Documentação Técnica Detalhada
`gateway_server.py`

wender13

1 de julho de 2025

Sumário

1 Introdução e Papel na Arquitetura

O arquivo `gateway_server.py` implementa a camada intermediária inteligente da arquitetura de 3 camadas do BigFS. Ele atua como o único ponto de entrada para o cliente, abstraindo toda a complexidade do backend distribuído.

Suas principais responsabilidades são receber arquivos do cliente, orquestrar o particionamento (*sharding*) e a distribuição dos dados para os `Storage Nodes`, e atuar como um proxy para requisições de metadados direcionadas ao `Metadata Server`.

2 Análise da Classe `GatewayService`

Esta classe implementa a API do serviço `GatewayService` definido no arquivo `.proto`.

2.1 `__init__(self)`

O construtor prepara o Gateway para operar, estabelecendo sua conexão com o cérebro do sistema.

```
class GatewayService(bigfs_pb2_grpc.GatewayServiceServicer):
    def __init__(self):
        # 1. Conexão com o Metadata Server
        self.metadata_channel = grpc.insecure_channel(
   METADATA_SERVER_ADDRESS)
        self.metadata_stub = bigfs_pb2_grpc.MetadataServiceStub(self.
   metadata_channel)

        # 2. Criação do diretório temporário
        self.temp_dir = "gateway_temp"
        if not os.path.exists(self.temp_dir):
            os.makedirs(self.temp_dir)
```

Explicação:

1. Ao ser inicializado, o Gateway imediatamente cria um canal de comunicação gRPC para o `Metadata Server`. O objeto `self.metadata_stub` funciona como um "controle remoto", permitindo que o Gateway chame as funções do Mestre (como `GetWritePlan`).

2. Ele também garante a existência de um diretório `gateway_temp`, que será usado para armazenar temporariamente os arquivos durante o processo de upload.

2.2 `UploadFile(self, request_iterator, context)`

Este é o método mais complexo, orquestrando todo o fluxo de ingestão e distribuição de um arquivo.

```python
def UploadFile(self, request_iterator, context):
    # Cria um nome de arquivo único para armazenamento temporário
    temp_filename = os.path.join(self.temp_dir, str(uuid.uuid4()))
    try:
        # Etapa 1: Recebe o stream completo do cliente
        first_chunk = next(request_iterator)
        remote_path = first_chunk.metadata.remote_path
        file_size = 0
        with open(temp_filename, 'wb') as f:
            for chunk in request_iterator:
                f.write(chunk.data); file_size += len(chunk.data)

        # Etapa 2: Pede um plano de escrita ao Metadata Server
        file_req = bigfs_pb2.FileRequest(filename=remote_path, size=
    file_size)
        write_plan = self.metadata_stub.GetWritePlan(file_req)
        if not write_plan.locations:
            raise Exception("Falha ao obter plano de escrita.")

        # Etapa 3: Particiona e distribui os chunks
        with open(temp_filename, 'rb') as f:
            for loc in write_plan.locations:
                chunk_data = f.read(CHUNK_SIZE_BYTES)
                with grpc.insecure_channel(loc.primary_node_id) as channel:
                    stub = bigfs_pb2_grpc.StorageServiceStub(channel)
                    chunk_msg = bigfs_pb2.Chunk(...)
                    stub.StoreChunk(chunk_msg)
        return bigfs_pb2.SimpleResponse(success=True)
    except Exception as e:
        # ... tratamento de erro ...
    finally:
        # Etapa 4: Limpeza
        if os.path.exists(temp_filename): os.remove(temp_filename)
```

Explicação do Fluxo:

Etapa 1: Recebimento do Stream O método recebe um iterador do cliente. A primeira mensagem contém os metadados (nome do arquivo de destino). As mensagens seguintes contêm os dados brutos, que são escritos em um único arquivo temporário no disco do Gateway.

Etapa 2: Obtenção do Plano Com o arquivo completo e seu tamanho total, o Gateway age como um cliente para o Metadata Server, chamando `GetWritePlan` para receber as instruções de como particionar e para quais nós enviar os dados.

Etapa 3: Particionamento e Distribuição O Gateway lê o seu próprio arquivo temporário, pedaço por pedaço (`f.read(CHUNK_SIZE_BYTES)`). Este é o momento em que o **sharding** acontece. Para cada chunk lido, ele envia um RPC `StoreChunk` para o nó primário correto, conforme as instruções do plano.

Etapa 4: Limpeza O bloco `finally` garante que, independentemente de sucesso ou falha, o arquivo temporário seja sempre apagado, mantendo o servidor limpo.

2.3 Funções de Proxy (`GetDownloadMap`, `ListFiles`)

Para operações que não envolvem manipulação de dados, o Gateway atua como um simples intermediário.

```
def GetDownloadMap(self, request, context):
    return self.metadata_stub.GetFileLocation(request)

def ListFiles(self, request, context):
    return self.metadata_stub.ListFiles(request)
```

Explicação: Estes métodos são eficientes pois não possuem lógica própria. Eles simplesmente recebem a requisição do cliente e a repassam diretamente para o método correspondente no Metadata Server, devolvendo a resposta ao cliente. Isso mantém a lógica de metadados centralizada no Mestre.

3 Inicialização do Servidor

O bloco final do arquivo contém o código padrão para iniciar o serviço gRPC.

```
def serve():
    server = grpc.server(futures.ThreadPoolExecutor(max_workers=10))
    bigfs_pb2_grpc.add_GatewayServiceServicer_to_server(GatewayService(),
    server)
    server.add_insecure_port('[::]:50050'); server.start()
    print("       Gateway Server escutando na porta 50050.")
    server.wait_for_termination()

if __name__ == '__main__':
    serve()
```

Explicação:

- `grpc.server(...)`: Cria uma instância de servidor com um pool de 10 threads para lidar com requisições concorrentes.
- `add_..._to_server(...)`: Registra a nossa classe `GatewayService` no servidor, "ligando"os métodos da nossa API.
- `add_insecure_port(...)`: Vincula o servidor à porta `50050` em todas as interfaces de rede.
- `server.start() ... wait_for_termination()`: Inicia o servidor e o mantém rodando indefinidamente, aguardando por requisições.