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> */ }]
```

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> */ }]
```

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 [...] - WE are subscribed
Filter 1 [...] - 3 NEAREST NEIGHBORS are subscribed
Filter 2 [...] - 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

The Storj network operates entirely over HTTPS. TLS *must* be used - there is no cleartext supported. In general this means that certificates are self-signed and you must accept them in order to communicate with others on the network. Because of this, it is recommended that certificate pinning be used when implementing the Storj protocol.

> It's important to note that while it may be possible for a man-in-the-middle to intercept RPC messages if certificate pinning is not used, it is not possible for this attacker to manipulate messages. Furthermore, data transfer is encrypted such that only the sender is capable of decrypting it. The most damage this type of attack could cause is targeted denial-of-service.

Each Storj node exposes 2 endpoints to other nodes; one for receiving RPC messages *(4. Remote Procedure Calls)* and the other for serving and accepting raw data streams associated with held contracts *(5. Data Transfer Endpoints)*. Requests sent to the RPC endpoint require a special HTTP header `x-kad-message-id` to be included that matches the `id` parameter in the associated RPC message *(4.1 Structure and Authentication)*.

4 Remote Procedure Calls

- **Method:** `POST`
- **Path:** `/rpc/`
- **Content Type:** `application/json`
- **Headers:** `x-kad-message-id`

4.1 Structure and Authentication

Each remote procedure call sent and received between nodes is composed in the same structure. Messages are formatted as a JSON-RPC 2.0 *batch* payload containing 3 objects. These objects are positional, so ordering matters. The anatomy of a message takes the form of:

```
[{ /* rpc */ },{ /* notification */ },{ /* notification */ }]
```

At position 0 is the RPC request/response object, which must follow the JSON-RPC specification for such an object. It must contain the properties: `jsonrpc`, `id`, `method`, and `params` if it is a request. It must contain the properties: `jsonrpc`, `id`, and one of `result` or `error` if it is a response.

At positions 1 and 2 are a JSON-RPC notification object, meaning that it is not required to contain an `id` property since no response is required. These two notifications always assert methods `IDENTIFY` and `AUTHENTICATE` respectively. Together, these objects provide the recipient with information regarding the identity and addressing information of the sender as well as a cryptographic signature to authenticate the payload.

> Positions 3 and beyond in this structure are reserved for future protocol extensions related to global message processing.

Example: Request

```
[
  {
    "jsonrpc": "2.0",
    "id": "<uuid_version_4>",
    "method": "<method_name>",
    "params": ["<parameter_one>", "<parameter_two>"]
  },
  {
```

```
    "jsonrpc": "2.0",
    "method": "IDENTIFY",
    "params": [
      "<public_key_hash>",
      {
        "hostname": "sender.hostname",
        "port": 8443,
        "protocol": "https:",
        "xpub": "<public_extended_key>",
        "index": "<child_key_derivation_index>"
      }
    ]
  },
  {
    "jsonrpc": "2.0",
    "method": "AUTHENTICATE",
    "params": [
      "<payload_signature>",
      "<child_public_key>",
      ["<public_extended_key>", "<child_key_derivation_index>"]
    ]
  }
]
```

Example: Response

```
[
  {
    "jsonrpc": "2.0",
    "id": "<uuid_version_4_from_request>",
    "result": ["<result_one>", "<result_two>"]
  },
  {
    "jsonrpc": "2.0",
    "method": "IDENTIFY",
    "params": [
      "<public_key_hash>",
      {
        "hostname": "receiver.hostname",
        "port": 8443,
        "protocol": "https:",
        "xpub": "<public_extended_key>",
        "index": "<child_key_derivation_index>"
      }
    ]
```

```
  },
  {
    "jsonrpc": "2.0",
    "method": "AUTHENTICATE",
    "params": [
      "<payload_signature>",
      "<child_public_key>",
      ["<public_extended_key>", "<child_key_derivation_index>"]
    ]
  }
]
```

In the examples above, `public_key_hash` and `child_public_key` must be encoded as hexidecimal strings, `public_extended_key` must be encoded as a base58 string (in accordance with BIP32), and `payload_signature` must be encoded as a base64 string which is the concatenation of the public key recovery number with the actual signature of the payload - excluding the object at index 2 (`AUTHENTICATE`). This means that the message to be signed is `[rpc, identify]`.

> Note the exclusion of a timestamp or incrementing nonce in the payload means that a man-in-the-middle could carry out a replay attack. To combat this, it is urged that the `id` parameter of the RPC message (which is a universally unique identifier) be stored for a reasonable period of time and nodes should reject messages that attempt to use a duplicate UUID.

The rest of this section describes each individual method in the base protocol and defines the parameter and result signatures that are expected. If any RPC message yields an error, then an `error` property including `code` and `message` should be send in place of the `result` property.

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

Initiating the transfer of data between nodes after a contract has been signed is straightforward. First, the initiator must request a transfer token from the custodian. If uploading the shard for the first time to a farmer, a CONSIGN RPC *(4.13 CONSIGN)* must be sent. If downloading the shard, a RETRIEVE RPC *(4.14 RETRIEVE)* is sent. The result of either of those messages should yield an authorization token that is included in the query string of the next request.

5.1 Uploading

- **Method:** `POST`
- **Path:** `/shards/{hash}?token={consign_token}`
- **Content Type:** `binary/octet-stream`

5.2 Downloading

- **Method:** `GET`
- **Path:** `/shards/{hash}?token={retrieve_token}`
- **Content Type:** `binary/octet-stream`

6 Storage Contracts

TODO

6.1 Descriptor Schema

TODO

6.2 Topic Codes

Storj defines a matrix of *criteria* and *descriptors* in the form of codes representing the degree of which the criteria must be met. The resulting topic code is derived from the associated contract and is used as the key for cross- referencing neighborhood bloom filters to determine how the publication should be routed. At the time of writing, there are 4 criteria column in the topic matrix:

- **Size:** refers to the size of the data to be stored
- **Duration:** refers to the length of time which the data should be stored
- **Availability:** refers to the relative uptime of required by the contract
- **Speed:** refers to the throughput desired for retrieval of the stored data

At the time of writing, there are 3 descriptor opcodes representing *low*, *medium*, and *high* degrees of the criteria.

- **Low:** `0x01`
- **Medium:** `0x02`
- **High:** `0x03`

> The ranges represented by these descriptors are advisory and may change based on network performance and improvements to hardware over time.

When publishing or subscribing to a given topic representing the degrees of these criteria, nodes must serialize the opcodes as the hex representation of the bytes in proper sequence. This sequence is defined as:

`prefix + size + duration + availability + speed`

The prefix byte is the static identifier for a contract publication. Contracts may not be the only type of publication relayed in the network, so the prefix acts as a namespace for a type of publication topic. The prefix for a contract publication is `0x0f`.

To illustrate by example, we can determine the proper topic by analyzing the use case for a given file shard. For instance, if we want to store an asset that is displayed on a web page we can infer the following:

- The file is small
- The file may change often, so we should only store it for medium duration
- The file needs to always be available
- The file should be transferred quickly

Using the matrix, we can determine the proper opcode sequence:

`[0x0f, 0x01, 0x02, 0x03, 0x03]`

Serialized as hex, our topic string becomes:

`0f01020303`

Another example, by contrast, is data backup. Data backup is quite different than the previous example:

- The file is large (perhaps part of a hard drive backup)
- The file will not change and should be stored long term
- The file will not be accessed often, if ever
- The file does not need to be transferred at high speed

Using the matrix, we can determine the proper opcode sequence:

`[0x0f, 0x03, 0x03, 0x01, 0x01]`

Serialized as hex, our topic string becomes:

`0f03030101`

The resulting hex string from the serialized opcode byte sequence should be used as the `topic` parameter of a `PUBLISH` RPC *(4.9 PUBLISH)*. Nodes that are subscribed to the topic will receive the proposed storage contract and may begin contract negotiation with you directly.

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`)