Documentação Técnica Completa
Sistema de Arquivos Distribuído BigFS

wender13

30 de junho de 2025

Sumário

1 Introdução

O **BigFS** é uma simulação funcional de um sistema de arquivos distribuído, desenvolvido em Python com gRPC. O projeto demonstra na prática os princípios de escalabilidade, tolerância a falhas, replicação e gerenciamento de dados em larga escala. A arquitetura final adota um modelo de **3 camadas** (Cliente-Gateway-Backend), inspirando-se em conceitos de sistemas robustos como HDFS e na usabilidade de ferramentas modernas como o Minio Client. Este documento serve como um manual técnico detalhado da arquitetura e implementação do sistema.

2 Guia de Instalação e Uso

Esta seção detalha os passos para configurar o ambiente e executar o projeto em sistemas Linux, macOS ou Windows.

2.1 Pré-requisitos

- Git
- Python 3.8 ou superior

2.2 Instalação

1. **Clonar o Repositório:**

   ```
   git clone https://github.com/wender13/BigFS.git
   cd BigFS/
   ```

2. **Criar e Ativar Ambiente Virtual:**

   No Linux/macOS: `python3 -m venv venv` e `source venv/bin/activate`

   No Windows: `python -m venv venv` e `.\venv\Scripts\Activate.ps1`

3. **Instalar Dependências:**

   ```
   pip install grpcio grpcio-tools
   ```

4. **Compilar Contrato gRPC:** Este passo crucial traduz a API definida em `.proto` para código Python. Deve ser executado da raiz do projeto.

   ```
   python3 -m grpc_tools.protoc -Iproject --python_out=project
   --grpc_python_out=project project/bigfs.proto
   ```

2.3 Execução

Inicie os componentes em terminais separados, na ordem especificada:

1. **Metadata Server:** `python3 project/metadata_server.py`
2. **Storage Nodes (mínimo 3):** `python3 project/storage_node.py localhost 50052 localhost:50051`
3. **Gateway Server:** `python3 project/gateway_server.py`
4. **Cliente Interativo:** `python3 project/client.py localhost:50050` (conecta-se ao Gateway)

3 Arquitetura Detalhada e Componentes

O BigFS opera em um modelo de 3 camadas que desacopla as responsabilidades de forma clara.

3.1 Camada 1: Cliente Interativo (`client.py`)

O cliente é a interface de usuário do sistema, implementado como um shell interativo com a biblioteca `cmd` do Python. Ele traduz comandos amigáveis (ex: `cp, ls, get`) em chamadas de API gRPC para o Gateway. Toda a complexidade do backend é abstraída do usuário.

3.2 Camada 2: Gateway Server (`gateway_server.py`)

Atua como o ponto de entrada único e a camada de lógica de negócio. Sua principal função é receber arquivos completos do cliente, assumir o trabalho pesado de **particioná-los em chunks** e orquestrar a comunicação com o backend para o armazenamento distribuído e replicado. Ele também atua como um proxy para comandos de metadados, como `ls` e `mkdir`.

3.3 Camada 3: Backend Distribuído

3.3.1 Metadata Server (`metadata_server.py`)

O cérebro do cluster. Ele não armazena dados, apenas metadados. Suas responsabilidades são:

- Manter a estrutura do sistema de arquivos em uma árvore de **Inodes** em memória.
- Gerenciar a localização de cada chunk, incluindo seu nó primário e suas réplicas.
- Monitorar a saúde dos Storage Nodes via **heartbeats** e detectar falhas.
- Implementar a lógica de alocação de dados, escolhendo os nós menos ocupados para novas escritas.

3.3.2 Storage Node (`storage_node.py`)

O músculo do cluster. É um serviço simples e robusto com duas funções principais:

- Armazenar e servir os chunks de dados que lhe são designados.
- Executar a **replicação assíncrona**: quando atua como nó primário, ele é responsável por copiar os dados para os nós de réplica em threads de segundo plano.

4 Análise do Fluxo de Operações: O Caminho de um 'cp'

Para entender como os componentes se conectam, vamos rastrear o comando `cp arquivo.txt bfs://docs/remoto.txt`.

1. **Cliente:** O método `do_cp` no shell chama `client.copy_to_bigfs()`. Esta função abre o arquivo local e inicia um **stream gRPC** para o método `UploadFile` do Gateway. O primeiro item do stream contém os metadados (caminho de destino) e os seguintes contêm os dados do arquivo em pedaços.
2. **Gateway Server:** O método `UploadFile` recebe o stream. Ele salva os dados recebidos em um arquivo temporário em seu próprio disco.
3. **Gateway → Metadata Server:** Com o arquivo completo, o Gateway contata o Metadata Server através do RPC `GetWritePlan`, enviando o caminho e o tamanho total do arquivo.
4. **Metadata Server:** A lógica de `GetWritePlan` é executada:
   - A lista de Storage Nodes ativos é ordenada pela menor quantidade de chunks.
   - Os nós menos ocupados são selecionados para serem o Primário e as Réplicas.
   - Um "plano de escrita"(o mapa de chunks e suas localizações) é gerado e retornado ao Gateway.
5. **Gateway → Storage Nodes:** O Gateway lê o arquivo temporário, o particiona em chunks e, para cada chunk, envia um RPC `StoreChunk` para o nó Primário correspondente, incluindo a lista de endereços das Réplicas.
6. **Storage Node (Primário):** Ao receber o `StoreChunk`, ele salva o dado em seu disco e imediatamente inicia **threads em background** para replicar aquele chunk para os nós de Réplica indicados.
7. **Conclusão:** O Gateway, após distribuir todos os chunks, apaga seu arquivo temporário e retorna uma mensagem de sucesso ao Cliente, que a exibe ao usuário.

5 Conceitos Fundamentais Implementados

5.1 Tolerância a Falhas por Replicação

O sistema garante a **durabilidade** dos dados ao replicar cada chunk 3 vezes (configurável) em nós distintos. Para garantir a **disponibilidade**, o cliente implementa um mecanismo de **failover**: durante uma leitura, se o nó primário de um chunk falhar, o cliente automaticamente tenta a leitura a partir do próximo nó de réplica em sua lista, tornando a falha transparente para o usuário.

5.2 Balanceamento de Carga por Ocupação

Diferente de uma alocação aleatória, o Metadata Server ativamente monitora a quantidade de chunks em cada Storage Node (via heartbeats enriquecidos). Ao alocar novos arquivos, ele sempre prioriza os nós com menos chunks armazenados, distribuindo a carga de armazenamento de forma mais equilibrada pelo cluster.