Storj Protocol Specification

Version 2 (May 23, 2017)

Gordon Hall (gordonh@member.fsf.org)
Braydon Fuller (braydon@storj.io)
Ryan Foran (ryan@storj.io)

0: License

1: Introduction

This specification documents the Storj network protocol in its entirety for the purpose of enabling its implementation in other languages. Described here, is the protocol **base** - the minimum specification for compatibility with the Storj network. Additional optional extensions to this work are defined as Storj Improvement Proposals (or "SIPs"), some of which have been folded into the base protocol since Version 1.

2: Identities

Every node (host computer speaking the Storj protocol) on the network possesses a unique cryptographic identity. This identity is used to derive a special 160 bit identifier for the purpose of organizaing the overlay structure and routing messages *(3.1: Kademlia)*. In order for a node to join the network it must generate an identity.

Identities are described as **hierarchically deterministic** and serve the purpose of running a cluster of nodes that can all share the same contracts and act on behalf of each other in the network. The specification extends Bitcoin ECDSA derivation standard BIP32 and BIP43.

Key derivation must match the specification of Bitcoin Hierarchical Deterministic Wallets (BIP32) with the purpose field described in Bitcoin Purpose Field for Deterministic Wallets (BIP43).

We define the following levels in BIP32 path:

```
m / purpose' / group_index' / node_index
```

The apostrophe in the path indicates that BIP32 hardened derivation is used. Purpose is a constant set to 3000, so as to not collide with any bitcoin related proposals which recommends to use the BIP number.

```
m / 3000' / group_index' / node_index
```

The `group_index` for most purposes will be 0. However is reserved for a future use to be able to increment in the case that the contracts should be updated with a new key. The `node_index` can be a number from 0 through 2 ^ 31 - 1, so that it's using a non-hardened paths and it's always possible to derive the public key for a node using the `m / 3000' / group_index'` derived extended public key. This gives a total of 2.147 billion possible nodes to run in a group cluster.

> As noted in BIP32, a compromised private key at the `node_index` level in combination with the extended public key at the `group_index` level will compromise all descending private keys derived from the `group_index` level, this is the rationale for a hardened path for the `group_index`.

In every message exchanged on the network, each party will include a tuple structure which includes enough information to locate and authenticate each party.

```
["<node_id>", { /* <contact_hash_map> */ }]
```

2.1: Node ID Generation

Once a HD identity has been generated, a child identity should be derived and used for a single node. The resulting public key from that child identity is used to derive the Node ID. The node's identifier is the `RIPEMD160( SHA256( CHILD_PUBLIC_KEY ) )` encoded in hexidecimal. This value is inserted as the first item in the identity tuple.

```
["705e93f855e60847fda4c48adff0dc1b1f7c40ef", { /* <contact_hash_map> */ }]
```

2.2: Contact Hash Map

The second entry in the identity tuple contains additional information specific to addressing the node on the network. This includes:

```
{
  "hostname": "ip.address.or.domain.name",
  "port": 8443,
  "protocol": "https:",
  "xpub": "<child_identity_public_extended_key>",
```

```
  "index": "<child_identity_derivation_index>"
}
```

Additional properties may be included based on individual use cases within the network, however the properties above are **required**.

3: Network Structure

Storj employs a **structured** network, meaning that nodes are organized and route messages based on a deterministic metric. The network uses a Kademlia distributed hash table as the basis for the network overlay. In addition to Kademlia, Storj also employs other extensions to mitigate issues and attacks defined by the work on S/Kademlia.

In addition to the distributed hash table, Storj also implements a publish-subscribe system, Quasar, atop the Kademlia overlay to provide effective delivery of publications related to the solicitation of storage space *(6: Storage Contracts)*.

3.1: Kademlia

Once a Storj node has completed generating its identity, it bootstraps its routing table by following the Kademlia "join" procedure. This involves querying a single known "seed" node for contact information about other nodes that possess a Node ID that is close (XOR distance) to its own *(4.4 FIND_NODE)*. This is done iteratively, sending the same query to the `ALPHA` (3) results that are closest, until the further queries no longer yield results that are closer or the routing table is sufficiently bootstrapped.

3.2: Quasar

Upon successfully bootstrapping a routing table, a node may choose to subscribe to certain publication topics related to types of storage contracts they wish to accept *(6.2 Topic Codes)*. Each node in the network, maintains an attenuated bloom filter, meaning a **list** of exactly 3 bloom filters, each containing the the topics in which neighboring nodes are interested.

```
Filter 0 [...] - Topics which WE are subscribed
Filter 1 [...] - Topics which OUR 3 NEAREST NEIGHBORS are subscribed
Filter 2 [...] - Topics which OUR NEIGHBORS' 3 NEAREST NEIGHBORS' are subscribed
```

The Storj network expects these blooms filters to be constructed and modified in a specific manner. Each filter's bitfield must be exactly 160 bits in size. Items are hashed with FNV in a manner consistent with the paper "Less Hashing, Same Performance: Building a Better Bloom Filter" using 2 slices. To illustrate the hashing in pseudo-code:

```
function calc(key, size = 160, slices = 2) {
  function fnv(seed, data) {
    const hash = new FNV();

    hash.update(seed);
    hash.update(data);

    return hash.value() >>> 0;
  }

  const hash1 = fnv(Buffer.from([83]), key);
  const hash2 = fnv(Buffer.from([87]), key);
  const hashes = [];

  for (let i = 0; i < slices; i++) {
    hashes.push((hash1 + i * hash2) % size);
  }

  return hashes;
}
```

The above example illustrates how to calculate the values to insert into a bloom filter for a given key (or topic code). To subscribe to a given topic, the code(s) should be processed as shown above and then inserted into the filter at index 0. Once the filter at index 0 represents what the node wants to receive, it must exchange this information with its 3 nearest neighbors *(4.7 SUBSCRIBE + 4.8 UPDATE)*. This allows publications to be properly relayed to nodes who are most likely to be subscribed to the given topic.

3.3: Transport

TODO

4: Remote Procedure Calls

TODO

4.1: Structure and Authentication

TODO

4.2: PROBE

TODO

4.3: PING

TODO

4.4: FIND_NODE

TODO

4.5: FIND_VALUE

TODO

4.6: STORE

TODO

4.7: SUBSCRIBE

TODO

4.8: UPDATE

TODO

4.9: PUBLISH

TODO

4.10: OFFER

TODO

4.11: CONSIGN

TODO

4.12: `AUDIT`

TODO

4.13: `MIRROR`

TODO

4.14: `RETRIEVE`

TODO

4.15: `RENEW`

TODO

4.16: `ALLOCATE`

TODO

4.17: `CLAIM`

TODO

4.18: `TRIGGER`

TODO

5: Data Transfer Endpoints

TODO

5.1: Uploading

TODO

5.2: Downloading

TODO

6: Storage Contracts

TODO

6.1: Descriptor Schema

TODO

6.2: Topic Codes

TODO

7: Retrievability Proofs

TODO

8: References

- Storj Improvement Proposals (`https://github.com/storj/sips`)
- BIP32 (`https://github.com/bitcoin/bips/blob/master/bip-0032.mediawiki`)
- BIP43 (`https://github.com/bitcoin/bips/blob/master/bip-0043.mediawiki`)
- Kademlia (`http://www.scs.stanford.edu/~dm/home/papers/kpos.pdf`)
- S/Kademlia (`http://www.tm.uka.de/doc/SKademlia_2007.pdf`)
- Quasar (`https://www.microsoft.com/en-us/research/wp-content/uploads/2008/02/iptps08-quas`
- FNV (`https://en.wikipedia.org/wiki/Fowler%E2%80%93Noll%E2%80%93Vo_hash_function`)
- Less Hashing, Same Performance: Building a Better Bloom Filter (`http://www.eecs.harvard.edu/~michaelm/postscripts/rsa2008.pdf`)