Practical Work 2: RPC File Transfer System using gRPC

Tran Vu Cong Minh
ID: 23BI14303

Introduction

This report presents the design and implementation of a Remote Procedure [cite_start]Call (RPC)-based file transfer system using the gRPC framework in C++[cite: 1]. [cite_start]The objective is to provide a simple one-to-one file transfer service [cite: 2] [cite_start]where a client can send a file to a server through remote procedure calls[cite: 2]. [cite_start]This approach avoids directly using low-level socket operations[cite: 2].

System Goal

The main goals of the system are:

- to implement a single client and a single server[cite: 3], [cite_start]
- to transfer file data from the client to the server[cite: 3], [cite_start]
- to use RPC as the primary communication mechanism[cite: 3], [cite_start]
- to define a clear service interface via a `.proto` file[cite: 3].

The system uses gRPC over TCP/IP and Protocol Buffers for message [cite_start]serialization[cite: 4].

RPC Interface Specification

The interface between the client and the server is defined in the file [cite_start]`file_transfer.proto`[cite: 5]. This file specifies the RPC service [cite_start]`FileTransfer` and the request/response message formats[cite: 6].

Protocol Definition

```
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 {}
```

The `SendFile` RPC is used for sending a complete file from the [cite_start]client to the server[cite: 10]. The `ReceiveFile` RPC is defined to support [cite_start]chunk-based transfer and future extensions[cite: 11].

System Architecture

The system architecture consists of:

- a gRPC client that reads a file and calls `SendFile`[cite: 12],
- a gRPC server that implements `SendFile` and [cite_start]`ReceiveFile`[cite: 12], [cite_start]
- a `.proto` file shared by both sides[cite: 12],
- code generated by the Protocol Buffers compiler and gRPC [cite_start]plugins[cite: 12].

Conceptually, the architecture can be summarized as:

The client code invokes methods on the stub as if it were calling local [cite_start]functions[cite: 13]. gRPC transparently serializes the messages, sends them over [cite_start]the network, and invokes the appropriate server-side implementation[cite: 14].

Server Implementation

[cite_start]The server implementation is contained in `server.cc`[cite: 15]. It registers an instance of `FileTransferServiceImpl` with a gRPC [cite_start]server and waits for incoming RPC calls[cite: 15].

```cpp
#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, Fi

        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, E

        std::ofstream file("received_file.txt", std::ios::binary | std::io
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::InsecureServerCredentia
    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;
}
```

The method `SendFile` creates a file using the filename provided [cite_start]in the `FileRequest` and writes the binary content into it[cite: 26]. The `ReceiveFile` method appends additional chunks to a file named [cite_start]`received_file.txt`[cite: 27], which allows the system to be extended with [cite_start]chunk-based transfer[cite: 27].

Client Implementation

[cite_start]The client implementation is contained in `client.cc`[cite: 28]. It [cite_start]creates a stub to the `FileTransfer` service [cite: 28][cite_start], reads a local file[cite: 28], [cite_start]and sends it to the server using the `SendFile` RPC[cite: 28].

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

using grpc::Channel;
using grpc::ClientContext;
using grpc::Status;
using filetransfer::FileTransfer;
using filetransfer::FileRequest;
using filetransfer::FileResponse;
using filetransfer::FileChunk;
using filetransfer::Empty;

class FileTransferClient {
public:
    FileTransferClient(std::shared_ptr<Channel> channel)
            : stub_(FileTransfer::NewStub(channel)) {}

    bool SendFile(const std::string& filename) {
        std::ifstream file(filename, std::ios::binary);
if (!file.is_open()) {
            std::cerr << "Error opening file for reading" << std::endl;
return false;
        }

        FileRequest request;
        request.set_filename(filename);
        std::string content((std::istreambuf_iterator<char>(file)), (std::
        request.set_content(content);

        FileResponse response;
ClientContext context;
        Status status = stub_->SendFile(&context, request, &response);
        if (status.ok() && response.success()) {
            std::cout << "File sent successfully!"
<< std::endl;
            return true;
        } else {
            std::cerr << "Error sending file: " << status.error_message()
return false;
        }
    }

    void ReceiveFile() {
        FileTransfer::Stub stub(grpc::CreateChannel("localhost:50051", grp
Empty request;
        FileChunk chunk;
        ClientContext context;

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

        while (!file.eof()) {
            chunk.set_content(std::string((std::istreambuf_iterator<char>(
Status status = stub.ReceiveFile(&context, chunk, &request);
            if (!status.ok()) {
```

```
            if (!status.ok()) {
                std::cerr << "Error receiving file: " << status.error_mess
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:
client.SendFile("sample_file.txt");
    client.ReceiveFile();
    return 0;
}
```

The method `SendFile` reads the entire file into a [cite_start]`std::string` [cite: 41] [cite_start], populates a `FileRequest` message[cite: 41], and [cite_start]invokes the `SendFile` RPC[cite: 41]. The `ReceiveFile` method demonstrates how the client can call the `ReceiveFile` RPC on the [cite_start]server to receive data[cite: 42], which can be extended into a full download [cite_start]feature[cite: 42].

Build and Execution

To build and run the system, the following generic steps can be used:

1. Generate gRPC and Protocol Buffers code from [cite_start]`file_transfer.proto` using `protoc`[cite: 43].
2. Compile `server.cc` and `client.cc` and link [cite_start]them with gRPC and Protocol Buffers libraries[cite: 44]. [cite_start]
3. Start the server executable[cite: 44]. [cite_start]
4. Run the client executable to send a sample file[cite: 44].

Once the client has executed `SendFile`, the transferred file [cite_start]appears in the server's working directory with the same file name[cite: 46].

Conclusion

This practical work demonstrates a complete RPC-based file transfer [cite_start]system using gRPC in C++[cite: 47]. The design is centered around a clear service [cite_start]definition in `file_transfer.proto`[cite: 48], and the client and server implementations follow this interface to exchange file data reliably [cite_start]over the network[cite: 48]. By using gRPC, the system avoids direct socket programming and relies on high-level remote procedure calls, which simplifies development and provides a structured way to extend the system with additional features [cite_start]such as streaming, authentication, or integrity checks[cite: 49].