dir_sync

Protobuf-powered file synchronization

Philip Trauner

Contents

1 License

Boost Software License - Version 1.0 - August 17th, 2003

Permission is hereby granted, free of charge, to any person or organization obtaining a copy of the software and accompanying documentation covered by this license (the "Software") to use, reproduce, display, distribute, execute, and transmit the Software, and to prepare derivative works of the Software, and to permit third-parties to whom the Software is furnished to do so, all subject to the following:

The copyright notices in the Software and this entire statement, including the above license grant, this restriction and the following disclaimer, must be included in all copies of the Software, in whole or in part, and all derivative works of the Software, unless such copies or derivative works are solely in the form of machine-executable object code generated by a source language processor.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, TITLE AND NON-INFRINGEMENT. IN NO EVENT SHALL THE COPYRIGHT HOLDERS OR ANYONE DISTRIBUTING THE SOFTWARE BE LIABLE FOR ANY DAMAGES OR OTHER LIABILITY, WHETHER IN CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

2 Task

Implementation of a networked command-line file synchronization utility. Timestamps and hashes should be utilized for comparison operations.

2.1 Technologies

`dir_sync` is using features introduced in the C++14 standard and therefor relies on a C++ compiler adhering to this specification. `g++` (version 6.3.0-18) is used to compile `dir_sync` on Debian 9.3 "Stretch" and `clang` (version 900.0.39.2) is used on macOS 10.13.2.

Use-Case	Technology
Networking	asio
Logging	spdlog
String formatting	format
Command line argument parsing	clipp
Configuration files	json
SHA-512 file hashing	OpenSSL
Data exchange format	Protobuf

Table 1: Used libraries/technologies.

All libraries beside OpenSSL were mandatory as by project definition. OpenSSL was chosen for file hashing because no header-only library that supports SHA-512, which was deemed a requirement because SHA-256 is not considered secure anymore, could be located. It is also often pre-installed on common Linux distributions.

2.2 Assumptions

1. Only Unix-like operating system are supported
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. No control files exist in regular directories
4. Both peers have working clocks that are approximately synchronized
5. Synchronization is not continuous →
   (a) Deletion of files is not handled
6. Conflicts are solved server-side
7. Forced synchronisation termination can result in inconsistent state

3 Implementation

3.1 Code sharing

Functions and constants used by server as well as client are defined in a module aptly named `shared` to prevent needless code duplication.

3.2 Recursive directory traversal

The `filesystem` library was introduced into the C++ language standard in 2016. Support is still relatively poor, neither the LLVM toolchain bundled with macOS nor the GNU toolchain bundled with Debian 9.3 include a current non-experimental version of the library. It was decided to sacrifice support of non Unix-like operating systems and use the `nftw(...)` (`<ftw.h>`) function instead. It expects a pointer to a function that is called for every item in the file tree it traverses, therefor persistent data has to stored in an overlying scope, which, in this case, is the root scope. To prevent possible data races the `nftw()` call itself is wrapped in another function that locks a global `Mutex` on execution.

3.3 Protocol Buffers

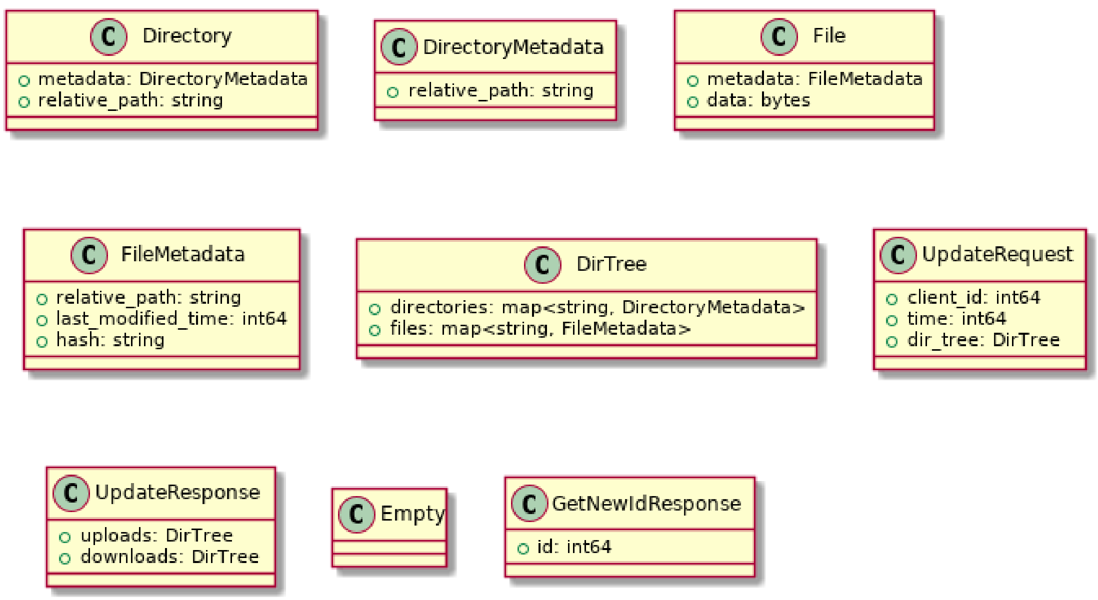

Figure 1: All defined Protobuf messages.

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 7.

3.4 Command line interface

```
SYNOPSIS
        ./dir_sync_server <directory> [--strict]
             [--verbose] [-p <port>] [-c <config>]
        ./dir_sync_client <directory> <address>
             [--verbose] [-p <port>] [-c <config>]

OPTIONS
        --strict    do not continue if client is deemed insane
        --verbose   log additional debug info
        -p, --port  provide alternative port

        -c, --config
                    override command line parameters with config
```

`dir_sync_server` and `dir_sync_client` provide very similar command line interfaces. The address is not configurable for `dir_sync_server` because its IPv4 socket is automatically bound to 0.0.0.0.

A configuration file in the `JSON` file format can also be supplied to substitute optional parameters. All command line options are available as keys.

```json
{
        "strict": true,
        "verbose": true,
        "port": 1337,
}
```

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.

```cpp
enum class MessageType {
        FileTree = 1,
        FileRequest = 2,
        // ...
};

const unordered_map<type_index, MessageType> proto_type_mapping {
        {typeid(FileTree), MessageType::FileTree},
        {typeid(FileRequest), MessageType::FileRequest},
        // ...
};
```

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 to complete.

```cpp
void send_proto(tcp::socket& sock, Message& message) {
        u_int8_t message_type{to_underlying(
                proto_type_mapping.at(typeid(message)))};
        u_int64_t message_size{message.ByteSizeLong()};

        write(sock, buffer(&message_type, U_INT_8_SIZE));
        write(sock, buffer(&message_size, U_INT_64_SIZE));

        streambuf buf;
        ostream os(&buf);
        message.SerializeToOstream(&os);

        write(sock, buf);
}
```

Figure 5: Send function (error handling stripped out to reduce length). All Protobuf messages are subclasses of Message.

This design decision implies that the receiving end always knows which message type will be received next, which is not possible in scenarios where the same message type is sent multiple times in a row (`FileBlock`, `FileRequest`, ...). A special message type called `ProtocolSeparator` was introduced to solve this issue. It has no content and is sent to delimit protocol stages.

An example of its usage in the actual protocol can be found on page 7.

The `recv_proto` function returns a special status code if a `ProtocolSeparator` has been received instead of the expected message type. It is the responsibility of the application logic to validate the returned status code for each received message.

4.1 Protocol phases

4.1.1 First phase

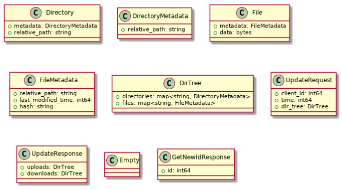

Figure 6: First phase: Transmission of `SanityCheck` and `FileTree`.

After the client establishes a connection to the server it creates a `SanityCheck` message and fills it with its current time which is then forwarded to the server. The client is deemed sane if its clock offset relative to the server time is less than five seconds. If the client is deemed insane a warning message is displayed by the server.

The client then traverses all files and directories in the specified folder to compose a `FileTree` message, which is used by the server to compute which files/directories need the be created/sent/requested.

4.1.2 Second phase

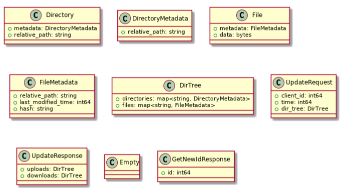

Figure 7: Second phase: Transmission of directories and files as well as requests for files that are not present on the server.

After the server has created all folders that are present in the received `FileTree` but are missing locally it emits `DirectoryMetadata` messages filled with directories that have to be created by the client. After all necessary `DirectoryMetadata` messages have been transmitted a `ProtocolSeparator` is sent.

```
for (const string& dir_path : get<0>(file_tree_dir_diff_)) {
        send_directory(sock, dir_path);
}

send_protocol_separator(sock);
```

Figure 8: Transmission of `DirectoryMetadata` messages.

Files that are present on the server but missing on the client are handled next. A `MinimalFileMetadata` message containing the path of the file that is about to be transfered as well as its last modification time-stamp is sent before 65 536 B file parts are transmitted in the form of `FileResponse` messages. A `ProtocolSeparator` marks the end of a file. If two consecutive `ProtocolSeparator`s are received the client assumes that all missing files have been received.

Lastly, the server requests all missing files from the client.

4.1.3 Third phase

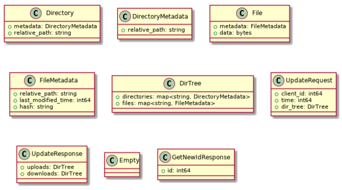

Figure 9: Third phase: Files that have been requested by the server are sent by the client.

After all `FileRequest` messages have been received the client sends the requested files in the same manner as the server does in phase two.

5 Conclusion

Just use `rsync -avP --checksum <source> <destination>`.