Practical Work 2: RPC File Transfer

Group ID: 18

1 RPC Service Design

This file transfer system uses **gRPC** (Google Remote Procedure Call) framework with Protocol Buffers for efficient communication.

1.1 Architecture Overview

The RPC architecture consists of:

- **Protocol Buffer Definition** (.proto file): Defines the service interface and message types
- **gRPC Server**: Implements the RPC methods and handles file reception
- **gRPC Client**: Calls RPC methods to upload files
- **Generated Code**: Auto-generated from .proto file for serialization/deserialization

1.2 RPC Communication Flow

```
Client                                Server
  |                                     |
  |  Create gRPC Channel                |
  |------------------------------------>|
  |                                     | gRPC Server Running
  |                                     | Listening on port 50051
  |                                     |
  |  Call UploadFile() RPC              |
  |  (Client-side streaming)            |
  |------------------------------------>|
  |                                     |
  |  Stream FileChunk #1                |
  |------------------------------------>| Receive chunk
  |                                     |
  |  Stream FileChunk #2                |
  |------------------------------------>| Receive chunk
  |                                     |
  |  Stream FileChunk #3 (is_last=True) |
  |------------------------------------>| Receive chunk
```

```
|                                     | Assemble file
|                                     | Save to disk
|                                     |
|  <--- UploadResponse (success)      |
|<------------------------------------|
|                                     |
|  Close channel                      |
|------------------------------------>|
```

1.3 Protocol Buffer Messages

The system uses the following message types:

```
message FileChunk {
    string filename = 1;     // File name
    bytes content = 2;       // Chunk data
    bool is_last = 3;        // Last chunk flag
}

message UploadResponse {
    bool success = 1;        // Success status
    string message = 2;      // Response message
}

service FileTransferService {
    rpc UploadFile(stream FileChunk)
        returns (UploadResponse);
}
```

Listing 1: Protocol Buffer Definition

2 System Organization

2.1 Directory Structure

```
practical2/RPC/
|-- proto/
|   |-- file_transfer.proto      # Protocol Buffer definition
|-- generated/
|   |-- __init__.py
|   |-- file_transfer_pb2.py     # Generated messages
|   |-- file_transfer_pb2_grpc.py # Generated service
|-- server.py                     # gRPC server
|-- client.py                     # gRPC client
|-- test_file.txt                 # Test file
|-- requirements.txt              # Dependencies
|-- 02.rpc.file.transfer.tex     # This report
```

2.2 Component Interaction

1. **.proto file**: Defines the contract between client and server
2. **protoc compiler**: Generates Python code from .proto file
3. **Generated code**: Provides message classes and service stubs
4. **Server**: Implements FileTransferServicer class
5. **Client**: Uses FileTransferServiceStub to call RPC methods

2.3 Data Flow

```
[Client File]
    --> Read in chunks (4096 bytes)
    --> Create FileChunk messages
    --> Stream via gRPC channel
    --> [Server receives stream]
    --> Assemble chunks
    --> Write to disk
    --> [Received File]
```

3 Implementation

3.1 Server Implementation

The server implements the `FileTransferServicer` class:

```
class FileTransferServicer(
    file_transfer_pb2_grpc.FileTransferServiceServicer):

    def UploadFile(self, request_iterator, context):
        """
        RPC method to receive file from client
        request_iterator: stream of FileChunk from client
        """
        filename = None
        received_data = bytearray()

        # Receive chunks from client stream
        for chunk in request_iterator:
            if filename is None:
                filename = chunk.filename

            # Append chunk data
            received_data.extend(chunk.content)

            if chunk.is_last:
                break

        # Save file to disk
```

```python
        output_filename = f"received_{filename}"
        with open(output_filename, 'wb') as f:
            f.write(received_data)

        # Return response
        return file_transfer_pb2.UploadResponse(
            success=True,
            message=f"File uploaded successfully"
        )
```

Listing 2: Server - RPC Method Implementation

```python
def serve():
    # Create gRPC server with thread pool
    server = grpc.server(
        futures.ThreadPoolExecutor(max_workers=10)
    )

    # Register servicer
    file_transfer_pb2_grpc.
       add_FileTransferServiceServicer_to_server(
        FileTransferServicer(), server
    )

    # Bind to address and start
    server.add_insecure_port('127.0.0.1:50051')
    server.start()
    server.wait_for_termination()
```

Listing 3: Server - Start gRPC Server

3.2 Client Implementation

The client uses streaming to send file chunks:

```python
def generate_file_chunks(filename):
    """
    Generator function to create stream of FileChunk
    """
    file_basename = os.path.basename(filename)

    with open(filename, 'rb') as f:
        while True:
            chunk_data = f.read(4096)

            if not chunk_data:
                # Send last chunk with empty content
                yield file_transfer_pb2.FileChunk(
                    filename=file_basename,
                    content=b'',
                    is_last=True
                )
                break

            # Send chunk
            yield file_transfer_pb2.FileChunk(
                filename=file_basename,
                content=chunk_data,
                is_last=False
            )
```

Listing 4: Client - Generate File Chunks

```python
def upload_file(filename):
    # Create gRPC channel
    with grpc.insecure_channel('127.0.0.1:50051') as channel:
        # Create stub (client)
        stub = file_transfer_pb2_grpc.FileTransferServiceStub(
           channel)

        # Call RPC method with streaming
        response = stub.UploadFile(generate_file_chunks(filename)
           )

        # Handle response
        if response.success:
            print(f"Success: {response.message}")
```

Listing 5: Client - Call RPC Method

3.3 Comparison: TCP vs RPC

Aspect	**TCP (Practical 1)**	**RPC (Practical 2)**
Protocol	Raw TCP sockets	gRPC over HTTP/2
Data Format	Raw bytes	Protocol Buffers
Communication	Manual send/recv	RPC method calls
Streaming	Manual chunking	Built-in streaming
Error Handling	Manual	Built-in RPC errors
Code Generation	None	Auto-generated from .proto

4 Who Does What

4.1 Protocol Buffer Compiler (protoc)

- Reads `file_transfer.proto`
- Generates `file_transfer_pb2.py` (message classes)
- Generates `file_transfer_pb2_grpc.py` (service classes)

4.2 Server Responsibilities

- Implements `FileTransferServicer` class

- Overrides `UploadFile()` RPC method
- Receives streaming FileChunk messages
- Assembles chunks into complete file
- Saves file to disk
- Returns UploadResponse to client

4.3 Client Responsibilities

- Creates gRPC channel to server
- Creates FileTransferServiceStub
- Reads file from disk in chunks
- Generates stream of FileChunk messages
- Calls `UploadFile()` RPC method
- Handles UploadResponse from server

4.4 gRPC Framework Responsibilities

- Manages network connections (HTTP/2)
- Handles serialization/deserialization (Protocol Buffers)
- Manages streaming data flow
- Provides error handling and status codes
- Thread pool management for concurrent requests

5 Testing

5.1 Setup

Install dependencies:

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

Generate code from .proto file:

```
python -m grpc_tools.protoc -I./proto
    --python_out=./generated
    --grpc_python_out=./generated
    ./proto/file_transfer.proto
```

5.2 Running the System

1. Open first terminal, run server:

   ```
   cd practical2/RPC
   python server.py
   ```

2. Open second terminal, run client:

   ```
   cd practical2/RPC
   python client.py
   ```

3. Verify that `received_test_file.txt` is created
4. Compare original and received files to verify correctness

5.3 Expected Output

Server output:

```
[SERVER] gRPC Server started on 127.0.0.1:50051
[SERVER] Waiting for clients...
[SERVER] Receiving file: test_file.txt
[SERVER] Received last chunk
[SERVER] File saved successfully: received_test_file.txt
[SERVER] Total bytes received: 407
```

Client output:

```
[CLIENT] Connecting to server 127.0.0.1:50051
[CLIENT] Preparing to send file: test_file.txt (407 bytes)
[CLIENT] Sending chunk 1 (407 bytes)
[CLIENT] Success: File test_file.txt uploaded successfully
```