. Obter uma imagem

3. Verificar `DELETE`

Raft Eleição

1. Casos de sucesso - líder eleito
2. Derrubar um líder e garantir que um novo é eleito
3. Testar em caso de votos divididos, garantindo que um líder é eleito

Consistência

1. Garantir que todos os nós replicam o nó do líder corretamente
2. Verificar que, ao nó ser `committed` no líder, ele também é `committed` nos seguidores (2-phase commit)
3. Testar nó sem o log e garantir que ele irá recuperar o log do líder

Nestes testes, foi possível validar a consistência do sistema. Ao inserir propositalmente nós desatualizados, os mesmos devem ser capazes de se tornar consistentes com o líder do sistema.

Safety

1. Testar com dois líderes e garantir que o conflito é resolvido
2. Garantir que o candidato deve ter `committed` todos os logs para se tornar líder

Fault Tolerance

1. Simular quedas na rede e garantir que o sistema continua funcionando somente se tivermos a maioria operante
2. Simular falha em nós que não causem a queda do sistema
3. Garantir que nós que falharam conseguem voltar a funcionar

6 Considerações Finais

6.1 Conclusões do Projeto de Formatura

Através da conclusão deste projeto de formatura, foi possível um enorme aprendizado sobre sistemas distribuídos, concorrência, paralelismo em software e sobre o funcionamento de sistemas de armazenamento e de bancos de dados.

6.2 Contribuições

Neste projeto, grande parte da execução foi realizada através da criação de código próprio, com auxílio de pesquisas. A equipe construiu a implementação das APIs REST (utilizando como base o framework `gin`, o orquestrador (utilizando o algoritmo de Round-Robbin, com base na especificação do mesmo), as APIs gRPC (com base na biblioteca de grpc construída pelo Google) e o algoritmo de consenso Raft.

6.3 Perspectivas de Continuidade

Com base neste trabalho, adaptações e melhorias podem ser feitas, como a implementação de seu funcionamento em nuvem, utilizando virtualização e orquestradores de containers (como kubernetes).

As camadas da aplicação podem ser remodeladas, considerando principalmente a separação das entidades de API REST e balanceamento.

Por fim, outras aplicações poderiam ser construídas usando esta com base. Uma aplicação web poderia usá-la como cache e um outro banco de dados SQL poderia ser construído usando este como base.

Referências

Amazon Web Services. *O que é um banco de dados de chave-valor?* 2024. Disponível em: <https://aws.amazon.com/pt/nosql/key-value/>. Citado na página 17.

BREWER, E. A.; FOX, A. Lessons from giant-scale services. *IEEE Internet Computing*, v. 3, n. 4, p. 46–55, 1999. Citado na página 12.

CARLSON, J. L. *Redis in Action*. [S.l.]: Manning Publications, 2013. Citado na página 17.

HUNT, P. et al. Zookeeper: Wait-free coordination for internet-scale systems. In: *Proceedings of the USENIX Annual Technical Conference*. [S.l.: s.n.], 2010. p. 145–158. Citado na página 12.

IMASTERS. *Introdução ao Redis, o NoSQL chave-valor mais famoso*. 2024. Disponível em: <https://imasters.com.br/banco-de-dados/introducao-ao-redis-o-nosql-chave-valor-mais-famoso>. Citado na página 17.

NAPOLEON. *O que é: Key-Value Database*. 2024. Disponível em: <https://napoleon.com.br/glossario/o-que-e-key-value-database/>. Citado na página 17.

ONGARO, D.; OUSTERHOUT, J. In search of an understandable consensus algorithm. In: *Proceedings of the USENIX Annual Technical Conference*. [S.l.: s.n.], 2014. p. 305–320. Citado na página 12.

Uber Technologies, Inc. *Zap Logger*. 2024. Acesso em: 09 dez. 2024. Disponível em: <https://github.com/uber-go/zap>. Citado na página 23.

BREWER, Eric A.; FOX, Armando. **Lessons from giant-scale services**. *IEEE Internet Computing*, v. 3, n. 4, p. 46-55, 1999. Este artigo introduz o teorema CAP, descrevendo as características de consistência, disponibilidade e tolerância a partições, relevantes para sistemas distribuídos. Citado na página 12.

ONGARO, Diego; OUSTERHOUT, John. **In search of an understandable consensus algorithm**. *Proceedings of the USENIX Annual Technical Conference*, p. 305-320, 2014. Este trabalho apresenta o algoritmo Raft, um protocolo de consenso que visa garantir a consistência em sistemas distribuídos de maneira simplificada e compreensível. Citado na página 12.

HUNT, Patrick et al. **ZooKeeper: Wait-free coordination for Internet-scale systems**. *Proceedings of the USENIX Annual Technical Conference*, p. 145-158, 2010. Este estudo discute o Zab, algoritmo de consenso utilizado pelo ZooKeeper para coordenar sistemas distribuídos, com ênfase na consistência e tolerância a falhas. Citado na página 12.