Mermade – Zama Merkle Tree Client/Server Challenge

Challenge

Imagine a client has a large set of potentially small files {F0, F1, …, Fn} and wants to upload them to a server and then delete its local copies. The client wants, however, to later download an arbitrary file from the server and be convinced that the file is correct and is not corrupted in any way (in transport, tampered with by the server, etc.).

You should implement the client, the server and a Merkle tree to support the above (we expect you to implement the Merkle tree rather than use a library, but you are free to use a library for the underlying hash functions).

The client must compute a single Merkle tree root hash and keep it on its disk after uploading the files to the server and deleting its local copies. The client can request the i-th file Fi and a Merkle proof Pi for it from the server. The client uses the proof and compares the resulting root hash with the one it persisted before deleting the files - if they match, file is correct.

You can use any programming language you want (we use Rust internally). We would like to see a solution with networking that can be deployed across multiple machines, and as close to production-ready as you have time for. Please describe the short-coming your solution have in a report, and how you would improve on them given more time.

We expect you to send us within 7 days:

a demo of your app that we can try (ideally using eg Docker Compose) the code of the app a report (max 2-3 pages) explaining your approach, your other ideas, what went well or not, etc..

Assumptions

1. I assume the purpose of the challenge is to demonstrate the ability to code, design and implement a solution that is production ready, but not necessarily to implement a production ready solution. I will therefore make some assumptions and take some shortcuts to save our time.
2. There are many questions that I would ask if this was a real project, but I won't. I'll make more assumptions instead.
3. I assume the "large set of potentially small files" means that eash file is small enough to fit in memory to calculate its hash without streaming, and "large set" means up to several millions of files. My implementation stores the whole Merkle tree in memory, requiring ~64 bytes per file. So, for 1

million files it will require ~64MB of memory. I assume this is acceptable on a modern client or server.
4. I assume the client and the server are on the same network, so I don't need to implement any authentication or encryption.
5. I assume the server has enough disk space to store all files.
6. I assume the server has only one client, although it's easy to extend the solution to support multiple clients, and even shard to multiple servers using a load balancer.

Solution

I've implemented a solution in Rust, using Actix Web framework for the server and Reqwest for the client.

In prod-level solution I would split the project into 3 separate sub-projects: a shared library, a client and a server. Here, I'll keep everything in one application that can be run as a client or a server. My approach simplifies code review, which is the purpose of the challenge.

The client is a CLI tool.

```
Usage: mermade <command> [args]
Commands:
  server <port> -- will start the server on the given port
  upload <server url> <files_dir> -- will upload all files in the <files_dir> directory to t
          output the merkle root to STDOUT and delete the files.
          The Merkle Root is written to STDOUT in HEX format.
          Example: mermade upload http://localhost:8080 files > merkle_root.txt

  download <server url> <index> -- will download the file with the given index from the serv
          verify its merkle proof and output the file to stdout.
          The Merkle Root is read from STDIN in HEX format.
          If the merkle proof is invalid, the program will exit with an error code.
          Example: mermade download http://localhost:8080 0 > file.txt < merkle_root.txt
```

To start the server on port 8080, run:

```
mermade server 8080
```

The server exposes 3 REST API endpoints:

```
POST /upload -- accepts a file upload
GET /files/{index} -- returns a file by its index
GET /proofs/{index} -- returns a Merkle proof for a file by its index
```

The server stores all files in a directory named "files" in its current working directory.

On client's first GET request after an upload, the server computes Merkle proofs for each file and stores them in *proof* files in "proofs" directory.

Each proof file contains a Merkle proof for a file with the same index as the proof file, in binary format.

This is a very simple and efficient solution. The Merkle tree and proofs are computed only once, and proofs are essentially cached. Serving static files is very efficient.

Then, on client GET request, the server simply `sendfile` the file and the proof file to the client.

Merke Tree

I use SHA256 as a hash function. It's fast and secure enough for this purpose.

I compute and store the full Merkle tree in memory in the following form on both client and server. It's a vector of levels of the tree, where each level is a vector of hashes of the nodes on that level. So, for 3 files with hashes "aa", "bb", "cc" the tree will look like this:

```
[
    ["aa", "bb", "cc", "cc"],
    ["dd", "ee"],
    ["ff"]
]
```

where "ff" is the Merkle Root.

This is not the most efficient way to store the tree, but it's simple, easy to implement, and it works. From this implementation it's trivial to derive both Merkle root and proofs.

This implementation requires ~2*32 bytes per file, which is not too bad.

If needed I can implement a "rolling" Merkle root computation, requiring ~2*log2(N) memory, where N is the number of files.

There is a property-based test that verifies that the Merkle tree is correct. It generates random hashes and verifies that for every Merkle proof the computed Merkle root is the same as computed from the tree.

Other

In prod-level solution I would add logging, metrics, more unit and integration tests, better error handling, configuration, multiple clients, authentication and authorization, same files sharing, merkle proofs recalculation on file changes, backpressure, rate limiting, etc.

How to build

I use Nix Flakes to setup my dev environment. You can use it too, or you can install Rust and Cargo manually.

Run `nix develop` to enter the dev environment.

Run `cargo build --release` to build the project.

How to run

Run `cargo run -- server 8080` to start the server on port 8080.

Run `cargo run -- upload http://localhost:8080 files > merkle_root` to upload all files from the “files” directory to the server and store the Merkle Root in the “merkle_root” file.

Run `cargo run -- download http://localhost:8080 0 > file0 < merkle_root` to download the file with index 0 from the server.