Report: RPC File Transfer Implementation

1. How you design your RPC service

To implement an efficient file transfer system, we designed a **gRPC service** using **Protocol Buffers (Protobuf)** as the Interface Definition Language (IDL).

Design Rationale: Instead of loading the entire file into memory (which is inefficient for large files) or sending separate requests for metadata and data, we utilized **Client-side Streaming RPC**. This allows the client to send a continuous stream of messages to the server, while the server listens and processes them sequentially.

We implemented a specific message structure using the `oneof` keyword. This allows us to send two different types of information within the same stream:

1. **Metadata:** The first message contains the `FileInfo` (filename).
2. **File Content:** Subsequent messages contain `chunk_data` (raw binary bytes).

Figure 1: RPC Service Design

```
sequenceDiagram
    participant Client
    participant Server

    Note over Client, Server: RPC Method: Upload (Stream)
    Client->>Server: Stream Message 1: Metadata (filename)
    Client->>Server: Stream Message 2: Chunk Data (1MB)
    Client->>Server: Stream Message 3: Chunk Data (1MB)
    Client-->Server: ... (Remaining Chunks) ...
    Client->>Server: End of Stream
    Server-->>Client: Response: UploadStatus (Success)
```

2. How you organize your system

The system is organized into a standard Client-Server architecture, decoupled by the gRPC interface.

- **Shared Interface (`.proto`):** Defines the contract between client and server. Both sides rely on the generated Python code (Stubs/Skeletons) from this file.
- **Client Side:** Responsible for file I/O (reading) and logic. It utilizes a Python Generator to read the file in small chunks (e.g., 1MB) and yields `UploadRequest` messages to the gRPC stub.

- **Server Side:** Responsible for file I/O (writing) and concurrency. It uses a `ThreadPoolExecutor` to handle multiple clients. The implementation logic iterates through the incoming request stream, differentiates between metadata and data, and reconstructs the file on the server's disk.

Figure 2: System Organization

```
graph LR
    subgraph Client System
        A[File on Disk] -->|Read 1MB| B[Client Logic]
        B -->|Generator| C[gRPC Stub]
    end

    C -->|Network Stream| D[gRPC Channel]

    subgraph Server System
        D --> E[gRPC Skeleton]
        E -->|Request Iterator| F[Server Logic]
        F -->|Write Bytes| G[File on Disk]
    end
```

3. How you implement the file transfer

The implementation is broken down into three key components: the Protocol Buffer definition, the Client logic, and the Server logic.

A. Protocol Buffer Definition (`file_transfer.proto`) We define the `Upload` service and the `UploadRequest` message using `oneof` to handle both file info and binary data in a single stream.

```
syntax = "proto3";
package file_transfer;

service FileTransfer {
  // Client-side streaming RPC
  rpc Upload (stream UploadRequest) returns (UploadStatus) {}
}

message UploadRequest {
  // "oneof" ensures only one field is set per message
  oneof request_type {
    FileInfo info = 1;
    bytes chunk_data = 2;
  }
}
```

```
message FileInfo {
  string filename = 1;
}

message UploadStatus {
  string message = 1;
  bool success = 2;
}
```

B. Client Implementation (Streaming Generator) The client reads the file and yields the

```
import grpc
import os
import file_transfer_pb2
import file_transfer_pb2_grpc

# Hàm này sinh ra các gói tin (generator)
def generate_requests(file_path):
    # 1. Xử lý tên file
    # Lấy ra: "boc_phot.mp4" - Gộp cả tên và đuôi [cite: 449]
    filename_only = os.path.basename(file_path)

    # 2. Gửi gói tin đầu tiên: CHỈ CHỨA THÔNG TIN FILE
    print(f"Client: Đang gửi thông tin file '{filename_only}'...")
    yield file_transfer_pb2.UploadRequest(
        info=file_transfer_pb2.FileInfo(filename=filename_only)
    )

    # 3. Gửi các gói tin tiếp theo: CHỨA DỮ LIỆU FILE (CHUNKS)
    print("Client: Đang xả hàng...")
    with open(file_path, 'rb') as f:
        while True:
            # Cắt mỗi miếng 1MB (1024*1024 bytes)
            chunk = f.read(1024 * 1024)
            if not chunk:
                break # Hết phim

            # Gói vào UploadRequest và bắn đi
            yield file_transfer_pb2.UploadRequest(chunk_data=chunk)

def run():
    # Kết nối tới cái động bàn tơ của Server
    #[cite_start]# [cite: 413] Create stub functions
    with grpc.insecure_channel('localhost:50051') as channel:
```

```
        stub = file_transfer_pb2_grpc.FileTransferStub(channel)

        # Nhập tên file cần gửi (nhớ là file phải tồn tại nhé các thần
đồng)
        file_to_send = input("Nhap file de gui vao day:")

        # Kiểm tra xem file có tồn tại không đã, không lại lỗi sấp mặt
        if not os.path.exists(file_to_send):
            print("Client: Ủa file đâu? Nhập sai đường dẫn rồi cha
nội!")
            return

        try:
            # Gọi hàm Upload và truyền vào cái máy bắn đá (generator)
            response = stub.Upload(generate_requests(file_to_send))

            # In ra phán quyết cuối cùng của Server
            print(f"Server phản hồi: {response.message}")

        except grpc.RpcError as e:
            print(f"Toang rồi ông giáo ạ! Lỗi RPC: {e}")

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

C. Server Implementation (Stream Consumption) The server iterates through the stream. It expects the `info` field first to open the file, then writes incoming `chunk_data`.

```
import grpc
from concurrent import futures
import file_transfer_pb2
import file_transfer_pb2_grpc

# Class này kế thừa từ bộ khung xương (Skeleton) mà gRPC tạo ra
class
FileTransferServicer(file_transfer_pb2_grpc.FileTransferServicer):

    def Upload(self, request_iterator, context):
        # request_iterator chính là cái vòi nước đang xả dữ liệu từ
Client sang

        file_object = None
```

```
        final_name = "unknown.bin" # Tên mặc định phòng khi Client bị
ngáo

        print("Server: Đang mở cổng kết nối... Chờ hàng về.")

        for request in request_iterator:
            # KIỂM TRA: Gói này là Tên file hay là Dữ liệu?
            if request.HasField("info"):
                # Đây là Metadata (Tên file)
                final_name = request.info.filename
                print(f"Server: Phát hiện mục tiêu! Đang chuẩn bị nhận
file '{final_name}'")

                # Mở file với chế độ 'wb' (write binary) - Ghi đè không
thương tiếc
                file_object = open(final_name, 'wb')

            elif request.HasField("chunk_data"):
                # Đây là Thịt (Dữ liệu)
                if file_object:
                    file_object.write(request.chunk_data)
                else:
                    # Nếu Client ném dữ liệu trước khi ném tên -> Chửi
ngay
                    print("Server: Ê! Gửi cái tên file trước đi bạn
ơi!")
                    return file_transfer_pb2.UploadStatus(
                        success=False,
                        message="Lỗi quy trình: Không thấy tên file đâu
cả!"
                    )

        # Sau khi vòng lặp kết thúc (Client ngừng gửi)
        if file_object:
            file_object.close()
            print(f"Server: Đã nuốt trọn file '{final_name}'.")

        # Trả lời Client một câu cho nó yên tâm
        return file_transfer_pb2.UploadStatus(
            success=True,
            message=f"Upload thành công file {final_name}. Server đã
nhận đủ!"
        )
```

```python
def serve():
    # Khởi tạo Server với 10 luồng xử lý - Mạnh hơn cả dàn PC đào coin
    server = grpc.server(futures.ThreadPoolExecutor(max_workers=10))
    file_transfer_pb2_grpc.add_FileTransferServicer_to_server(
        FileTransferServicer(), server
    )

    # Mở port 50051
    server.add_insecure_port('[::]:50051')
    print("Server đang rình rập tại port 50051...")
    server.start()
    server.wait_for_termination()

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