Swarm Protocol Specification

Swarm Research Division

draft of September 11, 2025

Contents

List of definitions

List of Figures

List of Tables

Chapter 1

Protocols

1.1 Transport layer

Swarm is a network operated by its users. Each node in the network is supposed to run a client complying with the protocol specifications. On the lowest level, the nodes in the network connect using a peer-to-peer network protocol as their transport layer. This is called the underlay network. In its overall design, Swarm is agnostic of the particular underlay transport used as long as it satisfies the following requirements.

Underlay network

The libp2p networking stack provides all the required properties for the underlay network.

1. *Addressing* is provided in a form of multi address for every node, which is referred here as the underlay address. Every node can have multiple underlay addresses depending on transports and network listening addresses that are configured.
2. *Dialing* is provided over libp2p supported network transports.
3. *Listening* is provided by libp2p supported network transports.
4. *Live connections* are established between two peers and kept open for accepting or sending messages.
5. *Channel security* is provided with TLS and libp2p `secio` stream security transport.
6. *Protocol multiplexing* is provided by libp2p `mplex` stream Multiplexer protocol.
7. *Delivery guarantees* are provided by using libp2p bidirectional streams to validate the response from the peer on sent message.
8. *Serialization* is not enforced by libp2p, as it provides byte streams allowing flexibility for every protocol to choose the most appropriate serialization.

The underlaying transport layer that is used by all protocols is based on libp2p. This chapter only contains the information that is required to be taken into account by any client that intends to connect to the swarm network.

libp2p uses cryptographic key pairs to sign messages and derive unique peer identities (or "peer ids").

Although private keys are not transmitted over the wire, the serialization format used to store keys on disk is also included as a reference for libp2p implementors who would like to import existing libp2p key pairs.

Although RSA and Ed25519 should work fine - Swarm uses ECDSA secp256R1 and this standard is required by any alternative client that wants to connect to swarm.

Key encodings and message signing semantics are covered on this link.

Note that `PrivateKey` messages are never transmitted over the wire. Current libp2p implementations store private keys on disk as a serialized `PrivateKey` protobuf message. libp2p implementors who want to load existing keys can use the `PrivateKey` message definition to deserialize private key files.

In Swarm, the key value stored in the file is encrypted using a password.

What are keys used?

Keys are used in two places in libp2p. The first is for signing messages and the second is for generating peer ids.

Addressing

Once generated the keys should be persisted to avoid them getting re-generated "on-the-fly" and generating unnecessary churn in the network.

More detailed information on how nodes address each other in swarm can be found in the official libp2p documentation, specifically about peer ids format and addressing

Connecting to the swarm network using `dnsaddr` links

Swarm supports the `dnsaddr` format (example: `/dnsaddr/mainnet.ethswarm.org`). A detailed document about the format is describer here

In order for an alternative client to connect to swarm it needs to be able to resolve this format to a valid underlay address.

The steps are:

- getting the TXT value from the DNS server:
  - _dnsaddr.mainnet.ethswarm.org. 30 IN TXT "dnsaddr=/dnsaddr/swarm-1.mainnet.ethswarm.org"
- resolving this value to it's next TXT record:
  dig TXT _dnsaddr.swarm-1.mainnet.ethswarm.org _dnsaddr.swarm-1.mainnet.ethswarm.org. 30 IN TXT "dnsaddr=/dnsaddr/bee-1.mainnet.ethswarm.org"
- resolving the TXT record to the underlay address value:
  _dnsaddr.bee-1.mainnet.ethswarm.org. 30 IN TXT "dnsaddr=/ip4/3.75.238.129/tcp/31101/p2p/Qm
- the resulting address should be accessible and accepting connections.

While the implementation can support connecting to multiple addresses exposed by the same node (for better connectivity) it is enough to connect to one to join the swarm network.

Worth mentioning that the level at which the underlay value resides can be arbitrary and an alternative client should support recursive lookups until such value is found.

Optional components

NAT - the client is free to use any available tool as it does not interfere with the network in any significant way.

The recommended serialization is Protobuf with varint delimited messages in streams.

1.2 Protocols

Swarm Bee organizes P2P communication in protocols as logical units under a unique name that may define one or more streams for communication.

Headers

A Swarm specific requirement for all libp2p streams is to exchange Header protobuf messages on every stream initialization between two peers. This message encapsulates a stream scoped information that needs to be exchanged before any stream specific data or messages are exchanged. Headers are sequences of key value pairs, where keys are arbitrary strings and values are byte arrays that do not impose any specific encoding. Every key may use

appropriate encoding for the data that it relates to.

```
syntax = "proto3";

package headers;

message Headers {
    repeated Header headers = 1;
}

message Header {
  string key = 1;
  bytes value = 2;
}
```

On every stream initialization, the peer that creates it, is sending `Headers` message regardless if it contains header values or not. The receiving node must read this message and respond with response header using the same message type. This makes the header exchange sequence finished and any other stream data can be transmitted depending on the protocol.

Streams

Libp2p provides Streams as the basic channel for communication. Streams are full-duplex channels of bytes, multiplexed over a singe connection between two peers.

Every stream defines:

- a version that follows semantic versioning in semver form.
- data serialization definitions.
- sequence of data passing between peers over a full-duplex stream.

Streams are identified by libp2p case-sensitive protocol IDs. Swarm Bee uses the following convention to construct stream identifiers:

```
/ swarm /{ ProtocolName }/{ ProtocolVersion }/{ StreamName }
```

- All stream IDs are prefixed with `/swarm`.
- ProtocolName is a string in a free form that identifies the Swarm protocol.
- ProtocolVersion is a string in a semver form that is used to specify compatibility between protocol implementations over time.

— StreamName is a string in a free form that identifies a defined stream under the Swarm protocol.

Data passing sequence must be synchronous under one opened stream. Multiple streams can be opened at the same time that are multiplexed over the same connection exchanging data independently and asynchronously. Streams may use different data exchanging sequences, such as:

— single message sending not waiting for the response by the peer if it is not needed before closing the stream.
— multiple message sending a series of data that is sent to a peer without reading from it before closing the stream
— request/response requiring a single response for a single request before closing the stream
— multiple requests/response
— cycles requiring a synchronous response after every request before closing the stream exact message sequence requiring multiple message types over a single stream in an exact order, such as in the handshake protocol Streams must have a well predefined sequences, that are kept as simple as possible for a single purpose. For complex message exchanges, multiple streams should be used. Streams may be short lived for immediate data exchange or communication, or long lived for notifications if needed.

1.3 Bzz handshake protocol

Handshake protocol is the protocol that is always run after two peers are connected and before any other protocols are established. It communicated information about peer `Overlay` address, network `ID` and light node capability.

The handshake protocol defines only one stream:

— ID: `/swarm/handshake/1.0.0/handshake`
— Serialization: Varint delimited Protobuf

Message definition:

```
syntax = "proto3";

package handshake;

message Syn {
    bytes ObservedUnderlay = 1;
```

```
}

message Ack {
    BzzAddress Address = 1;
    uint64 NetworkID = 2;
    bool FullNode = 3;
    bytes Nonce = 4;
    string WelcomeMessage  = 99;
}

message SynAck {
    Syn Syn = 1;
    Ack Ack = 2;
}

message BzzAddress {
    bytes Underlay = 1;
    bytes Signature = 2;
    bytes Overlay = 3;
}
```

This message sequence is inspired by the TCP three way handshake to ensure message deliverability.

Upon connection a requesting peer constructs a new handshake stream and sends a `Syn` message with its `Overlay` address, `Network ID` and `Lightnode` capability flag, and waits for `SynAck` response message from the responding peer. It then sends its own `Syn` information and also `Ack` with the received `Overlay` address for confirmation by the requesting peer. After the requesting peer receives the `SynAck` message from the responding peer and validates that the received `Ack` information in it is correct, it sends the `Ack` message as a confirmation to the responding peer. The stream is closed by the responding peer after it receives the `Ack` message. If network IDs are not the same between to peers, the connection must be terminated during the `Syn - SynAck` exchange. Connection must be terminated if the handshake is performed between two peers multiple times over the same connection.

1.4 Hive protocol

The hive protocol enables nodes to get information about other peers that are relevant to them in order to bootstrap their connectivity. The information communicated are both overlay and underlay addresses of the known remote peers.

Hive

The hive protocol defines how nodes exchange information about their peers in order to reach and maintain a saturated Kademlia connectivity.

The exchange of this information happens upon connection, however nodes can broadcast newly received peers to their peers during the lifetime of connection.

The overlay address serves to select peers to achieve the connectivity pattern needed for the desired network topology, while the underlay address is needed to establish the peer connections by dialing selected peers.

Upon receiving a peer's message, nodes should store the peer information in their address book, i.e., a data structure containing info about peers known to the node that is meant to be persisted across sessions.

Upon request from downstream (connect) - the node will send the known peer addresses in batches of fixed size, until all are exhausted.

The process of sending the peer underlays is rate limited for a more fair distribution of resources among consumers of the protocol.

Hive protocol defines following streams:

— ID: `/swarm/hive/1.1.0/peers`
— Serialization: Varint delimited Protobuf

Message definitions:

```
syntax = "proto3";

package hive;

message Peers {
    repeated BzzAddress peers = 1;
}

message BzzAddress {
    bytes Underlay = 1;
    bytes Signature = 2;
    bytes Overlay = 3;
```

```
    bytes Nonce = 4;
}
```

When nodes are connected, they can request peers for the appropriate proximity bin, in order to achieve optimal saturation. This can be done in the beginning and/or during the lifetime of the connection, if needed (ex. when saturation of the node changes for the particular bin). This is done by sending `Peers` message over the `/swarm/hive/1.1.0/peers` stream, and receiving the list of known peers from the `Peers` message in the response.

During the lifetime of connection, nodes can broadcast newly found peers to their peers. This is done over `/swarm/hive/1.1.0/peers` notification by sending the `Peers` message with newly found or connected node.

All new (not known) peers found in `Peers` message, received either way, should be automatically broadcasted to all subscribed peers, in the same way already explained above.

Fetching peers using hive/peers stream

This is a request/response style communication that is happening over the `/swarm/hive/1.1.0/peers` stream. On each request, new stream is created with the appropriate id, and after the response is received, stream should be closed by both side.

Requesting side

Requesting node creates a stream and sends a `Peers` message, specifying the appropriate bin that it is interested in, and waits for the Peers message as response. The limit field can be used to limit the maximum number of peers in the response. The size `0` means that there is no limit. After the response is received, requesting node should close it's side of the stream, to let the other side know that response is read, and move on to processing the response. Each new peer (that is not previously known) received should be broadcasted to all subscribed peers.

Responding side

When the stream is created, responding node should wait for the `Peers` request from the requesting node, and send a `Peers` response, containing the appropriate Peers based on the request. After the response is sent, this node should wait for requesting node to close its side of the stream before closing the streaming and moving on.

Sending peers notification using hive/peers stream

This is a notification style communication that is happening over the `/swarm/hive/1.1.0/peers` notification stream. On each newly found or connected peer, node should send info about this peer to all subscribed peers.

Sending side

Stream with appropriate ID is created and the `Peers` message is sent over the stream. There is no response to this message. The sending node should wait for the receiving side to close its side of the stream before closing the stream and moving on.

Receiving side

When the stream is created, receiving node should wait for the `Peers` message. After receiving the message, node should close its side of the stream to let sender node that the message was received, and move on with processing. If the new node was not known, it should also be broadcasted to all subscribed peer. Nodes should keep track of peer info they already sent to other peers, or received it from, in order to avoid sending duplicate or even circular `Peers` notifications.

1.5 Retrieval

The retrieval of a chunk is a process which fetches a given chunk from the network by its address. Swarm involves a direct storage scheme of fixed size where chunks are stored on nodes with address corresponding to the chunk address. The retrieval protocols acts in such a way that it reaches those neighborhoods whenever a request is initiated. Such a route is sure to exist as a result of the Kademlia topology of keep-alive connections between peers.

The process of relaying the request from the initiator to the storer is called forwarding (pushsync) and also the process of passing the chunk data along the same path is called backwarding.

Conversely - Backwarding and Forwarding are both notions defined on a keep alive network of peers as strategies of reaching certain addresses.

If we zoom into a particular node in the retrieval path we see the following strategy:

— Receive the request

— Decide who to forward the request to (decision strategy)
— Have a way to match the the response to the original request

The key elements of the second step are:

— the strategy of choosing the peer to forward the request to
— how they react to failure like stream closure or nodes dropping offline or closing the protocol connection
— whether we proactively initiate several requests to peers

Retrieval protocol defines the following streams:

— ID: `/swarm/retrieval/1.4.0/retrieval`
— Serialization: Varint delimited Protobuf

Message definitions:

```
syntax = "proto3";

package retrieval;

message Request {
  bytes Addr = 1;
}

message Delivery {
  bytes Data = 1;
  bytes Stamp = 2;
  string Err = 3;
}
```

Requesting side

Requesting node creates a stream and sends a `Request` message, specifying the chunk address and waits for the `Delivery` response message. If the response message contains a non empty `Err` field the requesting node closes the stream and then can re-attempt retrieving the chunk from the next peer candidate.

Responding side

When the stream is created, the responding node should wait for the retrieval `Request` from the downstream. Once the request is received, the responding side will perform the steps below.

- Given a valid chunk address the first thing we do is lookup the chunk in the local store.
- If the chunk is not found locally we forward the request to the network.
- If we get the chunk from the network we might put it in the cache to avoid the extra cost in case the same address is requested repeatedly.
- If the chunk is not present or other errors have occurred in the process we return the unsuccessful response to the requester, otherwise we return the chunk with the associated stamp.

A successful response is a `Delivery` message with an empty error that will contain the chunk data and its associated stamp. After the response is sent, this node should wait for requesting node to close it's side of the stream before closing the streaming and moving on.

Retries and error cases

If the node that requested a chunk receives an error from the upstream it might retry the action unless it receives an explicit error code that the chunk has not been found. In this case it will moved on to the next peer candidate and repeat the request. If the failure reason is that the requesting peer is in overdraft, it will de-prioritize the corresponding upstream, giving it time to allow for balance replenishment. A 'backwarder' will give up after the first failure while an 'origin' node might repeat the request multiple time towards the same peer before giving up. A 'multiplexer' might attempt to fetch the chunk more than one time since being in the close reach of the neighborhood it has a higher chance of finding the chunk.

The retrieval of a chunk is a process which fetches a given chunk from the network by its address.

It follows the general semantics of the chunk syncing described above and follows the same network path as the push sync protocol, but in reverse.

1.6 Push-syncing

Pushsync protocol attempts to push a chunk to its destination neighborhood by selecting a peer (or peers) that is closer to it and further delegating the act of chunk transfer.

Every chunk is sure to have a destination neighborhood and is detected by comparing the overlay address of the peer and the chunk address. This operation returns the PO (proximity order). If this PO is greater or equal to the value of the storage radius then the chunk has reached its neighborhood.

That means that a chunk will travel through the network until it lands on a peer whose PO (between peer address and the chunk address) is greater or equal to the storage radius of that peer. The peer has the responsibility to store it in its reserve and sign a receipt with the peers signature and on the origin node it will confirm that proximity order between the storer node and chunk address is within the storage radius. If it's outside of the radius - we have a shallow receipts - and the chunk will be re-pushed eventually.

A chunk will be pushed multiple time if it's associated with more than one postage stamp. The receiving side is responsible to take this into account and persist the chunk accordingly.

Pushsync protocol defines the following streams:

— ID: `/swarm/pushsync/1.3.0/pushsync`
— Serialization: Varint delimited Protobuf

Message definitions:

```
syntax = "proto3";

package pushsync;

message Delivery {
  bytes Address = 1;
  bytes Data = 2;
  bytes Stamp = 3;
}

message Receipt {
  bytes Address = 1;
  bytes Signature = 2;
  bytes Nonce = 3;
  string Err = 4;
}
```

There are three roles that a peer can 'play': originator, forwarder and storer:

— originator is the node that is the first one to 'see' a new chunk

— forwarder is the node that takes the chunk and moves it closer to the storer
— storer is the node that has the responsibility to persist the chunk and to forward it to its neighborhood peers

Pushing side

An originator or an forwarder will create a new stream to which it will write a `Delivery` message containing the chunk data with its associated stamp. It will then wait for the response message. If the reply contains a non empty error, the pushing side will attempt pushing the chunk to the next best peer, as described bellow.

Storer side

Once the upstream peer receives the `Delivery` message and concludes that it is responsible to store the chunk it will perform the appropriate persistence interactions and will return a `Receipt` message. After the response is sent, this node should wait for requesting node to close it's side of the stream before closing the streaming and moving on.

Retries and error handling

If the node that pushed the chunk receives an error from the upstream it might retry the action multiple time if it's an origin node. If it exhaust its attempts to push the chunk to the closest peer it will move on to the next closest peer candidate and repeat the request. If the failure reason is that the pushing peer is in overdraft, it will de-prioritize the corresponding upstream, allowing for balance replenishment. A 'forwarder' will give up after the first failure while an 'origin' node might repeat the request multiple time towards the same peer before giving up. A 'multiplexer' might attempt to push the chunk more than one time since being in the close proximity to the neighborhood it has the confidence that the nodes in its reach are the ones responsible for persisting the chunk.

Further details

As a forwarder when we receive a chunk - if we're in reachable state and within the storage radius - we store the chunk to disk - and then we push the chunk to closest known peer.

If we are an origin peer we should not store the chunk initially so that the chunk is always forwarded into the network.

If no peer can be found from an origin peer, the origin peer may store the chunk.

Non-origin peers store the chunk if the chunk is within depth.

For non-origin peers, if the chunk is not within depth, they may store the chunk if they are the closest peer to the chunk.

In determining the closest peer we compare the proximity of the given peer with the chunk address. In order to determine if we act as a multiplexer and push the chunk in parallel to multiple peers we then compare the proximity value with the storage radius. We sent out the chunk to the neighborhoods in parallel (to maximum `2` peers at a time).

After pushing a chunk we await for reply containing a receipt. If the response comes back with an error we re-try the attempt. Once we exhaust the retries we return from the function with a `ErrNoPush` and cancel the context, thus stopping all ongoing go-routines.

— If that is true then the skipping of peers and selection of peers needs to be documented:
— 'choice strategy' - how kademlia table is under specifying as there's several nodes in the same proximity bin and you will need a secondary choice between them or some sort of multiplexing for selecting the optimal peer/s and it's not well documented:

The Kademlia component has a priority list on how to choose the closest peer:

— first it will decide if to include self (only full nodes who are not the origin node)
— lookup among peers who are reachable and healthy
— lookup among peers who are reachable
— lookup among all others

The default strategy for any downstream error is retry with a delay that is variable depending on how many 'in-flight' actions we have at any given moment.

Each time a chunk is being pushed to a peer - its (peer) address is added to skipList to avoid re-trying the same chunk/peer pair. The timeout is 5 min until the next retry. The peer address is added to the skiplist even if the attempt was successful - this is because the origin can notice that the receipt is too shallow, so the pusher can consider the attempt as unsuccessful.

Receiving an invalid chunk in the response will result in blocklisting of the peer that provided the chunk.

1.7 Pull-syncing

Pullsync is a subscription style protocol used to synchronize the chunks between neighborhood nodes. It bootstraps new nodes by filling up their storage with the chunks in range of their storage radius and also ensures eventual consistency - by making sure that the chunks will gradually migrate to their storer nodes.

Pullsync is done in parallel using multiple workers, one for each unique pair of peer/binID.

The chunks are served in batches (ordered by timestamp) and they cover contiguous ranges.

During pullsyncing the same chunk can be received multiple times if it has been stamped multiple times with different postage stamps. The pulling side must store it accordingly.

The downstream peers coordinate their syncing by requesting ranges from the upstream with the help of the "interval store" - to keep track of which ranges are left to be synchronized.

Because live syncing happens in sessions - it is inevitable that after a session is completed - the downstream peer disconnects and will be missing chunks that arrive later (after disconnect).

For this purpose the downstream peer will make a note about the timestamp of the last synced chunk on disconnect.

The point of the interval based approach is to cover those gaps that inevitably arise in between syncing sessions.

To save bandwidth, before the contents of the chunk is being sent over the wire, the upstream will sent a range of chunk addresses for approval. If the downstream decides that some (or all) addresses are desired - a confirmation message is sent to the upstream, to which it responds with the chunks mentioned in the request.

Pullsync protocol defines following streams:

- ID: `/swarm/pullsync/1.3.0/pullsync`
- ID: `/swarm/pullsync/1.3.0/cursors`
- Serialization: Varint delimited Protobuf

Message definitions:

```
syntax = "proto3";
```

```
package pullsync;

message Syn {}

message Ack {
  repeated uint64 Cursors = 1;
  uint64 Epoch = 2;
}

message Get {
  int32 Bin = 1;
  uint64 Start = 2;
}

message Chunk {
  bytes Address = 1;
  bytes BatchID = 2;
}

message Offer {
  uint64 Topmost = 1;
  repeated Chunk Chunks = 2;
}

message Want {
  bytes BitVector = 1;
}

message Delivery {
  bytes Address = 1;
  bytes Data = 2;
  bytes Stamp = 3;
}
```

Getting cursors

Initially it will request the cursors from the upstream by opening up a new `/swarm/pullsync/1.3.0/cursor` stream and sending a `Syn` message that includes a epoch timestamp. The expected response is the `Ack` message containing a collection of `int64` representing the cursor. The stream is closed on the requesting side.

To sync an interval the requesting peer sends a `Get` message to the upstream, message

containing the bin ID and the starting position.

Requesting the data

In response a `Offer` message is returned where the topmost index and a list of chunk addresses.

After inspecting the chunk addresses we issue a `Want` message with the addresses we're interested in fetching. The upstream responds with a series of `Deliver` messages containing the chunk data with their associated stamps. Once all requested chunks have been delivered the upstream will close the stream.

Error handling

If the upstream times out, returns an error or closes the stream unexpectedly it will be rescheduled to be synced at a later point in time.

During the first time we get the cursor from a new peer its 'epoch timestamp' will be persisted against the peer address. Epoch timestamp acts as the unique fingerprint of the peer's reserve. If the reserve is wiped out this will generate a new 'epoch timestamp'. In this way changes in epoch timestamp triggers a reset of the stored intervals for the given peer - forcing the pulling peer to start pullsync from scratch.

Also changes in Kademlia topology triggers a restart of pullsync - and if no changes happen in the span of a configurable amount of time (in Swarm it's 5 minutes) the pullsyncing will be restarted automatically by the scheduler after timeout.

If during pullsync an invalid chunk is received the upstream is blocklisted immediately.

Bootstrapping a node

A fresh node starts from the lowest bin ID which corresponds to the oldest chunks. Once done, the node continue with live syncing.

Live syncing implies that we've exhausted syncing from all lower bin IDs and reached a bin ID which does not have a chunk.

Interval store is a component that stores a list of intervals (start, end) with the last synced bin ID. Its purpose is to provide methods to `add` new intervals and retrieve missing intervals that need to be added.

It may be used in synchronization of streaming data to persist retrieved data ranges between sessions. There's a 'merge' functionality but is currently unused. In addition there is a 'last' method that returns the value that is at the end of the last interval.

Most important is the 'Next' method that returns the first range interval that is not fulfilled. Returned start and end values are both inclusive, meaning that the whole range including start and end need to be added in order to fill the gap in intervals.

Returned value for end is `0` if the next interval is after the whole range that is stored in Intervals. Zero end value represents no limit on the next interval length.

Returned empty boolean indicates if both start and end values have reached the ceiling value which means that the returned range is empty, not containing a single element.

Puller uses the storage radius as the indicator of the neighborhood, thus limiting the number of peers should be pulled from.

When syncing - we start at storage radius up to 31 (from the neighborhood).

For non-neighbours you only sync the bin that your peer is from (its equivalent).

This is only a safety mechanism that ensures that lost chunks are being eventually pull-synced to their destination by pulling from nodes that are in bins under the storage radius.

1.8 Pricing

Pricing protocol is used to announce payment threshold values. Nodes keep a price table for prices of every proximity order for each peer.

It defines the streams:

— ID: `/swarm/pricing/1.0.0/pricing`
— Serialization: Varint delimited Protobuf

Message definitions:

```
syntax = "proto3";

package pricing;

message AnnouncePaymentThreshold {
```

```
 bytes PaymentThreshold = 1;
}
```

Notifying side

When there's a need to upgrade the payment threshold the peer opens a stream and sends out a `AnnouncePaymentThreshold` with the new value. It then closes the stream.

Receiving side

The peer reads the message contents and stores the value locally against the notifying peer. If the value is below the minimum payment threshold the notifying peer is disconnected. After reading the message it closes the stream.

1.9 SWAP settlement protocol

The purpose of the settlement protocol is to exchange payments with other peers.

Settlement protocol defines the following streams:

- ID: `/swarm/pseudosettle/1.0.0/pseudosettle`
- Serialization: Varint delimited Protobuf

Message definitions:

```
syntax = "proto3";

package pseudosettle;

message Payment {
  bytes Amount = 1;
}

message PaymentAck {
  bytes Amount = 1;
  int64 Timestamp = 2;
}
```

Paying side

Paying side opens a stream to the peer it wants to send the payment and issues a `Payment` message. It then waits for the accepting side to respond and close the stream.

Accepting side

Reads the `Payment` message and after processing it returns a `PaymentAck` message containing the timestamp and outstanding debt for the paying side. It then closes the stream (or resets it - if any error has occurred).

Blocklisting

Blocklisting is the act of banning a certain peer from further interacting with us. It can be a temporary - for example due to accounting reasons, or permanent - for reasons such as returning an invalid chunk. Blocklisting a peer implies terminating any connections and disconnecting from it. A blocklisted peer will be immediately disconnected if it attempts to re-connect in the future. Blocklisting will happen automatically if a peer sends an invalid request based on the receiver's mode of operation, for example sending a pullsync request to a boot node.

1.10 Caching

Nodes can choose to store chunks that do not fall under area of responsibility in the event that the chunk belongs to some popular content. Additionally, during storage radius changes, the chunks evicted from the reserve are transferred into the cache. In Swarm the cache is a configurable parameter, individual peers can tune it to their needs. A larger cache size has the benefit of generating more revenue since the peer does not need to request the chunk from the network and avoids paying the fee for such action.

Garbage collection

Chunks leaves the reserve for two reasons, in both cases the chunk will go to cache after eviction:

— Expired postage stamp batch. Note: multiple batches can be associated with a single chunk.
— Reserve is full - we start evicting chunks (from the lowest value batch) starting from current radius and until the reserve size is below the capacity.

Part I

Appendix

Appendix A

Parameter constants

Table A.1 lists the constants used by Swarm with their type, default value and description.

CONSTANT NAME	TYPE	VALUE	DESCRIPTION
BZZ_NETWORK_ID	*uint64*	0	ID of the swarm network
PHASE_LENGTH	*uint64*	38	length of commit phase of the redistribution game round in number of blocks
ROUND_LENGTH	*uint64*	152	length of one redistribution game round in number of blocks, fixed at 4 times the PHASE_LENGTH
NODE_RESERVE_DEPTH	*uint8*	23	size requirement for client reserve capacity given in log number of chunks
SAMPLE_DEPTH	*uint8*	4	base 2 log number of chunks in sample
MAX_SAMPLE_VALUE	*uint256*	1.2844e+72	maximum value for last transformed address in reserve sample, i.e., $< 1\%$ chance the sampled set size is below a fourth of the prescribed node reserve size.
MINIMUM_STAKE	*uint256*	10	minimum stake amount in BZZ to be redefined as minimum stake given in storage rent units
MIN_STAKE_AGE	*uint256*	228	minimum number of blocks stakers need to wait after update or creation for the stake to be useable. Defaults to one and a half rounds to prevent opportunistic manipulation of stake after a neighbourhood is selected.

PRICE_2X_ROUNDS	*uint64*	64	number of rounds it takes for the price to double in the presence of a consistent lowest degree signal of undersupply.
NHOOD_PEER_COUNT	*uint8*	4	minimum number of nodes required to form a fully connected neighbourhood.

Table A.1: Parameter constants

Appendix B

Price oracle

The price oracle smart contract receives input from the redistribution game. Notably the input constitutes information on storage supply relative to the current demand. Since demand is maxed out with storage depth, this information is simply captured by the number of honest revealers per neighbourhood. Rounds when there are no revealers should be treated as rounds where the number of revealers is zero therefore the number of skipped rounds is passed to the price update function too.

Definition 1 – price function .
The *Price* function determines the unit price of storage (PLUR/chunk/block). It is defined in such a way that the ratio between the price of rent in consecutive rounds is an exponential function of supply. Supply is the deviation from the optimal number of peers in a neighbourhood as measured by the number of honest revealers in the previous round.

$$Price \quad : \quad \Gamma \mapsto uint256 \tag{B.1}$$

$$Price(\gamma) \quad \stackrel{\text{def}}{=} \quad Price(Prev(\gamma)) \cdot 2^{\sigma \cdot d} \tag{B.2}$$

where

$$\text{SUPPLY} \qquad d = \texttt{NHOOD\_PEER\_COUNT} - |Reveals(\gamma)| \tag{B.3}$$

$$\text{REACTIVITY} \qquad \sigma = \frac{1}{\texttt{PRICE\_2X\_ROUNDS}} \tag{B.4}$$

First, note that for a consistent deviation signal d, the price change after n rounds is given by multiplying the starting price with $\left(2^{\sigma \cdot d}\right)^n$. The price has doubled if this expression is 2, i.e., $n \cdot d \cdot \sigma = 1$. With the lowest signal of undersupply ($r = 3$), $d = 1$, and therefore the reactivity parameter is expressed as $\sigma = \frac{1}{n}$. In other words $\frac{1}{\sigma}$ measures the number of rounds it takes for the price to double.

Appendix C

Source of randomness

As the *neighbourhood selection anchor* will directly affect which neighbourhood wins the pot, it is prudent to derive the randomness from a source of entropy that cannot be manipulated. A common solution to obtain randomness is to have independent parties committing to a random nonce with a stake. The random seed for a round is defined as the xor of all revealed nonces. Given the nonces are independent sources fixed in the commit, no individual participant has the ability to skew randomness by selecting a particular nonce. Thanks to the commutativity of xor, the order of reveals is also irrelevant. However, if the reveal transactions are sequential, committers compete at holding out since the last one to reveal can effectively choose the resulting seed to be either including its committed nonce (if they do reveal) or not (if they do not reveal). The threat to slash the stake of non-revealers serves to eliminate this degree of freedom from last revealers and thus renders this scheme a secure random oracle assuming there is at least one honest (non-colluding) party.[1]

Now note that the redistribution scheme already has a commit reveal scheme as well as stake slashed for non-revealers, so a potential random oracle is already part of the proposed scheme. Incidentally, the beginning of the claim phase is when new randomness is needed to select the truth and a winner. Importantly, these random values are only needed if there is a claim which implies that there were some commits and reveals to choose from. Or conversely, if there are no reveals in the round,[2] the random seed is undefined but is also not needed for the claim.

The random seed that transpires at the beginning of the claim phase can serve as the *reserve*

[1] If the stake is higher than the reward pot, one cannot afford being slashed with even just one commit without a loss. If this cannot be guaranteed, slashing of the stake is not an effective deterrent.

[2] If saboteurs get slashed or frozen in the claim transaction, if there is no claim, the committers get away without being punished. This can be remedied if the staking contract keeps a flag on each overlay (set when commits, unsets when reveals in the same round) and the check and punishment happens as a result of a commit call in the case the flag is found set.

sample salt (nonce input to modified hash used in the sampling) with which the nodes in the selected neighbourhood can start calculating their reserve sample.

The neighbourhood for the next round is selected by the *neighbourhood selection anchor*, which is, similarly to the truth and winner selection nonces, deterministically derived from the same random seed. Unlike the nonces used to select from the reveals, neighbourhood selection should be well defined for the following round even if a round is skipped, i.e., when there is no reveals. To cover this case skipped rounds keep the random seed of the previous round. However, in order to rotate selected neighbourhoods through skipped rounds, we derive the neighbourhood selection anchor from the seed by factoring in the number of game rounds passed since the last reveal.[3]

In order to provide protection against the case when each committer in the neighbourhood is colluding, and can afford losing stake we need to make sure that the entropy is still high otherwise the nodes can influence the neighbourhood selection nonce and reselect themselves or a fixed colluding neighbourhood (or increase the chances of reselection).

Definition 2 – Random seed for the round.
Define the random seed of the round as the xor of all obfuscation keys sent as part of the reveal transaction data during the entire reveal period:

$$\mathcal{R} \quad : \quad \Gamma \mapsto \mathit{Nonce} \tag{C.1}$$

$$\mathcal{R}(\gamma) \quad \stackrel{\text{def}}{=} \quad \begin{cases} \mathcal{R}(\mathit{Prev}(\gamma)) & \text{if } |\mathit{Reveals}(\gamma)| = 0 \\ \bigvee_{r \in \mathit{Reveals}(\gamma)} \text{NONCE}(r) & \text{otherwise} \end{cases} \tag{C.2}$$

Lemma 3 – Round seed is a secure random oracle.
The nonce produced by xoring the revealed obfuscation keys is a correct source of entropy.

Proof. Assuming n independent parties committing, choosing any particular nonce will leave the outcome fully random.

[3]Otherwise a selected neighbourhood could collude maliciously not to commit/reveal and have the pot roll over to the following round. By simply holding out for a number of redistribution rounds, they could unfairly multiply their reward when they eventually claim the pot.

Appendix D

Density-based size estimation

Nodes in Swarm must utilise their full reserve capacity: nodes will potentially further replicate chunks in case they have unused reserve capacity beyond storing their share necessary for a system-wide level of redundancy required. To incentivise this, participating in the redistribution game involves a check called *proof of resources* which is supposed to verify the size of reserve from which the reserve samples are generated. The insight here is that the sample is the lowest range of a uniformly random variate over the entire 256-bit address space. Intuitively, the higher the original volume of the sampled set, the denser it is, the lower the expected maximum value in the sample. Conversely, a constraint on the maximum value of the last element in the sample practically puts a minimum cardinality requirement on the sampled set using a solution called *density based set size estimation*.

We are given n independent, uniformly distributed values between 0 and 1.[1] Let the value of the kth smallest of these be x_k (so the smallest of the n values is x_1, the second smallest is x_2, and so on, up to x_n). What is the distribution of x_k, given n? And what is the threshold value u such that for any given probability α, the chance of obtaining an x_k lower than u is α?

Note that the distance between any two adjacent values out of n independent uniform variates follows an exponential distribution, as long as n is sufficiently large.[2] The rate parameter of this exponential distribution is $n + 1$, where n is increased by one to account for the fact

[1] Because we work with densities, the actual integer range is not relevant and results obtained for the unit interval can simply be rescaled to the Swarm use case by multiplying with 2^{256}.

[2] This follows from the fact that n independent uniform variates can be thought of as realizing a Poisson process, whereby the timing of events is random, and it is known that the nearest-neighbour distribution (i.e., waiting time between two consecutive events) is then exponentially distributed.

that the expected gap between adjacent values is $1/(n+1)$.[3]

We are after the distribution of the kth value, [4] x_k, which then can be thought of as arising from the sum of k independent exponential variables, each with a rate parameter of $n+1$. This is known to result in an Erlang distribution with shape parameter k and rate parameter $n+1$. Using $X(\lambda)$ to denote the exponential distribution with rate λ and $E(k,\lambda)$ to denote the Erlang distribution with shape k and rate λ:

$$\sum_{i=1}^{k} X_i(n+1) = E(k, n+1), \tag{D.1}$$

where the subscript i in $X_i(n+1)$ distinguishes between independent exponentially distributed random variables. The probability density function $E(x,k,n+1)$ of the Erlang distribution itself is given by

$$E(x,k,n+1) = \frac{(n+1)^k x^{k-1} \mathrm{e}^{-(n+1)x}}{(k-1)!}. \tag{D.2}$$

This distribution contains the answer to the first question: what is the distribution of the kth smallest value out of n independent uniform variates between 0 and 1? For example, if $k = 16$ and n is either 500, 750, or 1000, we get the distributions shown in Figure D.1.

We can now answer the second question: given n and a confidence level α, what is the threshold value u for x_k such that the probability that $x_k < u$ is equal to α? That is, we wish to know the value $x = u$ at which the probability distribution has encompassed a given area of α (see figure D.2).

The area under the curve of the Erlang distribution is given by its cumulative distribution function $P(x,k,n+1)$, which is known to be

$$P(x,k,n+1) = \int_0^x E(y,k,n+1)\,\mathrm{d}y = \frac{1}{(k-1)!}\int_0^{(n+1)x} t^{k-1}\mathrm{e}^{-t}\,\mathrm{d}t. \tag{D.3}$$

The latter expression is sometimes written as $\tilde{\gamma}(k,(n+1)x)$, where

$$\tilde{\gamma}(k,x) = \frac{1}{\Gamma(k)}\int_0^x t^{k-1}\mathrm{e}^{-t}\,\mathrm{d}t \tag{D.4}$$

[3] For $n = 2$, the mean outcome is $x_1 = 1/3$ and $x_2 = 2/3$; for $n = 3$, it is $x_1 = 1/4$, $x_2 = 2/4$, $x_3 = 3/4$; and so on: for arbitrary n, $x_i = i/(n+1)$, with the gap between adjacent values in this ideal case always being $1/(n+1)$.

[4] An alternative approach using order statistic expresses x_k via a beta distribution. It is very difficult to prove that the Beta distribution's quantile function is a strictly decreasing function of n, which is a key piece of the argument presented here. Although this method is exact even for small n, in our case, n always a very large number, therefore we adopted the other method.

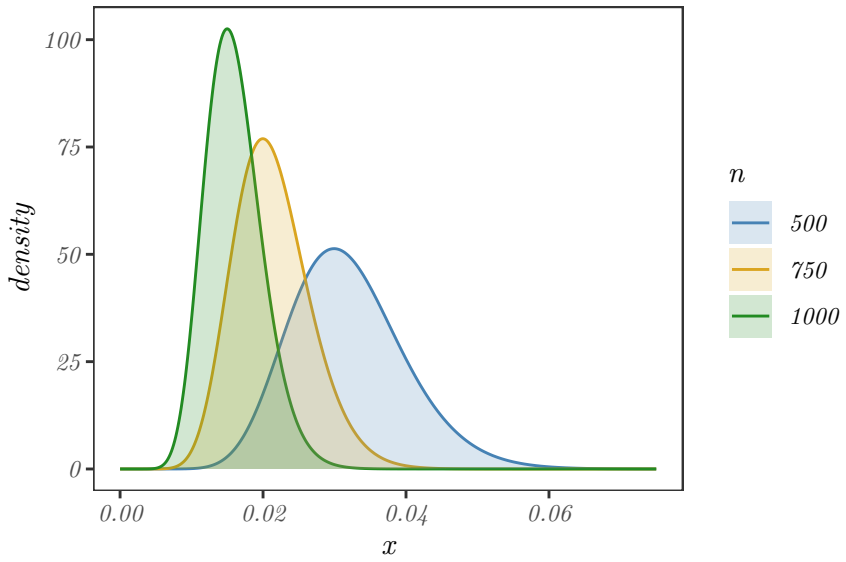

Figure D.1: Probability density function $E(x, k, n+1)$ of the Erlang distribution, with $k = 16$ and n either 500, 750, or 1000.

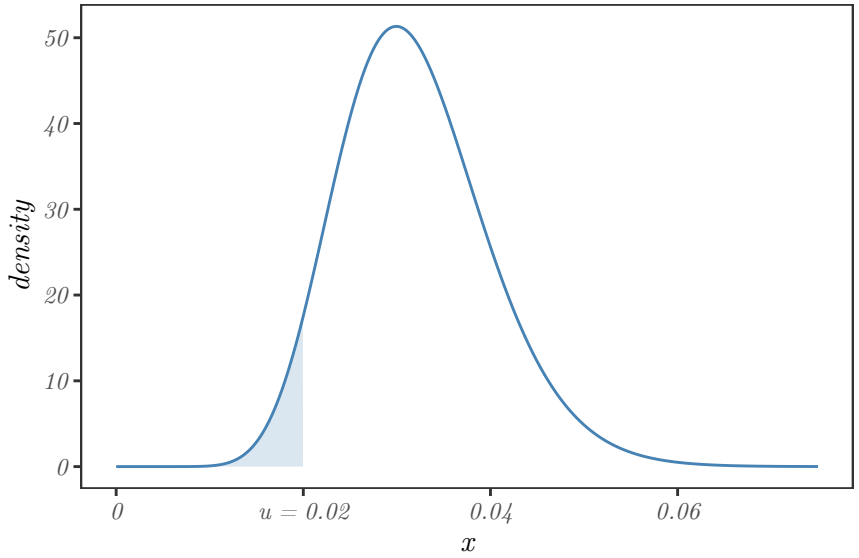

Figure D.2: The distribution of x_{16}, i.e., the 16th smallest value from among $n = 500$ independently and uniformly drawn variates between 0 and 1. The area under the curve is shaded up to 5% of its area. The point at which the shading stops is therefore the value u for which there is only a 5% chance of getting an even smaller x_{16}.

is the regularized lower incomplete gamma function. We therefore want to solve the equation

$$\alpha = \int_0^u E(y, k, n+1)\,\mathrm{d}y = P(u, k, n+1) \tag{D.5}$$

for u.

Inverting this expression in u (since the cumulative distribution function increases monotonically in u, the inverse exists) leads to the quantile function $Q(\alpha, k, n+1)$ of the Erlang distribution: $u = Q(\alpha, k, n+1)$. The quantile function is known to be expressible as

$$Q(\alpha, k, n+1) = \frac{\tilde{\gamma}^{-1}(k, x)}{n+1}, \tag{D.6}$$

where $\tilde{\gamma}^{-1}(k, x)$ is the inverse regularized lower incomplete gamma function. Its particular form is of no interest to us, except for two properties. First, it is positive for all x.[5] Second, it is independent of n. Instead, the entire dependence of $Q(\alpha, k, n+1)$ on n is given by the $n+1$ term in the denominator of Equation D.6. From this, we conclude that $Q(\alpha, k, n+1)$ is a strictly decreasing function of n.

These two points lead to an important consequence. Say we compute the threshold u for a given α and n in order to have an upper bound on a lower quantile. Now, if we were to decrease n but hold all other things equal, the threshold will always get higher than what it was before. The threshold obtained for higher values of n may therefore serve as a conservative estimate of the threshold for lower values: if u is a threshold such that the kth smallest out of n uniform variates is only smaller than u in α of cases, then for any amount $m < n$, the chance of the kth variate conforming to the same constraint (i.e., $x_k < u$) is now even smaller than α.

Conversely, if we were to constrain x_k so that the probability of not getting a value smaller than u is lower than β (minimising a higher quantile), we find that the constraint remains true as n is increased.

Armed with these results, let us see how Equation D.6 can be used for the estimation procedure. There are two problems to tackle, ultimately relating to the two aspects of a test's accuracy. First, we want to catch inadequate storers slacking on volume. In other words, we want to constrain the x_k values so that we can safely say that any attacker with a stored volume below an acceptable size n has a probability less than α to obtain such a small x_k by pure chance. Construing the condition for $x_k < u$ as a test to filter honest players (just based on the size of their reserve), $1 - \alpha$ expresses the *sensitivity* of the test. From the previous argument on the monotonic dependence of α on n, it is safe to use a condition that requires x_k to stay below a threshold obtained for n.

Second, we want to avoid situations when honest participants end up not satisfying the above constraint even though they sampled from a set larger than the required minimum. Given a target volume $m > n$, the error rate of false negatives is guaranteed to be less than

[5] This stands to reason: the quantile function of a distribution on $x \in [0, \infty)$ is itself between 0 and ∞, and $\tilde{\gamma}^{-1}(k, x)$ is just the quantile function of the Erlang distribution times the positive constant $n+1$.

β obtained from the quantile function with parameters $m, k,$ and u. The quantity $1 - \beta$ is the *specificity* or *precision* of the test.

Figure D.3 illustrates this idea, for two different distributions in both the lower- and upper-end estimation. What we want is to choose u to simultaneously make sure that dishonest players do not sneak through the system *and* also that honest players do not get excluded too often. This translates to make both α and β as small as possible.

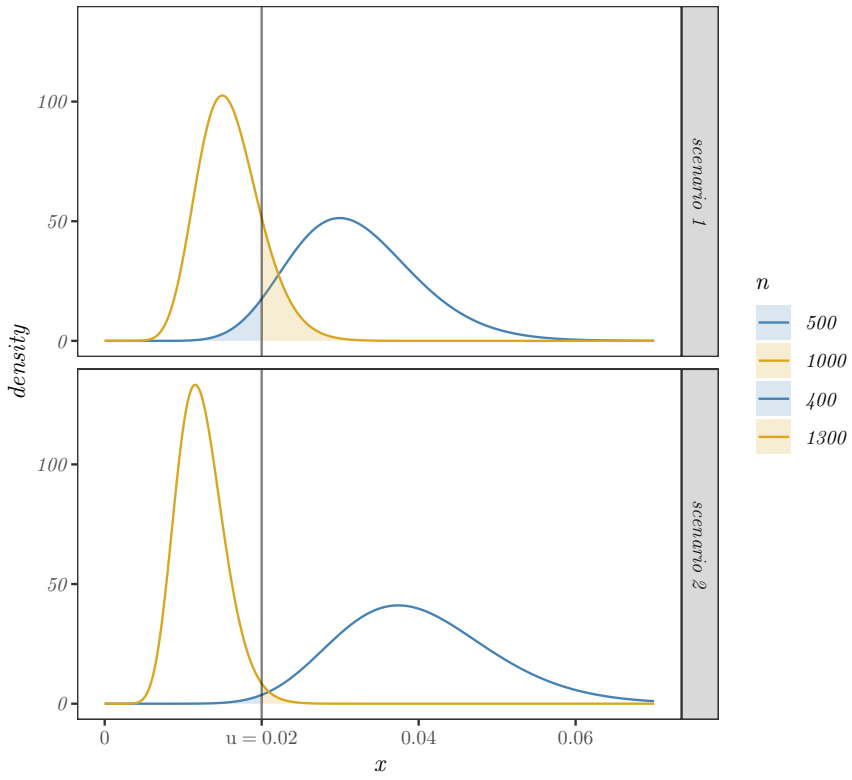

Figure D.3: Recall and precision of proof of reserve size validation: Any chosen u will lead to different α and β values, depending on n. Here u is fixed at 0.02. The top panel shows distributions for $x \equiv x_{16}$ with $n = 500$ (blue) and $n = 1000$ (yellow). The area left under the blue curve to the left of u is equal to α (blue shade); the area under the yellow curve right of u is equal to β (yellow shade). If the curves overlap considerably (top), it is impossible to choose an u such that α and β are simultaneously small.

One way to try and find the best compromise is by minimising $\alpha + \beta$ (the *accuracy* of the test) and pick the u value at the optimum to be used in the proof of resources test. To this end, one can vary α between 0 and 1 and, for each of its values, solve the equation $\alpha = Q(1 - \beta, k, n + 1)$ (where n is the larger value, used for estimating β). This way, we get a β value for every possible α. Then, we can find the combination which minimises $\alpha + \beta$, and determine the value of u that leads to this optimum. As illustrated in figure D.4, larger values of k yield a trade-off curve along which better accuracies can be achieved.

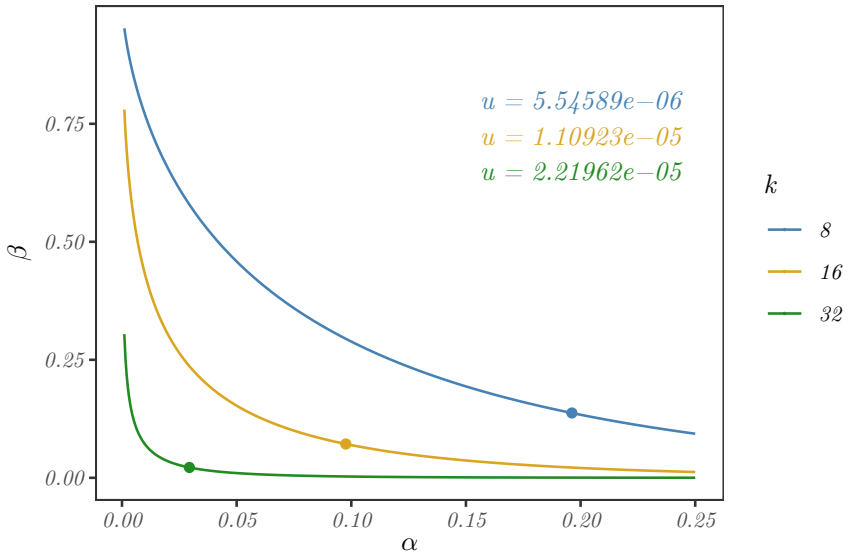

Figure D.4: Optimal accuracy of reserve size probe By increasing k, one can get better optima for minimising $\alpha + \beta$. Here $n = 10^6$ for estimating α and $2 \cdot 10^6$ for estimating β, and k is either 8, 16, or 32 (colors). The values of u associated with the optima are also shown.

Table D.1 summarizes the important numerical results in Figure D.4.[6]

k	α	β	u	$u \cdot 2^{256}$
8	0.196216	0.1373622	$5.54589 \cdot 10^{-6}$	$6.421705 \cdot 10^{71}$
16	0.097612	0.0716570	$1.10923 \cdot 10^{-5}$	$1.284401 \cdot 10^{72}$
32	0.029386	0.0219151	$2.21962 \cdot 10^{-5}$	$2.570140 \cdot 10^{72}$

Table D.1: Proof of density parameter calibration. Assuming $n = 10^6$ and $m = 2 \cdot 10^6$ to calculate recall and precision error rates α and β, respectively, the cutoff value for the proof is calibrated by optimizing on acccuracy using sample sizes $8, 16, 32$.

[6]Since the hash function used to generate random variates does so in the range $[0, 2^{256} - 1]$ instead of $[0, 1]$, the calculated thresholds are scaled with 2^{256} to show where they would fall in their actual range.