RPC File Transfer

Tran Hung Thinh (BI12-428)

December 11, 2024

1 Introduction

The RPC File Transfer system uses **gRPC** to enable seamless file transfers between a client and a server. This document describes its design, functionality, and implementation in detail.

2 Service Definition

2.1 Overview

The RPC service is defined using **Protocol Buffers (protobuf)**. It includes the following operations:

- **SendFile:** Transfers a file from the client to the server.
- **ReceiveFile:** Downloads a file from the server to the client in chunks.

2.2 Message Types

The service uses custom protobuf messages:

- **FileRequest:** Contains the file name and file content.
- **FileResponse:** Indicates the success of a file transfer.
- **FileChunk:** Represents chunks of a file for efficient transfer.
- **Empty:** Used as a placeholder when no additional data is needed.

3 System Organization

3.1 Server-Side

The server is responsible for:

- Listening for client requests.

- Handling file transfer operations using the `SendFile` and `ReceiveFile` methods.
- Writing received files to disk.

3.2 Client-Side

The client performs the following tasks:

- Sending files to the server using the `SendFile` method.
- Receiving files from the server in chunks using the `ReceiveFile` method.

4 Implementation Details

4.1 Protocol Buffer Definition

The protobuf file defines the service and message structures:

```
syntax = "proto3";

package filetransfer;

service FileTransfer {
    rpc SendFile(FileRequest) returns (FileResponse);
    rpc ReceiveFile(FileChunk) returns (Empty);
}

message FileRequest {
    string filename = 1;
    bytes content = 2;
}

message FileResponse {
    bool success = 1;
}

message FileChunk {
    bytes content = 1;
}

message Empty {}
```

Listing 1: Protocol Buffer Definition

4.2 Client Implementation

The client implementation (`rpcclient.cpp`) includes:

```
#include <iostream>
#include <fstream>
#include <string>
#include <grpcpp/grpcpp.h>
#include "file_transfer.grpc.pb.h"
```

```
6
7  using grpc::Channel;
8  using grpc::ClientContext;
9  using grpc::Status;
10 using filetransfer::FileTransfer;
11 using filetransfer::FileRequest;
12 using filetransfer::FileResponse;
13 using filetransfer::FileChunk;
14 using filetransfer::Empty;
15
16 class FileTransferClient {
17 public:
18     FileTransferClient(std::shared_ptr<Channel> channel)
19             : stub_(FileTransfer::NewStub(channel)) {}
20
21     bool SendFile(const std::string& filename) {
22         std::ifstream file(filename, std::ios::binary);
23         if (!file.is_open()) {
24             std::cerr << "Error opening file for reading" << std::
    endl;
25             return false;
26         }
27
28         FileRequest request;
29         request.set_filename(filename);
30         std::string content((std::istreambuf_iterator<char>(file)),
     (std::istreambuf_iterator<char>()));
31         request.set_content(content);
32
33         FileResponse response;
34         ClientContext context;
35         Status status = stub_->SendFile(&context, request, &
    response);
36         if (status.ok() && response.success()) {
37             std::cout << "File sent successfully!" << std::endl;
38             return true;
39         } else {
40             std::cerr << "Error sending file: " << status.
    error_message() << std::endl;
41             return false;
42         }
43     }
44
45     void ReceiveFile() {
46         FileTransfer::Stub stub(grpc::CreateChannel("localhost
    :50051", grpc::InsecureChannelCredentials()));
47         Empty request;
48         FileChunk chunk;
49         ClientContext context;
50
51         std::ofstream file("received_file.txt", std::ios::binary);
52         if (!file.is_open()) {
53             std::cerr << "Error opening file for writing" << std::
    endl;
54             return;
55         }
56
```

```
        while (!file.eof()) {
            chunk.set_content(std::string((std::istreambuf_iterator
    <char>(file)), (std::istreambuf_iterator<char>())));
            Status status = stub.ReceiveFile(&context, chunk, &
    request);
            if (!status.ok()) {
                std::cerr << "Error receiving file: " << status.
    error_message() << std::endl;
                return;
            }
        }
        std::cout << "File received successfully!" << std::endl;
    }

private:
    std::unique_ptr<FileTransfer::Stub> stub_;
};

int main(int argc, char** argv) {
    FileTransferClient client(grpc::CreateChannel("localhost:50051"
    , grpc::InsecureChannelCredentials()));
    client.SendFile("sample_file.txt");
    client.ReceiveFile();
    return 0;
}
```

Listing 2: Client Implementation

4.3 Server Implementation

The server implementation (`rpcserver.cpp`) includes:

```
#include <iostream>
#include <memory>
#include <string>
#include <fstream>
#include <grpcpp/grpcpp.h>
#include "file_transfer.grpc.pb.h"

using grpc::Server;
using grpc::ServerBuilder;
using grpc::ServerContext;
using grpc::Status;
using filetransfer::FileTransfer;
using filetransfer::FileRequest;
using filetransfer::FileResponse;
using filetransfer::FileChunk;
using filetransfer::Empty;

class FileTransferServiceImpl final : public FileTransfer::Service
    {

    Status SendFile(ServerContext* context, const FileRequest*
    request, FileResponse* response) override {

        std::ofstream file(request->filename(), std::ios::binary);
        if (!file.is_open()) {
```

```
            std::cerr << "Error opening file for writing" << std::endl;
            return Status::OK;
        }

        file.write(request->content().c_str(), request->content().length());
        file.close();

        response->set_success(true);
        return Status::OK;
    }

    Status ReceiveFile(ServerContext* context, const FileChunk* request, Empty* response) override {

        std::ofstream file("received_file.txt", std::ios::binary | std::ios::app);
        if (!file.is_open()) {
            std::cerr << "Error opening file for writing" << std::endl;
            return Status::OK;
        }

        file.write(request->content().c_str(), request->content().length());
        file.close();
        return Status::OK;
    }
};

void RunServer() {
    std::string server_address("0.0.0.0:50051");
    FileTransferServiceImpl service;

    ServerBuilder builder;
    builder.AddListeningPort(server_address, grpc::InsecureServerCredentials());
    builder.RegisterService(&service);

    std::unique_ptr<Server> server(builder.BuildAndStart());
    std::cout << "Server listening on " << server_address << std::endl;
    server->Wait();
}

int main() {
    RunServer();
    return 0;
}
```

Listing 3: Server Implementation

5 File Transfer Workflow

5.1 Sending a File

1. The client reads the file content.
2. Constructs a `FileRequest` message.
3. Sends the message to the server using `SendFile`.
4. The server writes the file to disk and responds with a success status.

5.2 Receiving a File

1. The client requests a file in chunks.
2. The server sends chunks using `FileChunk` messages.
3. The client reconstructs the file locally.

6 Conclusion

This RPC File Transfer system showcases efficient file handling using gRPC. The modular design ensures robust client-server communication. Future enhancements can include error handling and progress tracking.