Documentação Técnica Detalhada
`metadata_server.py`

wender13

1 de julho de 2025

Sumário

1 Introdução e Papel na Arquitetura

O arquivo `metadata_server.py` implementa o cérebro do cluster BigFS. Ele atua como o Mestre na arquitetura, sendo o único componente com uma visão global do sistema. Suas responsabilidades centrais são gerenciar o namespace dos arquivos, a localização de todos os *chunks* de dados e monitorar o estado de saúde de todos os `Storage Nodes`. Este servidor não armazena dados de arquivos, apenas as informações sobre eles, permitindo alta performance em operações de metadados.

2 Análise da Classe `MetadataService`

Esta classe implementa a API do serviço `MetadataService`, respondendo às requisições do `Gateway Server`.

2.1 Inicialização e Estruturas de Dados

O construtor `__init__` estabelece as estruturas de dados fundamentais para a operação do servidor.

```
class MetadataService(bigfs_pb2_grpc.MetadataServiceServicer):
    def __init__(self):
        # Dicionário para rastrear os nós ativos e seu status
        self.storage_nodes = {}
        # Dicionário para mapear nomes de arquivo para suas listas de chunks
        self.file_to_chunks = defaultdict(list)
        # Trava para garantir operações seguras em ambiente multithread
        self.lock = threading.Lock()
        # Inicia a thread de monitoramento em segundo plano
        threading.Thread(target=self._check_dead_nodes, daemon=True).start()
```

Explicação:

- `self.storage_nodes`: Armazena o estado de cada nó de armazenamento, incluindo o horário do último *heartbeat* e a contagem de chunks reportada. Ex: `{'node1': {'last_seen': 123, 'chunk_count': 10}}`.
- `self.file_to_chunks`: É o mapa principal do sistema de arquivos. Utiliza um namespace plano onde cada chave é o nome de um arquivo e o valor é uma lista de objetos `ChunkLocation`.
- `threading.Thread(...)`: Inicia o processo "vigia"que irá monitorar a saúde dos nós de forma contínua, sem bloquear o servidor principal de responder às requisições.

2.2 Monitoramento do Cluster

As funções `RegisterNode` e `_check_dead_nodes` trabalham juntas para manter uma lista precisa de nós ativos.

2.2.1 RegisterNode(self, request, context)

Este método é o ponto de entrada para os *heartbeats*.

```
def RegisterNode(self, request, context):
    with self.lock:
        self.storage_nodes[request.address] = {
            'last_seen': time.time(),
            'chunk_count': request.chunk_count
        }
        print(f"[Metadata] Heartbeat de: {request.address}...")
```

Explicação: Ao ser chamado por um `Storage Node`, este método adiciona ou atualiza o status daquele nó no dicionário `storage_nodes`, registrando o horário atual e a contagem de chunks reportada.

2.2.2 _check_dead_nodes(self)

Este método é o "vigia"que roda em segundo plano.

```
def _check_dead_nodes(self):
    while True:
        time.sleep(HEARTBEAT_TIMEOUT)
        with self.lock:
            now = time.time()
            dead_nodes = [nid for nid, status in self.storage_nodes.items()
                          if now - status['last_seen'] > HEARTBEAT_TIMEOUT]
            if dead_nodes:
                print(f"[Metadata] Nós inativos detectados: {dead_nodes}")
                for nid in dead_nodes:
                    if nid in self.storage_nodes:
                        del self.storage_nodes[nid]
```

Explicação: A cada 15 segundos, a função percorre a lista de nós e remove qualquer um cujo último heartbeat tenha sido recebido há mais de 15 segundos. Isso garante que o sistema não tente usar nós que falharam.

2.3 Lógica de Escrita e Balanceamento de Carga (GetWritePlan)

Este é um dos métodos mais inteligentes do sistema, responsável por decidir como e onde um novo arquivo será armazenado.

```
def GetWritePlan(self, request, context):
    with self.lock:
        active_nodes_status = list(self.storage_nodes.items())
        if len(active_nodes_status) < REPLICATION_FACTOR:
             # ... retorna erro ...

        # 1. Ordena os nós pelo número de chunks, do menor para o maior
        sorted_nodes = sorted(active_nodes_status,
                              key=lambda item: item[1]['chunk_count'])
        available_node_addrs = [node_id for node_id, status in sorted_nodes]
```

```
        num_chunks = # ... calcula o número de chunks ...
        plan = []
        for i in range(num_chunks):
            # 2. Seleciona os N nós menos carregados
            chosen_nodes = available_node_addrs[:REPLICATION_FACTOR]
            primary, replicas = chosen_nodes[0], chosen_ones[1:]
            # ... cria a localização do chunk e adiciona ao plano ...

            # 3. Rotaciona a lista para balancear escritas de um mesmo arquivo
            available_node_addrs = available_node_addrs[1:] + available_node_addrs[:1]

        self.file_to_chunks[request.filename] = plan
        return bigfs_pb2.FileLocationResponse(...)
```

Explicação: Em vez de uma seleção aleatória, o método primeiro ordena os nós ativos com base na contagem de chunks que eles reportaram. Em seguida, para cada chunk a ser escrito, ele seleciona os `N` nós no topo dessa lista (os mais "vazios"). A rotação da lista a cada iteração garante que, para um arquivo grande, os chunks primários sejam distribuídos entre os nós de menor carga, em vez de todos irem para um único nó.

2.4 Lógica de Leitura e Failover (`GetFileLocation`)

Este método fornece o mapa para um cliente que deseja ler um arquivo, garantindo que os endereços fornecidos sejam de nós ativos.

```
def GetFileLocation(self, request, context):
    with self.lock:
        # ... busca o arquivo no dicionário ...
        locations = self.file_to_chunks[request.filename]
        # Itera sobre o plano do arquivo para validar os nós
        for loc in locations:
            # Se o primário registrado não está mais na lista de nós ativos ...
            if loc.primary_node_id not in self.storage_nodes:
                promoted = False
                # ...procura uma réplica que esteja viva...
                for replica in loc.replica_node_ids:
                    if replica in self.storage_nodes:
                        # ...e a promove para primário APENAS nesta resposta.
                        loc.primary_node_id = replica
                        promoted = True
                        break
                # Se nenhuma cópia estiver online, retorna erro
                if not promoted:
                    # ... retorna erro ...
```

```
20          return bigfs_pb2.FileLocationResponse(...)
```

Explicação: A lógica de failover acontece aqui. Antes de enviar o mapa de volta, o servidor verifica se o nó primário de cada chunk ainda está online. Se não estiver, ele modifica a resposta em tempo real, promovendo a primeira réplica saudável da lista para a posição de primário. Isso garante que o cliente sempre receba um endereço funcional para tentar a leitura.