Documentação Técnica Detalhada
`client.py`

wender13

1 de julho de 2025

Sumário

1 Introdução e Visão Geral

O arquivo `client.py` é o ponto de entrada do usuário para interagir com o sistema de arquivos BigFS. Ele foi projetado com uma clara separação de responsabilidades em duas classes principais:

- **`BigFSClient`:** A classe que encapsula toda a lógica de comunicação com o backend distribuído (Metadata Server e Storage Nodes).
- **`BigFSShell`:** A classe que utiliza o módulo `cmd` do Python para criar uma interface de usuário de shell interativo, amigável e profissional.

Nesta arquitetura, o cliente é considerado "gordo"(fat client), pois detém a lógica de particionamento de arquivos (sharding) e a orquestração de transferências paralelas.

2 Análise da Classe `BigFSClient`

Esta classe é o motor do cliente. Ela lida com todas as chamadas gRPC e a lógica de manipulação de dados.

2.1 `__init__(self, metadata_address)`

O construtor estabelece a conexão inicial com o cérebro do cluster.

```
def __init__(self, metadata_address):
    try:
        # Nota: Nesta arquitetura, o cliente conecta-se diretamente
        # ao Metadata Server para obter os planos de I/O.
        self.metadata_channel = grpc.insecure_channel(metadata_address)
        grpc.channel_ready_future(self.metadata_channel).result(timeout=5)
        self.metadata_stub = bigfs_pb2_grpc.MetadataServiceStub(self.
    metadata_channel)
    except grpc.FutureTimeoutError:
        print(f"Erro fatal: Metadata Server em {metadata_address} não
    encontrado.");
        sys.exit(1)
```

Explicação: Ao ser instanciado, o cliente cria um canal de comunicação gRPC para o endereço do `Metadata Server`. O `timeout=5` garante que o programa encerre de forma limpa se o servidor não estiver acessível, em vez de congelar indefinidamente. O `self.metadata_stub` é o objeto "controle remoto"para chamar as funções do mestre.

2.2 `copy_to_bigfs(self, local, remote)`

Este método orquestra o processo completo de upload, incluindo o particionamento (sharding) dos dados.

```python
def copy_to_bigfs(self, local, remote):
    if not os.path.exists(local):
        print(f"Erro: Arquivo local '{local}' não encontrado.")
        return

    file_size = os.path.getsize(local)
    # 1. Pede ao Metadata Server um plano de escrita
    plan = self.metadata_stub.GetWritePlan(
        bigfs_pb2.FileRequest(filename=remote, size=file_size)
    )
    if not plan.locations:
        print("Falha ao obter plano de escrita do servidor.")
        return

    # 2. Abre o arquivo local e envia os chunks em paralelo
    with open(local, 'rb') as f, ThreadPoolExecutor(10) as executor:
        for loc in plan.locations:
            data = f.read(CHUNK_SIZE_BYTES)
            # 3. Submete a escrita de cada chunk para uma thread separada
            executor.submit(
                self._write_chunk,
                loc.primary_node_id,
                loc.chunk_id,
                data,
                loc.replica_node_ids
            )
```

Explicação:

1. **Planejamento:** O cliente primeiro obtém o tamanho do arquivo local e solicita ao Metadata Server um plano de escrita. O servidor responde com uma lista de `ChunkLocation`, detalhando para onde cada chunk deve ser enviado.

2. **Particionamento e Paralelismo:** O cliente abre o arquivo local e, dentro de um loop, lê o arquivo pedaço por pedaço (‘f.read(CHUNK_SIZE_BYTES)‘). Este é o momento em que o **sharding** acontece.

3. **Distribuição:** Para cada chunk lido, ele submete uma nova tarefa a um pool de threads. Cada thread executa a função auxiliar `_write_chunk`, enviando o dado diretamente ao ‘Storage Node‘ primário designado no plano. Isso permite que múltiplos chunks sejam enviados para múltiplos nós simultaneamente.

2.3 `get_from_bigfs(self, remote, local)`

Orquestra o download e a reconstrução de um arquivo.

```python
# O código deste método, como apresentado anteriormente,
# já está correto para esta análise.
def get_from_bigfs(self, remote, local):
```

```
    # 1. Obtém o mapa do arquivo do Metadata Server
    plan = self.metadata_stub.GetFileLocation(bigfs_pb2.FileRequest(filename
    =remote))
    if not plan.locations: print("Arquivo não encontrado."); return

    chunks = {i: None for i in range(len(plan.locations))}
    # 2. Usa um pool de threads para baixar todos os chunks em paralelo
    with ThreadPoolExecutor(10) as ex:
        futs = {ex.submit(self._read_chunk, loc): loc for loc in plan.
    locations}
        for f in futs:
            # 3. Coleta os resultados
            index, data = f.result()
            chunks[index] = data

    if any(c is None for c in chunks.values()): print("Falha ao baixar.");
    return

    # 4. Monta o arquivo final na ordem correta
    with open(local, 'wb') as f:
        [f.write(chunks[i]) for i in sorted(chunks.keys())]
```

Explicação: O processo é o inverso da escrita. O cliente primeiro obtém o mapa de localização do arquivo. Em seguida, ele dispara múltiplas tarefas em threads para baixar cada chunk. A função `_read_chunk` (não mostrada aqui, mas presente no código final) contém a lógica de **failover**, tentando as réplicas se o nó primário falhar. Finalmente, os chunks baixados são escritos no arquivo local na ordem correta, especificada pelo `index`.

3 Análise da Classe `BigFSShell`

Esta classe cria a interface de usuário (UI) interativa.

```
class BigFSShell(cmd.Cmd):
    prompt = 'bigfs > '
    intro = 'Bem-vindo ao shell do BigFS.'

    def __init__(self, client):
        super().__init__()
        self.client = client

    def do_cp(self, arg):
        'Uso: cp <local> <remoto_bfs://>'
        try:
            args = shlex.split(arg)
            if len(args) == 2:
                self.client.copy_to_bigfs(args[0], args[1])
            else:
                print("Uso: cp <arquivo_local> <caminho_remoto_bfs://>")
```

```
        except Exception as e:
            print(f"Erro: {e}")

    # ... outros comandos como do_ls, do_get, do_quit ...
```

Explicação:

- **Herança de `cmd.Cmd`:** Ela herda toda a funcionalidade de um shell, como histórico de comandos e o loop principal.
- **Métodos `do_<comando>`:** Cada comando disponível para o usuário (como 'cp', 'ls') é implementado como um método que começa com 'do_'.**Parsing de Argumentos:***Utiliz*

- **Delegação:** A principal tarefa de cada método `do_` é validar os argumentos e então chamar o método correspondente na instância de `BigFSClient` (`self.client`), delegando o trabalho pesado.
- **Ajuda Automática:** O texto na primeira linha de cada método `do_` (a docstring) é automaticamente usado como texto de ajuda quando o usuário digita `help cp`.

4 Ponto de Entrada Principal (`main`)

A função `main` e o bloco `if __name__ == '__main__':` são o ponto de partida do script.

```
def main():
    if len(sys.argv) != 2:
        print("Uso: python3 -m project.client <metadata_address>")
        sys.exit(1)

    # 1. Cria a instância do cliente com a lógica de negócio
    client = BigFSClient(sys.argv[1])
    # 2. Inicia o shell, passando o cliente como argumento
    BigFSShell(client).cmdloop()

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

Explicação: O código primeiramente verifica se o endereço do Metadata Server foi fornecido. Em seguida, ele cria uma única instância do `BigFSClient`, que estabelece a conexão com o servidor. Finalmente, ele passa essa instância para o `BigFSShell` e chama `.cmdloop()`, que inicia o loop interativo e fica aguardando os comandos do usuário.