Internet Protocols Project: Distributed Filesystem

Julien Coolen and Minoo Farsiabi

December 6, 2021

Contents

1 Usage

1.1 Start the node executable

Enter in the shell `go run dfs/node <peer_name>`.

1.2 Start the RPC HTTP client

1.2.1 List registered peers

Enter `go run dfs/rpc_client peers`

1.2.2 List files

Examples:

```
$ go run dfs/rpc_client ls -path / -peer jch.irif.fr
file: README.txt (50 bytes)
directory: documents (153177 bytes)
directory: images (224525 bytes)

$ go run dfs/rpc_client ls -path /documents -peer
 ↪  jch.irif.fr
file: internet1.pdf (44173 bytes)
file: internet2.pdf (54147 bytes)
file: internet3.pdf (54857 bytes)
```

1.2.3 Download files

Examples:

```
go run dfs/rpc_client download  -peer jch.irif.fr -hash
 ↪  <hash>

go run dfs/rpc_client downloadFromPath -path /documents
 ↪  -peer jch.irif.fr
```

2 Architecture

A thread listens to incoming packets, parses them, and notifies other threads that the message was received through a the following callback mechanism. The following hash map `PendingPacketQueries map[uint32]chan []byte` holds the pending queries, indexed by the transaction ID. Each thread is notified when the packet is returned thanks to Go's channels.

3 Extensions

3.1 ECDSA Signatures

3.2 Caching

We store the nodes of the Merkle tree in a LRU (thus, bounded) cache for later retrieval.

3.3 NAT Traversal

We implement the protocol described the project brief. Here is a tcpdump trace a node behind a NAT contacting another one behind a NAT:

```
2021/12/04 17:29:59 Addr = 78.199.36.29:8088
2021/12/04 17:29:59 Got addr = 78.199.36.29:8088
2021/12/04 17:30:00 limit reached
2021/12/04 17:30:09 stop
2021/12/04 17:30:09 cannot contact addr 78.199.36.29:8088
2021/12/04 17:30:09 Sent nat traversal request to
 ↪  Juliusz's peer
2021/12/04 17:30:09 Hello from 78.199.36.29:8088
2021/12/04 17:30:10 Sent hello to 78.199.36.29:8088
2021/12/04 17:30:10 HelloReply from 78.199.36.29:8088
```

3.4 HTTP RPC Server and Client

The peer executable, which we call node (a node as part of the peer-to-peer network), can be queried thanks to RPC HTTP calls. Users can send queries to the node using the client executable.

3.5 Extension 4: Session Key

We create a session key for two peers using the Diffie-Hellman key exchange protocol on the P-256 or P-521 curve. To prevent man-in-the-middle attacks from happening, we require the key exchange to be authenticated using ECDSA signatures.

According to the protocol, the extension 4 defines two packet numbers 68 and 196, which we define as follows:

1. Packet type 68 (64+4) DHKeyRequest. We use the standard generator for the P-256 curve (or P-512), as given in `https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.186-4.pdf#page=100`.
2. Packet type 196 (192+4) DHKey(g^s), for the standard generator g and the private key s.

One peer A sends a DHKeyRequest packet to an other peer B, which replies back with a DHKey(g^{s_B}). Upon receiving the latter packet, A sends a DHKey(g^{s_A}), thereby establishing the key session $(g^{s_A})^{s_B}$.

3.6 Extension 5: Symmetric Encryption

Once two peers established a common secret session key in an authenticated way, they can encrypt their packets using a symmetric encryption algorithm.

We present two encrypt-then-authenticate schemes, and implement the latter one:

1. We use as seen in class AES-128 bit in CBC mode, for the body of an already defined packet and its type (1 byte long):

$$\begin{aligned} m &= \text{type} \,||\, \text{body} \,||\, \text{fingerprint(public\_key)} \\ s &= \text{encrypt}(m) \,||\, \text{MAC}(m, \text{private\_key}) \\ s' &= s \,||\, \text{Sign}(s) \end{aligned}$$

   We do a combination of encrypt-then-sign and encrypt-then-MAC. In order to prevent an attacker from replacing the signature with his own, we need to ensure that the plaintext is tied to the signature. Hence we include in the plaintext a fingerprint of the public key.

2. The previous way is a bit convoluted. This can be simplified with the use of an Authenticated Encryption with Associated Data (AEAD). This algorithm combines a cipher and a MAC using a single key. We use AES-GCM, which is an AEAD. Indeed, the GCM mode is parallelized like CTR but also does MAC. According to Cloudflare blog: "Using a dedicated AEAD reduces the dangers of bad combinations of ciphers and MACs, and other mistakes, such as using related keys for encryption and authentication.

   Given the many vulnerabilities related to the use of AES-CBC with HMAC, and the weakness of RC4, AES-GCM is the de-facto secure standard on the web right now, as the only IETF-approved AEAD to use with TLS at the moment". Packets are encrypted-then-authenticated, and signed, as follows:

$$\begin{aligned} m &= \text{type} \mathbin{||} \text{body} \\ s &= \text{encrypt}(m, \text{nonce}, \text{additional\_data} = \text{Sign}(m)) \\ s' &= s \mathbin{||} \text{Sign}(m) \end{aligned}$$

This method has some advantages over the previous way:

(a) the signature proves the same thing as in the unencrypted version (signature of the plaintext, and not of the ciphertext)

(b) we also MAC the signature

(c) we bind the signature to its message thanks to the additional_data field

This does not fall into cryptographic doom since the authentication tag is checked before (or in Go's library, concurrently with) the decryption.

The structure of an encrypted packet is given in figure 1.

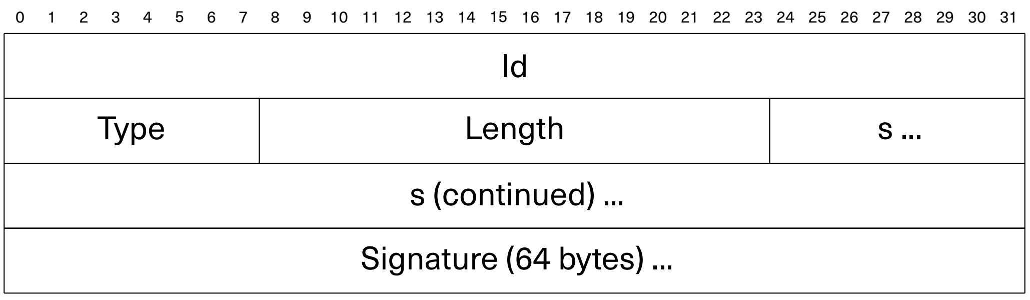

Figure 1