Protocol

Server/client are connected via TCP and the protocol is inspired by websockets but simplified to only fit our use case. For a production environment, proper websockets should be considered to allow usage of well-tested libraries for timeouts / retries and similar functionality.

The interaction works as follows:

- The client sends an API key which the server validates
- The server adds the validated client to the clients list
- client and server send frames of data to each other

A frame is defined by:

- 1 byte opcode
- 4 bytes payload length
- x bytes payload

Each client goes through an init phase, where they exchange their state of the folder with the server and receive back which files they need to send.

The server automatically adds all files that are missing for this client (be it because they are already synced with the server or because a new client connects) to a write queue, which then writes them to the client socket.

A typical interaction after API key validation, using all available opcodes, could look like this:

- Client: INIT (containing a description of all files the client has)
- Server: INIT (containing the files that the client needs to send)
- Client: SYNC_NEW_FILE_PATH
- Client: SYNC_NEW_FILE_PART
- Client: SYNC_NEW_FILE_PART
- Client: SYNC_NEW_FILE_END
- Server: SYNC_NEW_FILE_PATH (Server is sending file that is missing on client)
- Server: SYNC_NEW_FILE_PART
- Server: SYNC_NEW_FILE_PART
- Server: SYNC_NEW_FILE_END
- Client: SYNC_CLOSE

Except for the INIT, sending/recieving frames to/from the client happens in parallel.

Architecture

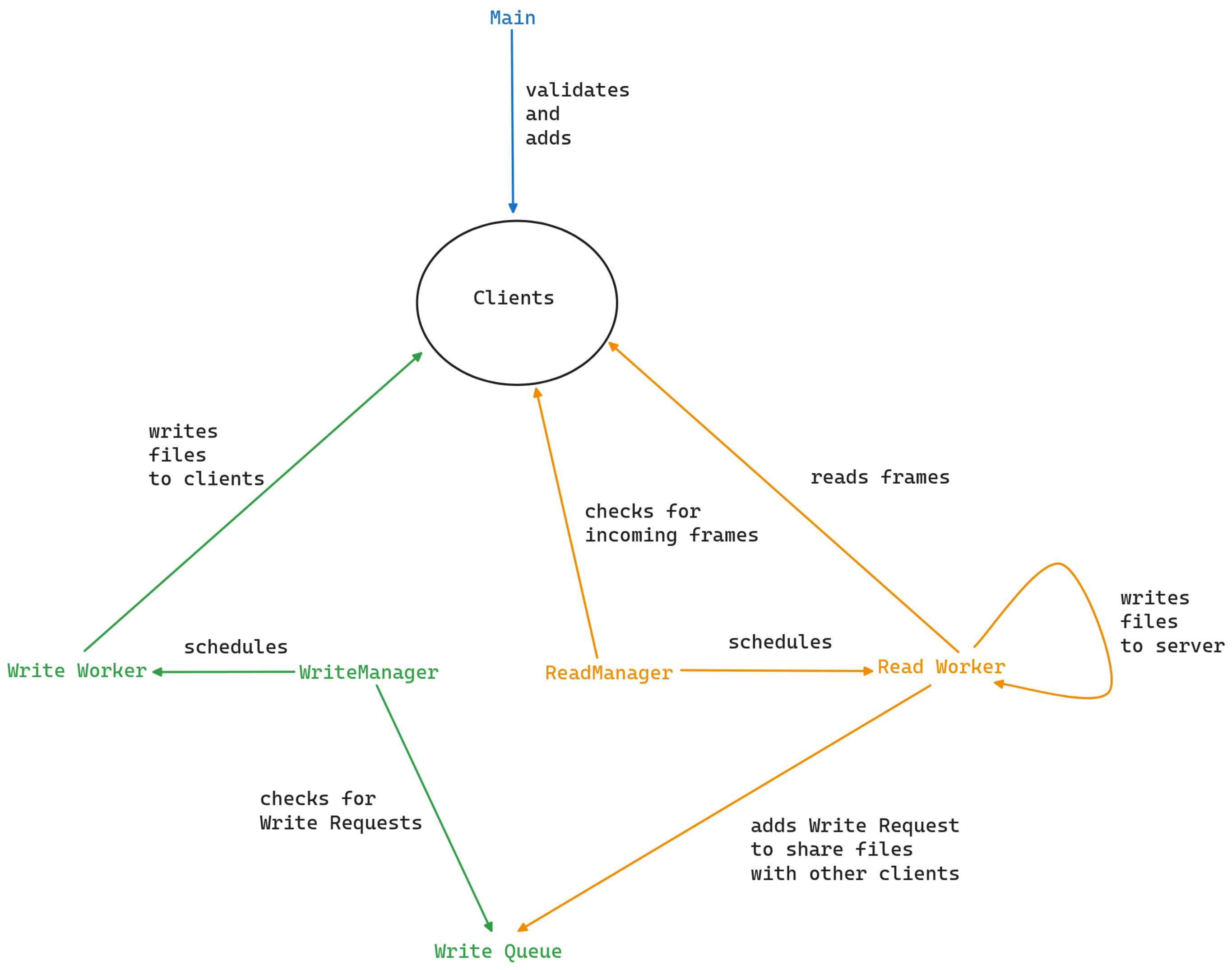
Server

The Java server contains a single source of truth for which files are available and their contents. The server gets its data by reading (from client socket) and writing (to client socket).

Excluding the INIT (which is a special case, see above), there are two loops ongoing in two threads:

- Reading Loop: receives client frames, ensures we are not reading in parallel from a single client, writes new files to the server folder, adds files that need to be send to clients to the write queue. The check for new frames is handled in one thread, while the actual reading and interpreting is handed over to a worker pool.
- Writing Loop: writes frames to clients, ensures we are not writing in parallel to a single client, takes write requests from the queue and hands them over to a worker pool.

The overall architecture of the server can be visualized like this:



The main building blocks (Main, ReadManger, WriteManger) correspond to the central classes of the application. An additional SyncManager class is used for easier clean up.

Client

The TypeScript and Go clients follow a mostly similar approach. The have a starting portion of the code which connects to the server, sends the API key and sets everything up. Then the rest of the work is delegated to a FrameHandler class, which listens for incoming frames from the server and reacts to them.

The TypeScript client is a little more complicated due to the callback-nature of the standard lib. Usually I would prefer to use a promise-based library, that builds on top of the standard sockets and writestreams.

The TypeScript client has the following classes:

- FrameHandler: handles server interactions, keeps folder state and delegates to the other classes.
- FrameParser: keeps incoming chunks of data until a frame can be parsed.
- FolderWriteHandler: interface to write frames to the file system with a write stream.

The reason to use classes here and not just plain functions, is that the incoming byte stream from the server is arriving in an async fashion, which necessitates collecting bytes before being able to react to frames. This makes classes very convenient to keep the state and allow for easy construction of the necessary callbacks.

The Go client only has one central FrameHandler class because it offers a blocking approach for reading from the socket, which greatly simplifies the state handling - making the other classes redundant. This is certainly not the optimal way to solve the reading/writing and would profit from a more concurrent approach. Due to the time-constraint I opted for this simple but working approach instead of not delivering a second client at all.

Tests

There is one collection of unit tests that can be found in the ReadManagerTest class of the Java server. This is not a complete test suite but illustrates my approach to testing - separating the IO from the core logic and then verifying both with different levels of granularity.

To be able to verify the interactions between the server and clients, I created an integration testing setup. The integration test scripts start the server and various clients and checks if the files are synced properly. This guarantees a base level of functionality. It could easily be expanded to test further scenarios and also verify performance characteristics and find the right chunk size for the data transfer.

Possible further improvements

File Watchers

The primary missing functionality is the continued exchange of files between client and server. Currently, files are only exchanged on the first connection. This means we receive the initial state of all clients but don't receive updates to the folder. This would need to be simulated by restarting the clients.

For the file watchers, I would strongly prefer using established libraries. For example, the NodeJS core lib file watching capability is famously unreliable and does not even support recursive watching on Linux, which is why a standard replacement has been established in the library "chokidar".

Edge cases and error handling

By having the server as the single source of truth, a certain reliability can be guaranteed. For example, the server decides which of the client files need to be requested. However, there is certainly a lot of open potential for improved handling of exceptions and errors - e.g. with respect to allowing clients to re-send frames instead of disconnecting them. Additionally, more validation on the server side is necessary - e.g. currently it is assumed that all paths have the correct format.

Performance

In practice, there are at most 6 threads active on the server that do actual work (from accepting clients to reading/writing). The writing to clients is currently bottle-necked on the fact that each incoming file needs to be send to all clients. This fan-out blocks most clients (all except for the one that send the file in the first place) from being written to. The current implementation assumes that server disk IO should be minimized (adjustable via the chunk size env var). However, with a production server this might not be the case. The performance differences between this "per-file" approach and a "per-client" approach, where we would write different files to different clients in parallel, could be evaluated.

One more bottleneck with big folders is the general model or approach to keeping the folder state: a simple list of paths. This contains redundant information and could be optimized for big amounts of files with a tree-inspired approach that keeps some kind of representation of the actual files and folders in memory.

Coding standards and CI/CD

Using the build scripts for CI/CD should be relatively easy. Additional static analysis could be applied.

Security

The API Key approach is of course very naive and should at least be supplemented with some kind of private/public key verification.