# Practical Work 2: RPC File Transfer System using gRPC

Nguyen Ngoc Quang
ID: 23BI14376

## 1 Introduction

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

## 2 System Goal

The main goals of the system are:

- to implement a single client and a single server,
- to transfer file data from the client to the server,
- to use RPC as the primary communication mechanism,
- to define a clear service interface via a .proto file.

The system uses gRPC over TCP/IP and Protocol Buffers for message serialization.

## 3 RPC Interface Specification

The interface between the client and the server is defined in the file file transfer.proto. This file specifies the RPC service FileTransfer and the request/response message formats.

### Protocol Definition

Listing 1: RPC Service Definition: file$_t$*ransfer.proto*

```
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 client to the server. The ReceiveFile RPC is defined to support chunk-based transfer and future extensions.

# 4 System Architecture

The system architecture consists of:

- a gRPC client that reads a file and calls SendFile,
- a gRPC server that implements SendFile and ReceiveFile,
- a .proto file shared by both sides,
- code generated by the Protocol Buffers compiler and gRPC plugins.

Conceptually, the architecture can be summarized as:

Client Application → gRPC Stub → HTTP/2 over TCP → gRPC Server → File Storage

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

# 5 Server Implementation

The server implementation is contained in server.cc. It registers an instance of FileTransferServiceImpl wit
a gRPC server and waits for incoming RPC calls. :contentReference[oaicite:0]index=0

Listing 2: Server implementation: server.cc

```
#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;
}
```

The method SendFile creates a file using the filename provided in the FileRequest and writes the binary content into it. The ReceiveFile method appends additional chunks to a file named received file.txt, which allows the system to be extended with chunk-based transfer.

## 6 Client Implementation

The client implementation is contained in client.cc. It creates a stub to the FileTransfer service, reads a local file, and sends it to the server using the SendFile RPC. :contentReference[oaicite:1]index=1

Listing 3: Client implementation: client.cc

```
#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::
                istreambuf_iterator<char>()));
        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()
                        << std::endl; return false;
        }
}

void ReceiveFile() {
        FileTransfer::Stub stub(grpc::CreateChannel("localhost:50051", grpc::InsecureChannelCredentials()));
        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>(      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;
}
```

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

## 7 Build and Execution

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

1. Generate gRPC and Protocol Buffers code from file transfer.proto using protoc.

2. Compile server.cc and client.cc and link them with gRPC and Protocol Buffers libraries.

3. Start the server executable.

4. Run the client executable to send a sample file.

Once the client has executed SendFile, the transferred file appears in the server's working directory with the same file name.

## 8 Conclusion

This practical work demonstrates a complete RPC-based file transfer system using gRPC in C++. The design is centered around a clear service definition in file transfer.proto, and the client and server implementations follow this interface to exchange file data reliably over the network.

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 such as streaming, authentication, or integrity checks.