DiSy

Directory and file synchronization using Protobuf

Konstantin Lampalzer

Contents

1 License

2 Task

Goal of the project is creating a Server-Client system used to synchronize directories and files between multiple network processes. In order to compare the files timestamps and hash values are used

2.1 Technologies

`DiSy` is using features introduced in the C++17 standard and therefor relies on a C++ compiler adhering to this specification. `g++` (Ubuntu 7.3.0-27ubuntu1 18.04) is used to compile `DiSy` on Ubuntu 18.04.2 LTS "bionic"..

Use-Case	Technology
Networking	asio
Networking	GRPC
Logging	spdlog
Command line argument parsing	CLI11
Configuration files	json
SHA-512 file hashing	OpenSSL
Data serialization	Protobuf

Table 1: Used libraries/technologies.

All libraries beside OpenSSL were mandatory as by project definition. OpenSSL was chosen for file hashing, because it supports SHA-512, which is deemed to be securely enough for file hashing. Additionally, it is often preinstalled on most common Linux distributions.

2.2 Assumptions

1. Only Unix-like operating systems are supported. It might work on other systems, but the software was only tested on the most recent version of Ubuntu.
2. The underlying file-system of client and server is the same →
   (a) Maximum path lengths are equal
   (b) Case sensitive / insensitive paths can not be mixed
3. All clients have working clocks that are approximately synchronized
4. Synchronization is done till a client stops the program→
   (a) Deletion of files is not included
   (b) File rights are not synchronized
   (c) Live editing on multiple clients is not supported
5. Conflicts are solved server-side
6. Forced synchronization termination can result in inconsistent state

3 Implementation

3.1 Project structure

DiSy

- build/:
- include/:
  - client/:.................All header files relevant to only the client
  - server/:................All header files relevant to only the server
  - shared/:.......All header files relevant to the server and the client
    - asioNetworking.hpp:..Header-only. Handles all the networking done by asio
    - crawler.hpp:.Header-only. Responsible for crawling a directory and creating a DirTree object.
    - hashing.hpp:...Header-only. Takes a file and retuns the hash of this file.
    - reader.hpp:.Header-only. Responsible for reading files from the filesystem and turning them into a byte-array.
    - shared.hpp:............Header-only. General shared functions.
    - writer.hpp:..Header-only. Writes files from a byte-array to the filesystem.
- src/:
  - client/:
    - client.cpp:................All general client-side functionality.
    - downloader.cpp:.Downloads files and directories from the server
    - main.cpp:.......................Main entrypoint for the client
    - uploader.cpp:........Uploads files and directories to the server
  - protos/: .4 DiSy.proto:..........Proto message definition file
  - server/: .4 main.cpp:..........Main entrypoint for the server
    - server.cpp:................All general server-side functionality.
- meson.build:..
- meson_options.txt:..

3.2 Recursive directory traversal

The `filesystem` library was introduced into the C++ language standard in 2016. but support is still relatively poor. Still, it was decided to sacrifice support of systems without the experimental filesystem library, as it provides a hassle-free way to iterate trough all files recursively. Additionally, support will be expanded with the next C++ version, C++20.

3.3 Protocol Buffers

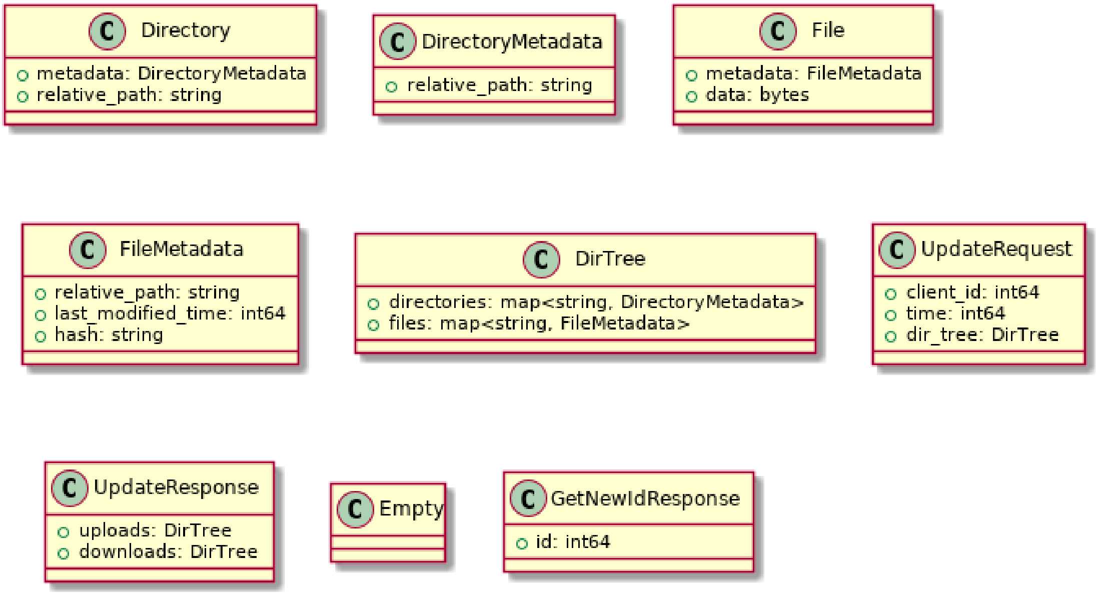

Figure 1: All defined Protobuf messages.

Communication between server and client is exclusively handled with Protobuf messages. An example of their usage in the actual protocol can be found on page 8.

3.4 Command line interface

```
DiSy server
Usage: ./server [OPTIONS]

Options:
    -h,--help                   Print this help message and exit
    -g,--gport INT              grpc server port
    -a,--aport INT              asio server port
    -d,--dir DIR REQUIRED       Directory to synchronize
    --debug                     debug messages

--------------------------------------------------------------

DiSy client
Usage: ./client [OPTIONS]

Options:
    -h,--help                   Print this help message and exit
    -d,--dir DIR                Directory to synchronize
    -c,--config FILE            Config file
    -g,--gaddres TEXT           grpc server address
    -a,--aaddres TEXT           asio server address
    -p,--port INT               asio server port
    -debug                      debug messages
```

DiSy server and DiSy client provide very similar command line interfaces. The address is not configurable for DiSy server because its IPv4 socket is automatically bound to 0.0.0.0.

Usage:
Start the DiSy server. Then start at least two clients. They will now automatically synchronize the supplied parameters.

A configuration file in the JSON file format can also be supplied instead of parameters on the client:

```json
{
        "asioAddress":"0.0.0.0",
        "asioPort":8081,
        "debug":false,
        "grpcAddress":"0.0.0.0:8080",
        "path":"default"
}
```

Figure 2: Example configuration file.

4 Networking

A slim header is perpended to Protobuf messages to determine their length and differentiate between different message types. Fixed size unsigned integers are used because their length should be identical between different operating systems and processor architectures.

asio's `mutable_buffers` are used to encode all header fields before transmission.

```
+---+------------------------+
| 1 |            8           |
+---+------------------------+
```

Figure 3: Header perpended to Protobuf messages.

To cut down on boilerplate code it was decided to utilize lookup tables to dynamically determine which type of Protobuf message is being sent or received.

```
enum class MessageType
{
        FileMetadata = 1,
        File = 2
};

const std::unordered_map<std::type_index, MessageType> typeMapping{
        {typeid(DiSy::FileMetadata), MessageType::FileMetadata},
        {typeid(DiSy::File), MessageType::File}
};
```

Figure 4: Shortened lookup table definition.

This decision rules out possible optimizations by the compiler, as message type lookup has to be performed at runtime. While this compromise does result in degraded performance, the difference is insignificant as most processor time will be spent waiting for file I/O operations anyways.

```
inline int sendProto(tcp::socket &socket, google::protobuf::Message &message)
{
        u_int8_t messageType{toUnderlying(typeMapping.at(typeid(message)))};
        u_int64_t messageSize{message.ByteSizeLong()};

        asio::write(socket, buffer(&messageType, sizeof(messageType)));
        asio::write(socket, buffer(&messageSize, sizeof(messageSize)));

        streambuf streamBuffer;
        ostream outputStream(&streamBuffer);
        message.SerializeToOstream(&outputStream);
        asio::write(socket, streamBuffer);
        return SEND_OK;
}
```

Figure 5: Send function. All Protobuf messages are subclasses of Message, which is provided by Protobuf.

Additionally, there are two functions provided to receive the next message type and the message itself. This enables me to easily transfer any message over the network using asio. Only files are transferred over asio, everything else is communicated via gRPC.

4.1 Protocol phases

4.1.1 First phase

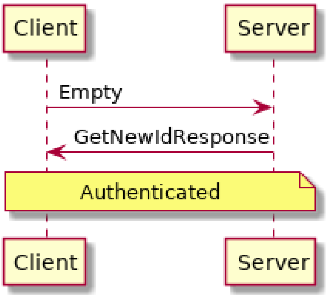

Figure 6: First phase: Client requesting an id from the server.

After the client establishes a connection to the server it requests an ID from the server. The server then creates an unique ID for the client and sends it back to the client. The client then traverses all files and directories in the specified folder to compose a `DirTree` message, which is used by the server to compute which files/directories need to be created/sent/requested.

4.1.2 Second phase

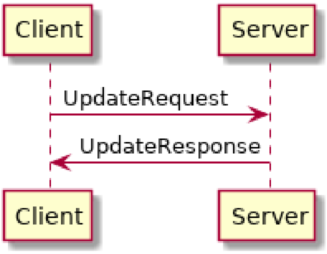

Figure 7: Second phase: Transmission of directories and files as well as requests for files that are not present on the server.

After the client traversed all his local files, he creates an `UpdateRequest` message containing the current timestamp and the DirTree. The message then computes the difference between the received DirTree and the server DirTree and afterwards sends an `UpdateResponse` with all the files and directories that need to be uploaded/downloaded by the client.

```
void Client::sendUpdate()
{
        grpc::ClientContext clientContext;
        DiSy::UpdateRequest updateRequest;
        updateRequest.set_client_id(clientId);
        updateRequest.set_allocated_dir_tree(crawler::crawlDirectory(path));
        updateRequest.set_time(shared::getCurrentTime());

        DiSy::UpdateResponse updateResponse;
        grpc::Status status{stub->Update(&clientContext, updateRequest, &updateResponse)};

        uploadDirectories(updateResponse.uploads().directories());
        downloadDirectories(updateResponse.downloads().directories());

        uploadFiles(updateResponse.uploads().files());
        downloadFiles(updateResponse.downloads().files());
}
```

Figure 8: Processing of `UpdateRequest` messages.

Directories that are present on the server but missing on the client are handled next. `DirectoryMetadata` messages are sent to the server next. For every directory request one `DirectoryMetadata` message is sent. The server responds with a Directory which is then created on the client. These steps are repeated for the FileRequests and then all the requested folders and files from the server are uploaded.

Finally, the update phase is completed and the client waits for a bit, till the update is repeated.

4.1.3 Third phase

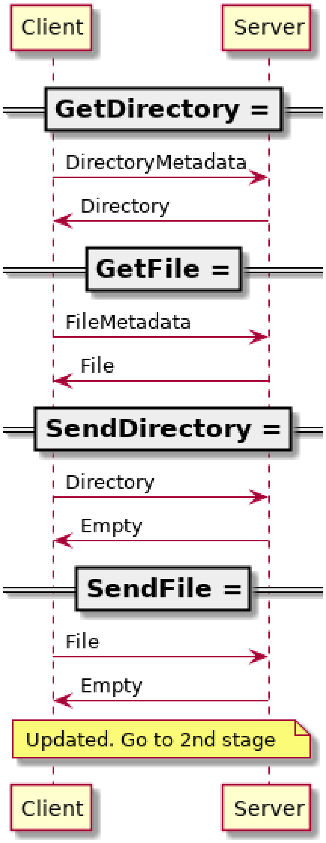

Figure 9: Third phase: Files and directories are exchanged between the server and the client.