Internet Programming Project

Juliusz Chroboczek

23 November 2025
(Updated on 8 December 2025)

1. Introduction

The goal of this project is to implement a distributed read-only file system: every peer exports a filesystem tree that is made available to all other peers. The tree exported by a peer may change at any time, but a peer cannot modify the files exported by a different peer.

The protocol is a hybrid protocol:

- a central REST server serves as a rendez-vous point and as a channel to distribute cryptographic keys;
- data transfer happens directly between peers, over UDP.

Every peer is identified by a name, which is an arbtrary string. Peer names are unique: the server rejects duplicate registrations.

The protocol uses cryptographic techniques in three places:

- communication with the central server happens over HTTP protected by TLS (HTTPS);
- data stored on peers are represented as a Merkle tree;
- messages exchanged between peers are signed with cryptographic signatures.

2. Informal description of the protocol

Peer discovery A peer discovers other peers by contacting the server over a REST-like API. The server maintains one or more socket addresses for every peer, as well as a cryptographic public key.

Registration with the server Registration with the server happens in two steps: first, the client sends its cryptographic signature to the server using a POST request over the HTTP API. It then registers each of its IP addresses by sending a *Hello* request to the server.

After the client sends a *Hello* request to the server, the server will verify that the client is able to receive requests by sending a *Hello* request to the client. If the client doesn't reply to the *Hello* request with a properly signed message, its address will not be published by the server.

Handshake In order to communicate, two peers exchange *Hello* and *HelloReply* messages. These must be protected by cryptographic signatures (see Section 4.3).

Data transfer Every peer maintains a content-indexed database of pieces of data: values are arbitrary pieces of data, while keys are the SHA-256 hashes of the data. A peer requests pieces of data by sending *DatumRequest* messages.

Since data are protected by end-to-end hashes in the form of a Merkle tree, *Datum* messages do not need to be protected by a cryptographic signature.

3. Description of the client-server protocol

The server implements a REST-like protocol, which is notably used to locate other peers. The server is already implemented, you only need to implement the client side.

Note that the client-server protocol does not include a request for registering IP addresses with the server: addresses are registered over UDP, using a subset of the peer-to-peer protocol.

3.1. Peer list

In order to obtain the list of known peers, a client sends a `GET` request to the URL `/peers/`. The server replies with a 200 reply with a list of peer names, one per line.

3.2. Registration

In order to register with the server, a peer ϕ makes a `PUT` request to the URL `/peers/`ϕ`/key` with its 64-byte public key in the body. In order to prevent nickname hijacking, the key cannot be changed after it has been registered.

The server expires peers after 30 minutes; the only way to change the key is to wait for the peer to expire from the server.

3.3. Cryptographic keys

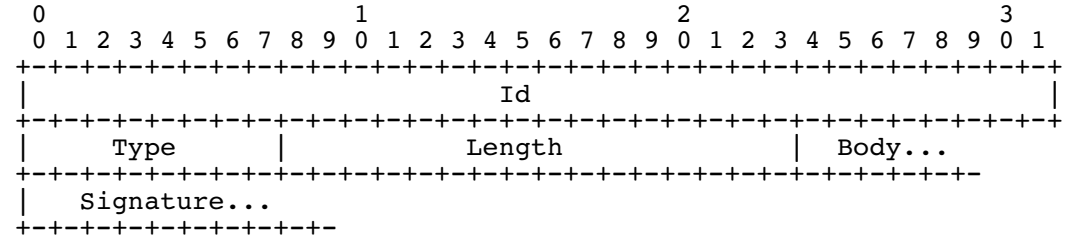
In order to obtain the public key of a peer ϕ, a client sends a `GET` request to the URL `/peers/`ϕ`/key`. The server replies with a 200 reply with the 64-byte key in the body.

3.4. Peer addresses

In order to discover the addresses of a peer ϕ, a peer sends a `GET` request to the URL `/peers/`ϕ`/addresses`. The server replies with a list of UDP socket addresses, one per line.

4. Peer-to-peer protocol

The server and all peers participate in a UDP-based peer-to-peer protocol. The protocol has a strict request-response structure, but it is symmetric: requests can be sent by either peer at any time. All messages have the following format:



The field *Type* indicates the type of the message; values 0 to 127 indicate requests, values 128 to 255 indicate replies. The field *Id* is arbitrary in requests, and is copied from the request to the reply. The field *Length* indicates the length of the field *Body*.

The body is optionally followed by a cryptographic signature, as defined in Section 4.3. The signature is 64 bytes long, and any extra bytes following the signature must be ignored.

4.1. Details of individual messages

4.1.1. Ping

The message *Ping* = 0 causes the peer to reply with amessage *Ok* = 128. Both messages have an empty body.

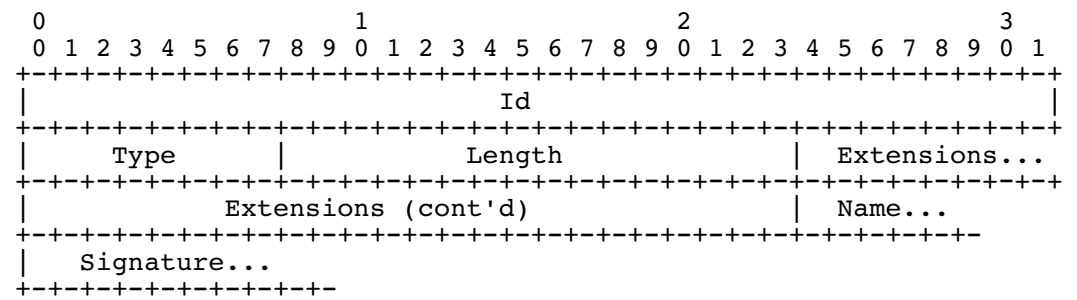
4.1.2. Error

The message *Error* = 129 may be sent in reply to any request, and is used to send a human-readable error message. The body consists of a human-readable UTF-8 string.

4.1.3. Handshake

Before they can communicate, two peers perform a handshake by exchanging a pair of *Hello* = 1 and *HelloReply* = 130 messages. This exchange is compulsory: a peer might ignore messages from a peer that didn't perform the handshake correctly.

Hello and *HelloReply* messages have the following format:



The field *Extensions* is a 32-bit bitmap of supported protocol extensions (optional features), see 4.2. The field *Name* contains the name of the sending peer.

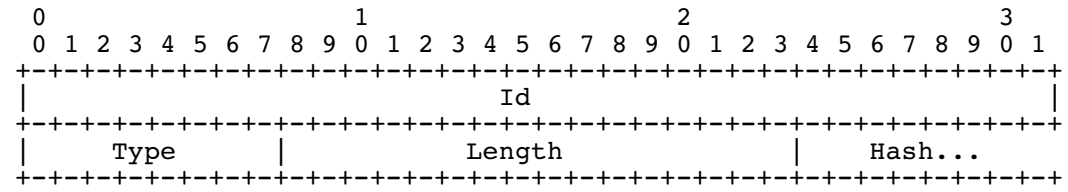
In order to verify the signature of the sending peer, the receiver of a *Hello* message must contact the server[1]. For that reason, it may take up to a few seconds to send a *HelloReply*, and the sender must use a large enough timeout before resending or giving up on a *Hello* message.

After two peers have exchanged *Hello* and *HelloReply* messages, they have established an *association*. Associations expire after a timeout that is no less than 5 minutes: if a peer ϕ hasn't received any message from a peer ψ for 5 minutes, it may forget its association with ϕ. A peer that wishes to maintain an association should send *Ping* messages after at most 4 minutes without a message exchange. (This mechanism also ensures that NAT mappings remain active.)

4.1.4. Root

The message *RootRequest* = 2 requests that the peer send its root hash, the hash of the datum representing the root of its filesystem tree; this message has an empty body. The peer replies with *RootReply* = 131, whose body contains the root hash as a string of 32 bytes.

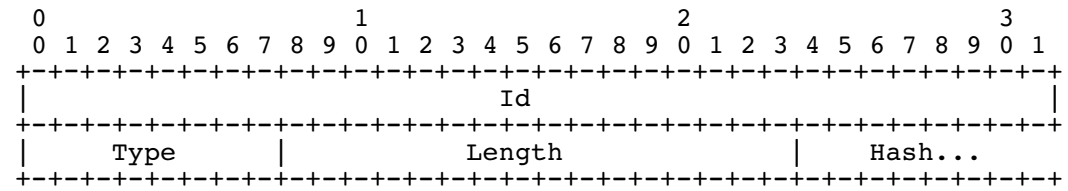
The *Root* message has the following format:



4.1.5. Data

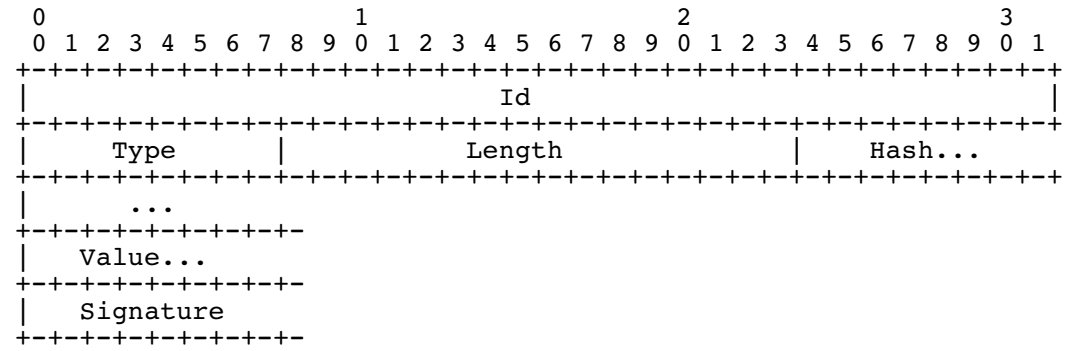
The message *DatumRequest* = 3 requests that a peer send a specific datum. The peer responds with either a message of type *Datum* = 132 if it has the corresponding datum, or a message of type *NoDatum* = 133 if it does not have a datum with the given hash.

The messages *DatumRequest* and *NoDatum* have the following format:



The message *Datum* has the following format:

1. This is a flaw of the protocol. A better solution would be to use certificates (public keys signed by the server), but this project is designed to be doable by students who are not cryptography specialists.



In order to ensure the integrity of the data, it is *required* to verify not only that the hash in the reply is equal to the hash in the request, but also to hash the data at the receiver and verify that the hash correponds to the one encoded in the message.

4.1.6. NAT traversal

NAT traversal is performed using an intermediary node. A peer announces that it is willing to act as an intermediary for NAT traversal by setting the bit 0 (the right-most bit) in the Extensions field of its *Hello* or *HelloReply* message. The server sets this bit, and may therefore be used as an intermediary for NAT traversal.

After a peer *A* has attempted to contact a peer *B* and failed, it may attempt NAT traversal. In order to do so, it sends a *NatTraversalRequest* = 4 message with *B*'s address to an intermediary peer *S* (typically the server) that is not behind NAT. *S* replies with *Ok*, and sends a *NatTraversalRequest2* = 5 message to the peer *B*, which replies with *Ok*. *B* then sends a *Ping* request to *A*.

Note that the peer *S* replies to the original *NatTraversalRequest* as soon as it has received the request. Peer *A* has no guarantee that peer *B* has received the request: this subprotocol is unreliable.

Peers that are not willing to act as relays for NAT traversal do not need to implement the *NatTraversalRequest* message, but all peers must implement the *NatTraversalRequest2* message.

Both messages have the same syntax: they contain a single socket address (IP address and port), and have a length of 6 bytes (for IPv4) or 18 bytes (or IPv6).

4.2. Extension mechanism

The protocol is extensible: new messages can be added to the protocol. A peer indicates that it understands messages outside of the base protocol by setting a bit in the *Extensions* field of the *Hello* or *HelloReply* packet.

Currently, the following extensions are defined:

- 0 (right-most bit of the extensions field): this peer is willing to act as an intermediate node for NAT traversal.

I act as the naming authority for the extensions space: if you need to define a new extension for your project, please contact me by e-mail, and I will assign you an integer between 1 and 31 that identifies your extension.

4.3. Cryptographic signatures

Messages may be signed with an ECDSA signature. The signature covers the whole packet up to the end of the *Body* field, i.e. bytes 0 through `Length` + 7 inclusive[2].

Since elliptic curve operations are expensive, not all messages are signed. The following messages *must* be signed, and should be dropped by the receiver if they are not correctly signed:

- *Hello*, *HelloReply*,
- *RootReply*,
- *NoDatum*,
- *NatTraversalRequest* and *NatTraversalRequest2*.

Other messages need not be signed, since they either are not security-critical, or their contents is protected by the Merkle tree. In particular, the messages *DataRequest* and *DataReply*, which constitute the bulk of the traffic, should not be signed. See Appendix A for details of the cryptographic algorithms.

5. Data structures

The transport subprotocol described above is data type agnostic: *Data* messages carry arbitrary strings of bytes. At some point, however, this data needs to be formatted by the sender and interpreted by the receiver. In this protocol, the contents of *Data* messages carry nodes of a *Merkle Tree*[3].

Every peer exports a filesystem tree. The Merkle tree contains four kinds of nodes:

- *chunk* nodes, which conain a sequence of at most 1024 bytes of data;
- *directory* nodes, which have between 0 and 16 children, which represent a directory or part of a directory of at most 16 entries;
- *big* nodes, which have between 2 and 32 children, and represent a piece of a file or a file of more than 1024 bytes;
- *big directory* nodes, which have between 2 and 32 children, and represent a piece of a directory or a directory of more than 16 entries.

Nodes are represented by a single byte indicating the type of the node immediately followed by the node data; the type byte is the first byte of the body of the *Value* field of a *Data* node. The type field can take the following values:

- *Chunk* = 0 indicates a chunk of data; the data immediately follows the type field;
- *Directory* = 1 indicates a directory or a directory fragment; the data that follows the type field is constituted of a number n ($0 \leq n \leq 16$) directory entries of 64 bytes each having the following structure:
  - 32 bytes containing the filename, padded with 0 bytes if necessary[4]

2. Which does not follow best practices: the signature should include the sender's and receiver's addresses in order to bind the message. But this project is designed to be easy to do.
3. `https://en.wikipedia.org/wiki/Merkle_tree`
4. I know, I know, I should be using a type-value pair here. But this is simpler.

  - 32 bytes containing the hash of the datum containing the file contents.
- *Big* = 2 represents the concatenation of its children; a list of 2 to 32 hashes immediately follows the type field; all children of a *Big* node must be either *Chunk* or *Big* nodes;
- *BigDirectory* = 3 have the same structure as *Big* nodes, but their children must be either *Directory* or *BigDirectory* nodes.

6. Minimal solution

You are expected to write a program that participates in the protocol described above. Your program may be written in the programming language of your choice, but must compile on a Debian Linux machine without installation of additional software. At the very minimum, your program should:

- register with the server and maintain its association for unbounded periods of time;
- make files available to other peers, both when behind NAT and not behind NAT;
- make directories of less than 16 entries available to other peers;
- download files from a peer not behind NAT.

The efficiency of your implementation will be taken into account in the evaluation. For example, I will take into account whether your implementation has a single packet in flight, whether it uses a sliding window, and whether it implements a congestion control algorithm.

The functionality of your implementation will be taken into account in the evaluation. For example, I will take into account whether your implementation is able to download single files selected by the user, or whether it can only download a full filesystem tree.

Other features (NAT traversal, user interface etc.) will be taken into account in the evaluation, but will most probably not prevent you from getting a passing grade.

7. Suggested extensions

I expect most of you to implement features beyond the minimal protocol described above. Any features that you choose to implement will be considered with interest and indulgence, and will be taken into account in the evaluation.

The most obvious extension is to provide a useful user interface. While a user interface will be welcome, do not spend too much time on it: this is a networking project, not a user interfaces project.

Obviously, you will want to implement optional features of the protocol, such as NAT traversal, large directories. Less obviously, you will want to implement features that are made possible by the protocol but not obvious. For example, you will want to make downloads faster by having multiple requests in flight (in which case I will ask you about congestion control), you will want to implement streaming downloads (think about watching a remote movie), and you might want to implement range requests.

The protocol guarantees authenticity but not confidentiality. You might want to implement a protocol extension for encrypted data transfer. If you do that, I will ask you whether it has the property of forward secrecy (Diffie-Hellman and its variants are your friends).

Any other extensions will be gladly accepted, and evaluated with indulgence and openness of mind.

8. Submission rules

You will submit your source code in a file called `name1-name2.tar.gz`, where *name1* and *name2* are your names. For example, if your names are Hugo Steinhaus and Stefan Banach, you will submit a file called `banach-steinhaus.tar.gz`.

The file you submit will contain the following:

- the complete source code of your program;
- a text file called README that indicates how to build and execute your program;
- a report in PDF format that indicates, among others:
  - what part of the project has been implemented;
  - what extensions have been implemented;
  - the parts of the program that are not original (for which you received help from your friendsm or from the Internet).

It is *compulsory* to clearly credit the sources of any help that you received: if you did receive external help, you *must* give proper credit, or you will be accused of plagiarism. For example, if you received help from a friend, you must indicate the name of the friend and which parts were done with their help. If you copied code from an online resource, you must give a pointer to the online publication. Note that an LLM (an "artificial intelligence chatbot") is not an acceptable source: if you receive help from an LLM, you must cite the original source that was used for training the LLM.

A. Implementation of cryptographic signatures

An ECDSA public key is a pair of integers (x, y). A signature is a pair of integers (r, s). In this project, we represent these pairs of integers as strings of 64 bytes, where the first 32 bytes represent the first integer and the second 32 bytes represent the second one.

In the following paragraphs, we provide implementations of the necessary cryptographic primitives in Go and Python. You are welcome to write your code in a different language, but in that case you will need to work out on your own how to implement the cryptographic primitives.

A.1. Implementation in Go

Preliminaires:

```
import (
    "crypto/ecdsa"
    "crypto/elliptic"
    "crypto/rand"
    "crypto/sha256"
```

```
    "math/big"
)
```

Generate a private key:

```
privateKey, err :=
    ecdsa.GenerateKey(elliptic.P256(), rand.Reader)
```

Extract the public key from a private key:

```
publicKey, ok := privateKey.Public().(*ecdsa.PublicKey)
```

Format a public key as a string of 64 bytes:

```
formatted := make([]byte, 64)
publicKey.X.FillBytes(formatted[:32])
publicKey.Y.FillBytes(formatted[32:])
```

Parse a public key:

```
var x, y big.Int
x.SetBytes(data[:32])
y.SetBytes(data[32:])
publicKey := ecdsa.PublicKey{
    Curve: elliptic.P256(),
    X: &x,
    Y: &y,
}
```

Compute the signature of a message:

```
hashed := sha256.Sum256(data)
r, s, err := ecdsa.Sign(rand.Reader, privateKey, hashed[:])
signature := make([]byte, 64)
r.FillBytes(signature[:32])
s.FillBytes(signature[32:])
```

Verify a signature:

```
var r, s big.Int
r.SetBytes(signature[:32])
s.SetBytes(signature[32:])
hashed := sha256.Sum256(data)
ok = ecdsa.Verify(publicKey, hashed[:], &r, &s)
```

A.2. Implementation in Python

Preliminaries:

```
import ecdsa
import hashlib
```

Generate a private key:

```
privateKey = ecdsa.SigningKey.generate(
    curve=ecdsa.NIST256p, hashfunc=hashlib.sha256,
)
```

Extract the public key:

```
publicKey = privateKey.get_verifying_key()
```

Format a public key:

```
publicKey.to_string()
```

Parse a public key:

```
publicKey = ecdsa.VerifyingKey.from_string(
    body, curve=ecdsa.NIST256p, hashfunc=hashlib.sha256,
)
```

Compute the signature of a message:

```
signature = privateKey.sign(data)
```

Verify a signature:

```
ok = publicKey.verify(signature, data)
```