Transport layer

Underlay network

Libp2p networking stack provides all required properties for 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.

Where 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`
- 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/3110`
- 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.

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.

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.

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;
}
```

Message sequence is replicated from TCP three way handshake to ensure message deliverability.

Upon connection a requesting peer construct a new handshake stream and sends the `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.

Hive protocol

Hive protocol enables nodes to get information about other peers that are relevant to them. The information communicated are both overlay and underlay addresses of the known remote peers.

This information is needed 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 peers 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.

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

Pushsync

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.

Pullsync

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/cursors` 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.

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.

Settlement

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.

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.

Glossary and definitions

- `reserve` is the space (fixed capacity 2^22) allocated for the chunks that fall under a nodes storage radius
- `proximity order` the number of similar bits in the address (from the left)
- `storage radius` (storage depth) defines the range of chunk addresses a node has the responsibility to store where the proximity order of the chunk addresses must be greater (or equal) to the storage radius.
- `neighborhood` constitutes a collection of nodes whose overlay addresses have a proximity order greater or equal to the value of the storage radius.
- `multiplexing` the act of forwarding (or backwarding) to multiple nodes when the forwarding peer is in immediate proximity to the neighborhood.