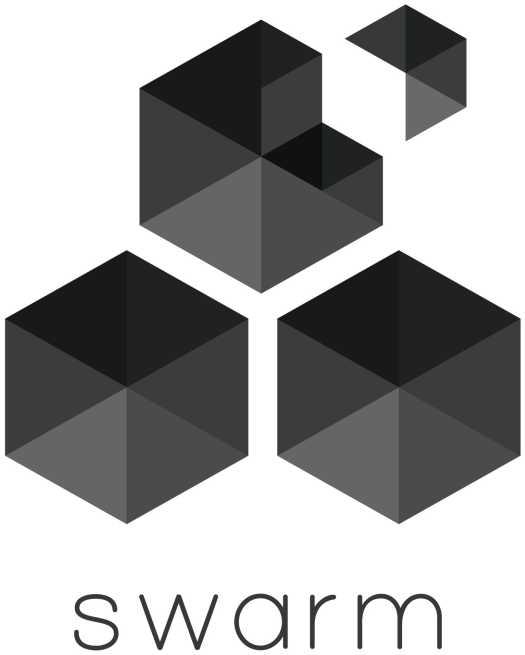


swarm documentation

Release 0.5

viktor trón, áron fischer, nick johnson, dániel a. nagy, zsolt felfö

Jul 01, 2020

Contents

1 Implementations of Swarm **3**

2 Public Swarm Networks **5**

3 Development status **7**
3.1 Introduction . . . 7
3.1.1 Implementations of Swarm . . . 8
3.1.2 Public Swarm Networks . . . 8
3.1.3 Development status . . . 8
3.2 Architectural Overview . . . 8
3.2.1 Preface . . . 8
3.2.2 Overlay network . . . 10
3.2.2.1 Logarithmic distance . . . 10
3.2.2.2 Kademlia topology . . . 10
3.2.2.3 Bootstrapping and discovery . . . 11
3.2.3 Distributed preimage archive . . . 11
3.2.3.1 Redundancy . . . 12
3.2.3.2 Caching and purging Storage . . . 12
3.2.3.3 Synchronisation . . . 13
3.2.4 Data layer . . . 13
3.2.4.1 Files . . . 13
3.2.4.2 Manifests . . . 14
3.2.5 Components . . . 15
3.2.5.1 Swarm Hash . . . 15
3.2.5.2 Chunker . . . 15
3.3 Incentives system . . . 16
3.3.1 Cryptoeconomics . . . 16
3.3.2 Accounting . . . 16
3.3.2.1 Thresholds . . . 17
3.3.2.2 Fraud risks . . . 17
3.3.2.3 Imbalances . . . 18
3.3.2.4 Settlement with cheques . . . 18
3.3.2.5 Interaction with the blockchain . . . 18
3.3.2.6 Smart contracts . . . 19
3.3.2.7 Metrics . . . 19
3.3.3 Honey Token . . . 20
3.3.4 PricedMessage . . . 20
3.3.5 Spam protection: Postage Stamps . . . 20
3.3.6 Multiple blockchains . . . 20
3.4 Swarm Specification Process . . . 21
3.5 Swarm for DApp-Developers . . . 21
3.5.1 Public gateways . . . 21
3.5.2 Uploading and downloading . . . 21
3.5.2.1 Introduction . . . 21
3.5.2.2 Using HTTP . . . 22
3.5.2.3 Using the CLI - Command Line Interface . . . 24
3.5.3 Example Dapps . . . 30

3.5.4 Working with content . . . 30
3.5.4.1 Ethereum Name System (ENS) . . . 30
3.5.4.2 Feeds . . . 31
3.5.4.3 Go API . . . 33
3.5.4.4 JavaScript example . . . 37
3.5.4.5 Manifests . . . 38
3.5.4.6 Encryption . . . 42
3.5.4.7 Access Control . . . 43
3.5.4.8 FUSE . . . 48
3.5.4.9 Pinning Content . . . 50
3.5.4.10 Tags . . . 51
3.5.4.11 BZZ URL schemes . . . 52
3.5.5 Swarm Messaging for DAPP-Developers . . . 56
3.5.5.1 PSS Usage . . . 56
3.5.6 API reference . . . 58
3.5.6.1 HTTP . . . 58
3.5.6.2 JavaScript . . . 59
3.5.6.3 RPC . . . 60
3.6 Swarm for Node-Operators . . . 61
3.6.1 Installation and Updates . . . 61
3.6.2 Download pre-compiled Swarm binaries . . . 61
3.6.3 Setting up Swarm in Docker . . . 61
3.6.4 Installing Swarm from source . . . 62
3.6.4.1 Prerequisites: Go and Git . . . 62
3.6.4.2 Configuring the Go environment . . . 62
3.6.4.3 Download and install Geth . . . 63
3.6.4.4 Compiling and installing Swarm . . . 63
3.6.4.5 Updating your client . . . 63
3.6.5 Running the Swarm Go-Client . . . 64
3.6.5.1 Verifying that your local Swarm node is running . . . 64
3.6.6 Interacting with Swarm . . . 64
3.6.7 How do I enable ENS name resolution? . . . 65
3.6.7.1 Using Swarm together with the testnet ENS . . . 66
3.6.7.2 Using an external ENS source . . . 66
3.6.8 Connect to the SWAP-enabled testnet . . . 67
3.6.8.1 Prerequisites . . . 67
3.6.8.2 Network setup . . . 67
3.6.8.3 Bootnodes . . . 67
3.6.8.4 Run your own swap-enabled node and connect to the cluster . . . 67
3.6.9 Running a SWAP-enabled node on an alternative blockchain . . . 69
3.6.10 Alternative modes . . . 69
3.6.10.1 Swarm in singleton mode (no peers) . . . 69
3.6.10.2 Adding enodes manually . . . 69
3.6.10.3 Connecting to the public Swarm cluster . . . 70
3.6.10.4 Metrics reporting . . . 70
3.6.11 Go-Client Command line options Configuration . . . 71
3.6.12 Config File . . . 71
3.6.13 General configuration parameters . . . 72
3.7 Swarm for Go-Client Contributors . . . 73
3.7.1 Introduction . . . 73
3.7.2 Reporting a bug and contributing . . . 74
3.7.3 High level component description . . . 74
3.7.3.1 Interfaces to Swarm . . . 74
3.7.3.2 Structural components and key processes . . . 75
3.7.3.3 Storage module . . . 75
3.7.3.4 Communication layer . . . 75
3.7.4 Simulation Framework . . . 75

Index 77

Important: New Bee client

In the effort to release a production-ready version of Swarm, the Swarm dev team has migrated their effort to build the new Bee client, a brand-new implementation of Swarm. The main reason for this switch was the availability of a more mature networking layer (libp2p) and the secondary reason being that the insight gained from developing Swarm taught us many lessons which can be implemented best from scratch. While Bee is currently not exposing all features you got used to in Swarm, the development is happening at lightspeed and soon, it will surpass Swarm in functionality and stability!

Please refer to Swarm webpage for more information about the state of Bee client and to Bee documentation for documentation.

Old Swarm client

Old Swarm client, described by this documentation, can still be used until the network exists, however no maintenance or upgrades are planned for it.

Compatibility of Bee with the first Swarm

Ethereum Swarm Bee is the second official Ethereum Swarm implementation. No compatibility on the network layer with the first Ethereum Swarm implementation can be provided, mainly because the change in underlying network protocol from devp2p to libp2p. This means that a Bee node cannot join first Swarm network and vice versa. Migrating data is possible, please refer to Bee documentation.

Swarm is a censorship resistant, permissionless, decentralised storage and communication infrastructure.

This base-layer infrastructure provides these services by contributing resources to each other. These contributions are accurately accounted for on a peer to peer basis, allowing nodes to trade resource for resource, but offering monetary compensation to nodes consuming less than they serve.

Swarm is using existing Smart-Contract platforms (e.g. Ethereum) to implement the financial incentivisation.

Chapter 1

Implementations of Swarm

Client-Name	Programming Language	Maintained by	Notes
Swarm Go-Client	golang	Swarm Core-Team	
Swarm Nim-Client	nim	Status.im	Development is planned to start Q1 2020

Chapter 2

Public Swarm Networks

The Ethereum Foundation operates a Swarm testnet that can be used to test out functionality in a similar manner to the Ethereum testnetwork. Everyone can join the network by running the Swarm client.

Chapter 3

Development status

Uploaded content is **not guaranteed to persist on the testnet** until storage insurance is implemented. All participating nodes should consider participation a voluntary service with no formal obligation whatsoever and should be expected to delete content at their will. Therefore, users should **under no circumstances regard Swarm as safe storage** until the incentive system is functional.

Important: New Bee client

In the effort to release a production-ready version of Swarm, the Swarm dev team has migrated their effort to build the new Bee client, a brand-new implementation of Swarm. The main reason for this switch was the availability of a more mature networking layer (libp2p) and the secondary reason being that the insight gained from developing Swarm taught us many lessons which can be implemented best from scratch. While Bee is currently not exposing all features you got used to in Swarm, the development is happening at lightspeed and soon, it will surpass Swarm in functionality and stability!

Please refer to Swarm webpage for more information about the state of Bee client and to Bee documentation for documentation.

Old Swarm client

Old Swarm client, described by this documentation, can still be used until the network exists, however no maintenance or upgrades are planned for it.

Compatibility of Bee with the first Swarm

Ethereum Swarm Bee is the second official Ethereum Swarm implementation. No compatibility on the network layer with the first Ethereum Swarm implementation can be provided, mainly because the change in underlying network protocol from devp2p to libp2p. This means that a Bee node cannot join first Swarm network and vice versa. Migrating data is possible, please refer to Bee documentation.

3.1 Introduction

Swarm is a censorship resistant, permissionless, decentralised storage and communication infrastructure.

This base-layer infrastructure provides these services by contributing resources to each other. These contributions are accurately accounted for on a peer to peer basis, allowing nodes to trade resource for resource, but offering monetary compensation to nodes consuming less than they serve.

Swarm is using existing Smart-Contract platforms (e.g. Ethereum) to implement the financial incentivisation.

3.1.1 Implementations of Swarm

Client-Name	Programming Language	Maintained by	Notes
Swarm Go-Client	golang	Swarm Core-Team	
Swarm Nim-Client	nim	Status.im	Development is planned to start Q1 2020

3.1.2 Public Swarm Networks

The Ethereum Foundation operates a Swarm testnet that can be used to test out functionality in a similar manner to the Ethereum testnetwork. Everyone can join the network by running the Swarm client.

3.1.3 Development status

Uploaded content is **not guaranteed to persist on the testnet** until storage insurance is implemented. All participating nodes should consider participation a voluntary service with no formal obligation whatsoever and should be expected to delete content at their will. Therefore, users should **under no circumstances regard Swarm as safe storage** until the incentive system is functional.

3.2 Architectural Overview

3.2.1 Preface

Swarm defines 3 crucial notions:

chunk Chunks are pieces of data of limited size (max 4K), the basic unit of storage and retrieval in the Swarm. The network layer only knows about chunks and has no notion of file or collection.

reference A reference is a unique identifier of a file that allows clients to retrieve and access the content. For unencrypted content the file reference is the cryptographic hash of the data and serves as its content address. This hash reference is a 32 byte hash, which is serialised with 64 hex bytes. In case of an encrypted file the reference has two equal-length components: the first 32 bytes are the content address of the encrypted asset, while the second 32 bytes are the decryption key, altogether 64 bytes, serialised as 128 hex bytes.

manifest A manifest is a data structure describing file collections; they specify paths and corresponding content hashes allowing for URL based content retrieval. The BZZ URL schemes assumes that the content referenced in the domain is a manifest and renders the content entry whose path matches the one in the request path. Manifests can also be mapped to a filesystem directory tree, which allows for uploading and downloading directories. Finally, manifests can also be considered indexes, so it can be used to implement a simple key-value store, or alternatively, a database index. This offers the functionality of *virtual hosting*, storing entire directories, web3

websites or primitive data structures; analogous to web2.0, with centralized hosting taken out of the equation.

In this guide, content is understood very broadly in a technical sense denoting any blob of data. Swarm defines a specific identifier for a file. This identifier part of the reference serves as the retrieval address for the content. This address needs to be

- collision free (two different blobs of data will never map to the same identifier)
- deterministic (same content will always receive the same identifier)
- uniformly distributed

The choice of identifier in Swarm is the hierarchical Swarm hash described in *Swarm Hash*. The properties above allow us to view hashes as addresses at which content is expected to be found. Since hashes can be assumed to be collision free, they are bound to one specific version of a content. Hash addressing is therefore immutable in the strong sense that you cannot even express mutable content: "changing the content changes the hash".

Users of the web, however, are accustomed to mutable resources, looking up domains and expect to see the most up to date version of the 'site'. Mutable resources are made possible by the ethereum name service (ENS) and Feeds. The ENS is a smart contract on the ethereum blockchain which enables domain owners to register a content reference to their domain. Using ENS for domain name resolution, the url scheme provides content retrieval based on mnemonic (or branded) names, much like the DNS of the world wide web, but without servers. Feeds is an off-chain solution for communicating updates to a resource, it offers cheaper and faster updates than ENS, yet the updates can be consolidated on ENS by any third party willing to pay for the transaction.

Just as content in Swarm is addressed with a 32-byte hash, so is every Swarm node in the network associated with a 32-byte hash address. All Swarm nodes have their own *base address* which is derived as the (Keccak 256bit SHA3) hash of the public key of an ethereum account:

Note: *Swarm node address = sha3(ethereum account public key)* - the so called *swarm base account* of the node. These node addresses define a location in the same address space as the data.

When content is uploaded to Swarm it is chopped up into pieces called chunks. Each chunk is accessed at the address deterministically derived from its content (using the chunk hash). The references of data chunks are themselves packaged into a chunk which in turn has its own hash. In this way the content gets mapped into a merkle tree. This hierarchical Swarm hash construct allows for merkle proofs for chunks within a piece of content, thus providing Swarm with integrity protected random access into (large) files (allowing for instance skipping safely in a streaming video or looking up a key in a database file).

Swarm implements a *distributed preimage archive*, which is essentially a specific type of content addressed distributed hash table, where the node(s) closest to the address of a chunk do not only serve information about the content but actually host the data.

The viability of both hinges on the assumption that any node (uploader/requester) can 'reach' any other node (storer). This assumption is guaranteed with a special *network topology* (called *kademlia*), which guarantees the existence as well a maximum number of forwarding hops logarithmic in network size.

Note: There is no such thing as delete/remove in Swarm. Once data is uploaded there is no way to revoke it.

Nodes cache content that they pass on at retrieval, resulting in an auto scaling elastic cloud: popular (oft-accessed) content is replicated throughout the network decreasing its retrieval latency. Caching also results in a *maximum resource utilisation* in as much as nodes will fill their dedicated storage space with data passing through them. If capacity is reached, least accessed chunks are purged by a garbage

collection process. As a consequence, unpopular content will end up getting deleted. Storage insurance (yet to be implemented) will offer users a secure guarantee to protect important content from being purged.

3.2.2 Overlay network

3.2.2.1 Logarithmic distance

The distance metric $MSB(x, y)$ of two equal length byte sequences x an y is the value of the binary integer cast of $xXORy$ (bitwise xor). The binary cast is big endian: most significant bit first (=MSB).

$Proximity(x, y)$ is a discrete logarithmic scaling of the MSB distance. It is defined as the reverse rank of the integer part of the base 2 logarithm of the distance. It is calculated by counting the number of common leading zeros in the (MSB) binary representation of $xXORy$ (0 farthest, 255 closest, 256 self).

Taking the *proximity order* relative to a fix point x classifies the points in the space (byte sequences of length n) into bins. Items in each are at most half as distant from x as items in the previous bin. Given a sample of uniformly distributed items (a hash function over arbitrary sequence) the proximity scale maps onto series of subsets with cardinalities on a negative exponential scale.

It also has the property that any two addresses belonging to the same bin are at most half as distant from each other as they are from x.

If we think of a random sample of items in the bins as connections in a network of interconnected nodes, then relative proximity can serve as the basis for local decisions for graph traversal where the task is to *find a route* between two points. Since on every hop, the finite distance halves, as long as each relevant bin is non-empty, there is a guaranteed constant maximum limit on the number of hops needed to reach one node from the other.

3.2.2.2 Kademlia topology

Swarm uses the ethereum devp2p rlpx suite as the transport layer of the underlay network. This uncommon variant allows semi-stable peer connections (over TCP), with authenticated, encrypted, synchronous data streams.

We say that a node has *kademlia connectivity* if (1) it is connected to at least one node for each proximity order up to (but excluding) some maximum value d (called the *saturation depth*) and (2) it is connected to all nodes whose proximity order relative to the node is greater or equal to d.

If each point of a connected subgraph has kademlia connectivity, then we say the subgraph has *kademlia topology*. In a graph with kademlia topology, (1) a path between any two points exists, (2) it can be found using only local decisions on each hop and (3) is guaranteed to terminate in no more steps than the depth of the destination plus one.

Given a set of points uniformly distributed in the space (e.g., the results of a hash function applied to Swarm data) the proximity bins map onto a series of subsets with cardinalities on a negative exponential scale, i.e., PO bin 0 has half of the points of any random sample, PO bin 1 has one fourth, PO bin 2 one eighth, etc. The expected value of saturation depth in the network of N nodes is $log2(N)$. The last bin can just merge all bins deeper than the depth and is called the *most proximate bin*.

Nodes in the Swarm network are identified by the hash of the ethereum address of the Swarm base account. This serves as their overlay address, the proximity order bins are calculated based on these addresses. Peers connected to a node define another, live kademlia table, where the graph edges represent devp2p rlpx connections.

If each node in a set has a saturated kademlia table of connected peers, then the nodes "live connection" graph has kademlia topology. The properties of a kademlia graph can be used for routing messages between nodes in a network using overlay addressing. In a *forwarding kademlia* network, a message is said to be *routable* if there exists a path from sender node to destination node through which the message could be relayed. In a mature subnetwork with kademlia topology every message is routable. A large proportion of nodes are not stably online; keeping several connected peers in their PO bins, each node can increase the chances that it can forward messages at any point in time, even if a relevant peer drops.

3.2.2.3 Bootstrapping and discovery

Nodes joining a decentralised network are supposed to be naive, i.e., potentially connect via a single known peer. For this reason, the bootstrapping process will need to include a discovery component with the help of which nodes exchange information about each other.

The protocol is as follows: Initially, each node has zero as their saturation depth. Nodes keep advertising to their connected peers info about their saturation depth as it changes. If a node establishes a new connection, it notifies each of its peers about this new connection if their proximity order relative to the respective peer is not lower than the peer's advertised saturation depth (i.e., if they are sufficiently close by). The notification is always sent to each peer that shares a PO bin with the new connection. These notification about connected peers contain full overlay and underlay address information. Light nodes that do not wish to relay messages and do not aspire to build up a healthy kademlia are discounted.

As a node is being notified of new peer addresses, it stores them in a kademlia table of known peers. While it listens to incoming connections, it also proactively attempts to connect to nodes in order to achieve saturation: it tries to connect to each known node that is within the PO boundary of N *nearest neighbours* called *nearest neighbour depth* and (2) it tries to fill each bin up to the nearest neighbour depth with healthy peers. To satisfy (1) most efficiently, it attempts to connect to the peer that is most needed at any point in time. Low (far) bins are more important to fill than high (near) ones since they handle more volume. Filling an empty bin with one peer is more important than adding a new peer to a non-empty bin, since it leads to a saturated kademlia earlier. Therefore the protocol uses a bottom-up, depth-first strategy to choose a peer to connect to. Nodes that are tried but failed to get connected are retried with an exponential backoff (i.e., after a time interval that doubles after each attempt). After a certain number of attempts such nodes are no longer considered.

After a sufficient number of nodes are connected, a bin becomes saturated, and the bin saturation depth can increase. Nodes keep advertising their current saturation depth to their peers if it changes. As their saturation depth increases, nodes will get notified of fewer and fewer new peers (since they already know their neighbourhood). Once the node finds all their nearest neighbours and has saturated all the bins, no new peers are expected. For this reason, a node can conclude a saturated kademlia state if it receives no new peers (for some time). The node does not need to know the number of nodes in the network. In fact, some time after the node stops receiving new peer addresses, the node can effectively estimate the size of the network from the depth (depth n implies 2^n nodes)

Such a network can readily be used for a forwarding-style messaging system. Swarm's PSS is based on this. Swarm also uses this network to implement its storage solution.

3.2.3 Distributed preimage archive

Distributed hash tables (DHTs) utilise an overlay network to implement a key-value store distributed over the nodes. The basic idea is that the keyspace is mapped onto the overlay address space, and information about an element in the container is to be found with nodes whose address is in the proximity of the key. DHTs for decentralised content addressed storage typically associate content fingerprints with a list of nodes (seeders) who can serve that content. However, the same structure can be used directly: it is not information about the location of content that is stored at the node closest to the address (fingerprint), but the content itself. We call this structure *distributed preimage archive* (DPA).

A DPA is opinionated about which nodes store what content and this implies a few more restrictions: (1) load balancing of content among nodes is required and is accomplished by splitting content into equal

sized chunks (*chunking*); (2) there has to be a process whereby chunks get to where they are supposed to be stored (*syncing*); and (3) since nodes do not have a say in what they store, measures of *plausible deniability* should be employed.

Chunk retrieval in this design is carried out by relaying retrieve requests from a requestor node to a storer node and passing the retrieved chunk from the storer back to the requestor.

Since Swarm implements a DPA (over chunks of 4096 bytes), relaying a retrieve request to the chunk address as destination is equivalent to passing the request towards the storer node. Forwarding kademlia is able to route such retrieve requests to the neighbourhood of the chunk address. For the delivery to happen we just need to assume that each node when it forwards a retrieve request, remembers the requestors. Once the request reaches the storer node, delivery of the content can be initiated and consists in relaying the chunk data back to the requestor(s).

In this context, a chunk is retrievable for a node if the retrieve request is routable to the storer closest to the chunk address and the delivery is routable from the storer back to the requestor node. The success of retrievals depends on (1) the availability of strategies for finding such routes and (2) the availability of chunks with the closest nodes (syncing). The latency of request–delivery roundtrips hinges on the number of hops and the bandwidth quality of each node along the way. The delay in availability after upload depends on the efficiency of the syncing protocol.

3.2.3.1 Redundancy

If the closest node is the only storer and drops out, there is no way to retrieve the content. This basic scenario is handled by having a set of nearest neighbours holding replicas of each chunk that is closest to any of them. A chunk is said to be *redundantly retrievable* of degree math:*n* if it is retrievable and would remain so after any math:*n-1* responsible nodes leave the network. In the case of request forwarding failures, one can retry, or start concurrent retrieve requests. Such fallback options are not available if the storer nodes go down. Therefore redundancy is of major importance.

The area of the fully connected neighbourhood defines an *area of responsibility*. A storer node is responsible for (storing) a chunk if the chunk falls within the node's area of responsibility. Let us assume, then, (1) a forwarding strategy that relays requests along stable nodes and (2) a storage strategy that each node in the nearest neighbourhood (of mimimum R peers) stores all chunks within the area of responsibility. As long as these assumptions hold, each chunk is retrievable even if $R-1$ storer nodes drop offline simultaneously. As for (2), we still need to assume that every node in the nearest neighbour set can store each chunk.

Further measures of redundancy, e.g. Erasure coding, will be implemented in the future.

3.2.3.2 Caching and purging Storage

Node synchronisation is the protocol that makes sure content ends up where it is queried. Since the Swarm has an address-key based retrieval protocol, content will be twice as likely be requested from a node that is one bit (one proximity bin) closer to the content's address. What a node stores is determined by the access count of chunks: if we reach the capacity limit for storage the oldest unaccessed chunks are removed. On the one hand, this is backed by an incentive system rewarding serving chunks. This directly translates to a motivation, that a content needs to be served with frequency X in order to make storing it profitable. On the one hand , frequency of access directly translates to storage count. On the other hand, it provides a way to combine proximity and popularity to dictate what is stored.

Based on distance alone (all else being equal, assuming random popularity of chunks), a node could be expected to store chunks up to a certain proximity radius. However, it is always possible to look for further content that is popular enough to make it worth storing. Given the power law of popularity rank and the uniform distribution of chunks in address space, one can be sure that any node can expand their storage with content where popularity of a stored chunk makes up for their distance.

Given absolute limits on popularity, there might be an actual upper limit on a storage capacity for a single base address that maximises profitablity. In order to efficiently utilise excess capacity, several nodes should be run in parallel.

This storage protocol is designed to result in an autoscaling elastic cloud where a growth in popularity automatically scales. An order of magnitude increase in popularity will result in an order of magnitude more nodes actually caching the chunk resulting in fewer hops to route the chunk, ie., a lower latency retrieval.

3.2.3.3 Synchronisation

Smart synchronisation is a protocol of distribution which makes sure that these transfers happen. Apart from access count which nodes use to determine which content to delete if capacity limit is reached, chunks also store their first entry index. This is an arbitrary monotonically increasing index, and nodes publish their current top index, so virtually they serve as timestamps of creation. This index helps keeping track what content to synchronise with a peer.

When two nodes connect and they engage in synchronisation, the upstream node offers all the chunks it stores locally in a datastream per proximity order bin. To receive chunks closer to a downstream than to the upstream, downstream peer subscribes to the data stream of the PO bin it belongs to in the upstream node's kademlia table. If the peer connection is within nearest neighbour depth the downstream node subscribes to all PO streams that constitute the most proximate bin.

Nodes keep track of when they stored a chunk locally for the first time (for instance by indexing them by an ever incrementing storage count). The downstream peer is said to have completed *history syncing* if it has (acknowledged) all the chunks of the upstream peer up from the beginning until the time the session started (up to the storage count that was the highest at the time the session started). Some node is said to have completed *session syncing* with its upstream peer if it has (acknowledged) all the chunks of the upstream peer up since the session started.

In order to reduce network traffic resulting from receiving chunks from multiple sources, all store requests can go via a confirmation roundtrip. For each peer connection in both directions, the source peer sends an *offeredHashes* message containing a batch of hashes offered to push to the recipient. Recipient responds with a *wantedHashes*.

3.2.4 Data layer

There are 4 different layers of data units relevant to Swarm:

- *message*: p2p RLPx network layer. Messages are relevant for the devp2p wire protocols
- *chunk*: fixed size data unit of storage in the distributed preimage archive
- *file*: the smallest unit that is associated with a mime-type and not guaranteed to have integrity unless it is complete. This is the smallest unit semantic to the user, basically a file on a filesystem.
- *collection*: a mapping of paths to files is represented by the *swarm manifest*. This layer has a mapping to file system directory tree. Given trivial routing conventions, a url can be mapped to files in a standardised way, allowing manifests to mimic site maps/routing tables. As a result, Swarm is able to act as a webserver, a virtual cloud hosting service.

The actual storage layer of Swarm consists of two main components, the *localstore* and the *netstore*. The local store consists of an in-memory fast cache (*memory store*) and a persistent disk storage (*dbstore*). The NetStore is extending local store to a distributed storage of Swarm and implements the *distributed preimage archive (DPA)*.

3.2.4.1 Files

The *FileStore* is the local interface for storage and retrieval of files. When a file is handed to the FileStore for storage, it chunks the document into a merkle hashtree and hands back its root key to the caller. This key can later be used to retrieve the document in question in part or whole.

The component that chunks the files into the merkle tree is called the *chunker*. Our chunker implements the *bzzhash* algorithm which is parallellized tree hash based on an arbitrary *chunk hash*. When the chunker is handed an I/O reader (be it a file or webcam stream), it chops the data stream into fixed sized chunks. The chunks are hashed using an arbitrary chunk hash (in our case the BMT hash, see below). If encryption is used the chunk is encrypted before hashing. The references to consecutive data chunks are concatenated and packaged into a so called *intermediate chunk*, which in turn is encrypted and hashed and packaged into the next level of intermediate chunks. For unencrypted content and 32-byte chunkhash, the 4K chunk size enables 128 branches in the resulting Swarm hash tree. If we use encryption, the reference is 64-bytes, allowing for 64 branches in the Swarm hash tree. This recursive process of constructing the Swarm hash tree will result in a single root chunk, the chunk hash of this root chunk is the Swarm hash of the file. The reference to the document is the Swarm hash itself if the upload is unencrypted, and the Swarm hash concatenated with the decryption key of the rootchunk if the upload is encrypted.

When the FileStore is handed a reference for file retrieval, it calls the Chunker which hands back a seekable document reader to the caller. This is a *lazy reader* in the sense that it retrieves parts of the underlying document only as they are being read (with some buffering similar to a video player in a browser). Given the reference, the FileStore takes the Swarm hash and using the NetStore retrieves the root chunk of the document. After decrypting it if needed, references to chunks on the next level are processed. Since data offsets can easily be mapped to a path of intermediate chunks, random access to a document is efficient and supported on the lowest level. The HTTP API offers range queries and can turn them to offset and span for the lower level API to provide integrity protected random access to files.

Swarm exposes the FileStore API via the bzz-raw URL scheme directly on the HTTP local proxy server (see BZZ URL schemes and API Reference). This API allows file upload via POST request as well as file download with GET request. Since on this level the files have no mime-type associated, in order to properly display or serve to an application, the `content_type` query parameter can be added to the url. This will set the proper content type in the HTTP response.

3.2.4.2 Manifests

The Swarm *manifest* is a structure that defines a mapping between arbitrary paths and files to handle collections. It also contains metadata associated with the collection and its objects (files). Most importantly a manifest entry specifies the media mime type of files so that browsers know how to handle them. You can think of a manifest as (1) routing table, (2) an index or (3) a directory tree, which make it possible for Swarm to implement (1) web sites, (2) databases and (3) filesystem directories. Manifests provide the main mechanism to allow URL based addressing in Swarm. The domain part of the URL maps onto a manifest in which the path part of the URL is looked up to arrive at a file entry to serve.

Manifests are currently respresented as a compacted trie (http://en.wikipedia.org/wiki/Trie) , with individual trie nodes serialised as json. The json structure has an array of *manifest entries* minimally with a path and a reference (Swarm hash address). The path part is used for matching the URL path, the reference may point to an embedded manifest if the path is a common prefix of more than one path in the collection. When you retrieve a file by url, Swarm resolves the domain to a reference to a root manifest, which is recursively traversed to find the matching path.

The high level API to the manifests provides functionality to upload and download individual documents as files, collections (manifests) as directories. It also provides an interface to add documents to a collection on a path, delete a document from a collection. Note that deletion here only means that a new manifest is created in which the path in question is missing. There is no other notion of deletion in the Swarm. Swarm exposes the manifest API via the *bzz* URL scheme, see BZZ URL schemes.

These HTTP proxy API is described in detail in the API Reference section.

Note: In POC4, json manifests will be replaced by a serialisation scheme that enables compact path proofs, essentially asserting that a file is part of a collection that can be verified by any third party or smart contract.

3.2.5 Components

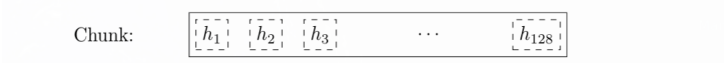
In what follows we describe the components in more detail.

3.2.5.1 Swarm Hash

Swarm Hash (a.k.a. *bzzhash*) is a Merkle tree hash designed for the purpose of efficient storage and retrieval in content-addressed storage, both local and networked. While it is used in Swarm, there is nothing Swarm-specific in it and the authors recommend it as a drop-in substitute of sequential-iterative hash functions (like SHA3) whenever one is used for referencing integrity-sensitive content, as it constitutes an improvement in terms of performance and usability without compromising security.

In particular, it can take advantage of parallelisation for faster calculation and verification, can be used to verify the integrity of partial content without having to transmit all of it (and thereby allowing random access to files). Proofs of security to the underlying hash function carry over to Swarm Hash.

Swarm Hash is constructed using any chunk hash function with a generalization of Merkle's tree hash scheme. The basic unit of hashing is a *chunk*, that can be either a *data chunk* containing a section of the content to be hashed or an *intermediate chunk* containing hashes of its children, which can be of either variety.



A swarm chunk consists of 4096 bytes of the file or a sequence of 128 subtree hashes.

Hashes of data chunks are defined as the hashes of the concatenation of the 64-bit length (in LSB-first order) of the content and the content itself. Because of the inclusion of the length, it is resistant to length extension attacks, even if the underlying chunk hash function is not.

Intermediate chunks are composed of the hashes of the concatenation of the 64-bit length (in LSB-first order) of the content subsumed under this chunk followed by the references to its children (reference is either a chunk hash or chunk hash plus decryption key for encrypted content).

To distinguish between the two, one should compare the length of the chunk to the 64-bit number with which every chunk begins. If the chunk is exactly 8 bytes longer than this number, it is a data chunk. If it is shorter than that, it is an intermediate chunk. Otherwise, it is not a valid Swarm Hash chunk.

For the chunk hash we use a hashing algorithm based on a binary merkle tree over the 32-byte segments of the chunk data using a base hash function. Our choice for this base hash is the ethereum-wide used Keccak 256 SHA3 hash. For integrity protection the 8 byte span metadata is hashed together with the root of the BMT resulting in the BMT hash. BMT hash is ideal for compact solidity-friendly inclusion proofs.

3.2.5.2 Chunker

Chunker is the interface to a component that is responsible for disassembling and assembling larger data. More precisely *Splitter* disassembles, while *Joiner* reassembles documents.

Our Splitter implementation is the *pyramid* chunker that does not need the size of the file, thus is able to process live capture streams. When *splitting* a document, the freshly created chunks are pushed to the DPA via the NetStore and calculates the Swarm hash tree to return the *root hash* of the document that can be used as a reference when retrieving the file.

When *joining* a document, the chunker needs the Swarm root hash and returns a *lazy reader*. While joining, for chunks not found locally, network protocol requests are initiated to retrieve chunks from other nodes. If chunks are retrieved (i.e. retrieved from memory cache, disk-persisted db or via cloud based Swarm delivery from other peers in the DPA), the chunker then puts these together on demand as and where the content is being read.

3.3 Incentives system

Important: Currently, running incentivized nodes is optional. Please refer to chapter 6.8 for information on how to enable incenvization on your node (currently testnet only). When mainnet is released, enabled incentives will be the default mode.

3.3.1 Cryptoeconomics

Cryptoeconomics is usually (and somewhat obviously) understood as the intersection of cryptography with economy [Cryptoeconomics].

The cryptographic part in a broader sense encompasses the vast world of cryptocurrencies, blockchain networks and secure digital decentralized systems, while the economics side generally deals with economic *incentives* for network participants. The basic principle is to reward individual network participants in some way for contributing resources to the network at large, with the aim to ensure smooth and successful operation of a network, guarantee protocol execution, and fulfill end users' requirements.

Every project with cryptoeconomic elements defines its own rules, protocols and incentives system to reward users, which reflect the different needs and operating principles of the specific network.

For Swarm to properly function as a decentralized p2p storage and communication infrastructure, on very basic tern

- contribute bandwidth for incoming and outgoing requests
- provide storage for users to upload and retrieve data
- forward incoming requests to peers who can fulfill them if they can not serve the request themselves

Swarm introduces its own incentives system for ensuring correct network behavior by rewarding nodes for serving

- accounting system
- file insurance
- litigation

Here we only describe the global behavior of the SWarm Accounting Protocol (SWAP) in practical terms. For more information on the whole incentive system or for more background into SWAP we refer to [Incentives system for Swarm].

3.3.2 Accounting

At the core of Swarm's incentive system is the SWarm Accounting Protocol, or SWAP. It is a simple protocol which r

- a node gets rewarded for serving resources
- a node gets charged for requesting resources

Every peer maintains a local database on its own file system which persists accounting information of all peers with which it exchanged data. On every change, the in-memory accounting structures are updated and immediately persisted to disk in order to avoid synchronization issues.

Note: SWAP excels at its simplicity. It is not the goal and would completely defeat the purpose of the system to introduce bullet-proof consistency and auditing - Swarm is not a blockchain in its own terms.

Important: The Swarm node operator is responsible for maintenance of the local accounting database (a LevelDB instance). Tampering with it, altering its contents or removing it altogether may result in the node becoming inoperable or a loss of funds.

If a node requests data from a peer, when the peer delivers the data, the peer credits an amount equivalent to the price of that message in its local database, at the index representing the node it delivers to. The node, when it receives the message (data) from the peer, debits itself in its local database the same amount at the index of the peer.

In the above example, A requested data from B, which B then delivers to A (could be a chunk). Assuming a price of 200 units for the chunk, B credits itself in its local database 200 at the index for node A, while A, when it finally receives the data from B, debits 200 in its local database at the index for node B.

If the balance tilts too much to either side as a result of unequal consumption or high variance in data usage, SWAP triggers a settlement process (see *Settlement with cheques*).

Due to the distributed nature of Swarm nodes, entries in the local databases do not happen simultaneously, and there is no notion of transactions nor confirmations; in normal operation, we assume that peers credit and debit roughly the same amounts to each other's balances over time. Of course, a small variance in balance may exist, but it is expected to be 0 in the long-run.

If a message is being sent, if we would account for it before sending, then a failure in sending would require a complex rollback (or an imbalance). If we send first and then account, then it would not be possible to actually check if there are enough funds. Receiving a message incurs in the same challenge. To address this, Swarm first does a read operation on the database to check if there are enough funds. If positive, the requested operation (send/receive) is performed, and finally if the operation succeeded, the accounting entry is persisted.

3.3.2.1 Thresholds

As described in the [Incentives system for Swarm], this accounting system works by allowing the mutual accounting to oscillate between a defined range. This range is delimited by two thresholds: * The payment threshold is a number at which if the node goes below that relative to a peer, it should trigger a payment to that peer (a cheque, see below). This is initiated by the debitor node. * The disconnect threshold is a number at which if the node goes above that number relative to a peer, it can (it's up to the peer but this is the expected behavior) disconnect the peer. This is monitored by the creditor.

The distance between the two thresholds is designed to be such that any normal variance in the balances between peers does not cause a disconnect. Swarm provides such configuration by default, but peers is free to set the disconnect threshold in the way they prefer.

The disconnect threshold amount must be bigger than the payment threshold in absolute terms.

We advise against changing the payment threshold to a value above the default value, as this may lead to being disconnected by other nodes.

3.3.2.2 Fraud risks

Of course, the design with individual peer databases means that nodes can alter their database and pretend to have different balances to other nodes. The simplicity of this mutual accounting though effectively significantly limits fraud, as if node A modifies its entry with B, it has only a local effect, as it can not trigger B to send it a cheque or to force it to do any action. Normal behavior is to disconnect a node in this case. However, we want to make it clear that a node can lose funds up to the disconnect threshold amount due to freeriders.

Another possibility is that a node does not initiate a settlement after he crosses the payment threshold with a peer.

Note: A peer can be freeriding by consuming resources up to the disconnect threshold. At this point, if there is no settlement, the peer simply gets disconnected.

3.3.2.3 Imbalances

Imbalances between nodes more generally leads to disconnects from peers. The downside of this is that if node A was able to send an accounted message to B, which successfully left A, but for some reason never arrived at B, then this would lead to imbalances as well. Swarm currently treats this case as an edge case and does not implement any balance synchronization nor clearance protocol to address such cases. It may though be considered for the future.

3.3.2.4 Settlement with cheques

The balance entries for each node in the local database represent accounting entries but are in the end just numbers in Swarm's internal accounting unit. We described above (*Thresholds*) the notion of a payment threshold at which a financial settlement protocol is initiated.

If a node's balance with a peer crosses the payment threshold, which is a number every node can set individually (see *swap-payment-threshold* flag), but has a reasonable default defined in the code, then the node kicks off the settlement process. This process involves a series of security and sanity checks, culminating in sending a **signed cheque** to its peer. This signed cheque is a piece of data containing the amount, the source chequebook address and the beneficiary address, as well as the signature of the cheque issuer.

The peer, upon receiving the cheque, resets the balance with the cheque value and will initiate a cashing transaction trying to cash the cheque (considering it makes economic sense as described below) - this is a transaction on a blockchain and represents real financial value. If the cheque was valid and backed by funds, this results in a real transfer of funds from the issuer's contract address to the beneficiary's. All of the accounting is maintained per peer and thus threshold crossing and cheque issuance is based on every individual balance with every peer.

3.3.2.5 Interaction with the blockchain

A cheque is sent as part of its own lower layer transport protocol (currently on top of *devp2p*, with the planned transition of the whole of Swarm to *libp2p*). The receiving peer handles the cheque and tries to cash it by issuing a transaction on the blockchain.

Every node has to deploy its own instance of the Swarm smart contract, also often referred to as *chequebook*. So when receiving a cheque, the beneficiary issues a transaction *on the contract of the issuer of the cheque* as a cashing request. As the beneficiary is initiating the transaction, it is also the beneficiary who is paying for the transaction. Swarm per default has a check to make sure it financially makes sense to do this: currently the transaction is only started if the payout of the cheque is twice as big as the (estimated) transaction costs.

The smart contract, after doing the appropriate sanity checks (checking the validity and the funds of the issuer), will transfer the funds from the issuer's contract to the beneficiary contract. If the cheque bounced due to insufficient funds in the issuer's smart contract, the peer who owns this chequebook is disconnected. Again, it is important to understand that a peer is able to freeride up to this threshold. The issuer cannot issue a cheque if there are insufficient funds; normal operation involves a smart contract call to check for funds. If a malicious node sends a bogus, malformed or altered cheque, the cheque will bounce and thus can be identified as fraud attempt, resulting in a disconnect.

Refer to the documentation [Incentives system for Swarm] for details and specification of how the protocol handles subsequent cheques and how this evolves over time.

Starting a SWAP enabled Swarm node thus requires a node to have funds. If no contract address is provided, the node will automatically create one, and a transaction is attempted when the node starts. The user can specify the amount to fund (in WEI) via command line parameter at boot (*swap-deposit-amount*). If the transaction failed, the node does not boot. Afterwards, the node remembers the contract address and uses it for future blockchain interaction. If no *swap-deposit-amount* flag is provided, a deposit prompt will ask the user for funds at boot and the transaction will happen on every boot, not just the first one. Set the amount to zero if no further deposit is desired (check also the *swap-skip-deposit* flag).

While currently cheque cash in is triggered automatically, in the future, Swarm might consider making this configurable by the user.

To prevent fraudulent creation of contract addresses for the sake of stealing funds via the protocol, incentivized nodes need to contact an audited factory contract address when creating the new chequebook. For every blockchain, one factory should be deployed. For Ethereum networks, Swarm will provide the network address (and the deployment of the factory). For other platforms, the factory address needs to be configured via command line parameter for incentivized nodes (*swap-chequebook-factory*).

Important: A public incentivized Swarm testnet is currently operational. For information on how to connect to it, please refer to chapter 6.8. The testnet is experimental and funds can be lost at all times. Use only test tokens. The release of the public incentivized mainnet will be announced when launched.

3.3.2.6 Smart contracts

The smart contract code is open source and there is a separate code repository for it: https://github.com/ethersphere/swap-swear-and-swindle.

3.3.2.7 Metrics

There is some instrumentation for observing SWAP performance based on the Swarm node's metrics setup. All metrics can be switched on via the *-metrics* flag. Specific metrics for SWAP are:

- Number of emitted cheques `swap.cheques.emitted.num`
- Number of received cheques `swap.cheques.received.num`
- Amount of emitted cheques `swap.cheques.emitted.honey`
- Amount of received cheques `swap.cheques.received.honey`
- Amount of cashed cheques `swap.cheques.cashed.honey`
- Number of bounced cheques `swap.cheques.cashed.bounced`
- Number of errors in cheques processing `swap.cheques.cashed.errors`

At a lower level, there are more metrics:

- Amount of bytes credited `account.bytes.credit`
- Amount of bytes debited `account.bytes.debit`
- Amount of accounted units credited `account.balance.credit`
- Amount of accounted units debited `account.balance.debit`
- Amount of accounted messages credited `account.msg.credit`
- Amount of accounted messages debited `account.msg.debit`
- Number of disconnected peers due to accounting errors `account.peerdrops`, `account.selfdrops`

For more information regarding the metrics system, refer to the chapter Metrics reporting.

3.3.3 Honey Token

Swarm introduces its own token: **Honey**, which is an ERC20 compatible token. The rationale is to allow a homogeneous operation inside the Swarm network in terms of accounting and settlement, externalizing value fluctuations if multiple blockchains are considered.

Honey test tokens can be obtained by an onchain faucet. Refer to the chapter Run your own swap-enabled node for instructions.

For details of emission and token design we have to refer to upcoming documentation which will be published soon. For now we only want to point out that for incentivized nodes to work, the prefunding of the chequebook contract for nodes needs to be done with Honey tokens.

3.3.4 PricedMessage

All data exchange between Swarm node is based on the underlying transport protocol and is modeled as message exchange. This means that for a node A to send data to a node B means that A sends a message to B.

Messages are identified by their type. Swarm accounts only for message types which are marked as "accountable". In the go implementation, this is done by implementing the interface *PricedMessage*.

Currently, only chunk delivery messages are priced. Thus accounting is only effective on messages which deliver chunks to peers. The delivering peer is credited, the receiving peer is debited.

Other message types are exchanged without incurring into accounting (e.g. syncing is free).

3.3.5 Spam protection: Postage Stamps

As described above, syncing is not accounted for. In Swarm, syncing is the process through which the network distributes chunks based on their hash. When a user uploads a resource to Swarm, the resource is chopped up in 4kB chunks which are content addressed. Based on this address, every chunk gets sent to the peers which are closest to that address for storage. As this is a network internal operation, it should not incur costs.

However, this introduces a major spamming problem: anyone could just upload junk data which would be distributed freely across the network, constituting a denial of service attack.

To counteract this, Swarm uses the analogy of postage stamps from conventional mail carrier systems. In conventional mail delivery, to be able to send a letter, it is usually required to buy a postage stamp to be stuck onto a letter or package. Essentially, it is prepaying for the delivery. Post offices world wide then can verify that the delivery is legitimate by looking at the postage stamp (and verifying that the value is correct).

In Swarm, a postage stamp will be a piece of cryptographic data attached to a chunk. Before uploading a resource, users must attach a postage stamp to its chunks by sending some amount to a smart contract which will then provide the functionality to attach valid stamps to the chunks. Syncing nodes will then look at every chunk and verify that the postage stamp is valid. If it is, the chunks will be forwarded / stored. Otherwise, the chunks will be rejected. In other words, uploads will cost some cryptocurrency. The amount and the details of this operation are still being refined, but it can be anticipated that the amount should be small, as it is only meant to prevent spam.

The implementation of postage stamps is still pending. For details, please consult the postage stamp specification at Postage Stamps.

3.3.6 Multiple blockchains

Swarm works by connecting different nodes based on their **Swarm Network ID** (also called BZZ Network ID). The network ID is just a number: during handshake nodes exchange their network ID, if it doesn't match, the nodes don't connect.

This same principle applies for incentivized nodes as well. However, in this case, the network ID also represents an actual blockchain network (to be precise, Swarm uses the blockchain network ID). There will be a mapping between network IDs and (public) blockchains. The reason for this is that Swarm per se is blockchain agnostic: although it was initially designed to work with Ethereum, it can potentially work with any blockchain (and currently most easily with any Ethereum-compatible platform).

Incentivized nodes exchange their contract address during handshake (in order to send each other cheques later). Thus, if the contract addresses would not be on the same smart contract platform, the cheques would fail. Therefore, incentivized nodes **must run on the same blockchain platform (backend)** for incentivization to work properly (strictly speaking, they could work in an agnostic mode as long as no cheque is being exchanged, as SWAP would be accounting independently, but settlement would not be possible).

Important: Incentivized nodes need to be connected on the same smart contract platform.

Incentivized nodes need to provide their operating blockchain platform at boot via command line parameter (*swap-backend-url*).

3.4 Swarm Specification Process

The Swarm Specification Process is named SWIP (Swarm Improvement Proposal) and can be found here: https://github.com/ethersphere/swarm-docs/tree/restructuring/SWIP

3.5 Swarm for DApp-Developers

This section is written for developers who want to use Swarm in their own applications.

Swarm offers a **local HTTP proxy** API that dapps or command line tools can use to interact with Swarm. Some modules like messaging are only available through RPC-JSON API. The foundation servers on the testnet are offering public gateways, which serve to easily demonstrate functionality and allow free access so that people can try Swarm without even running their own node.

Swarm is a collection of nodes of the devp2p network each of which run the BZZ URL schemes on the same network id.

3.5.1 Public gateways

Swarm offers an HTTP based API that DApps can use to interact with Swarm. The Ethereum Foundation is hosting a public gateway, which allows free access so that people can try Swarm without running their own node. The Swarm public gateway can be found at https://swarm-gateways.net and is always running the latest Swarm release.

Important: Swarm public gateways are temporary and users should not rely on their existence for production services.

3.5.2 Uploading and downloading

3.5.2.1 Introduction

Note: This guide assumes you've installed the Swarm client and have a running node that listens by default on port 8500. See Getting Started for details.

Arguably, uploading and downloading content is the raison d'être of Swarm. Uploading content consists of "uploading" content to your local Swarm node, followed by your local Swarm node "syncing" the resulting chunks of data with its peers in the network. Meanwhile, downloading content consists of your local Swarm node querying its peers in the network for the relevant chunks of data and then reassembling the content locally.

Uploading and downloading data can be done through the `swarm` command line interface (CLI) on the terminal or via the HTTP interface on http://localhost:8500.

3.5.2.2 Using HTTP

Swarm offers an HTTP API. Thus, a simple way to upload and download files to/from Swarm is through this API. We can use the `curl` tool to exemplify how to interact with this API.

Note: Files can be uploaded in a single HTTP request, where the body is either a single file to store, a tar stream (application/x-tar) or a multipart form (multipart/form-data).

To upload a single file to your node, run this:

```
$ curl -X POST -H "Content-Type: text/plain" --data "some-data" http://
↪localhost:8500/bzz:/
```

Once the file is uploaded, you will receive a hex string which will look similar to this:

```
027e57bcbae76c4b6a1c5ce589be41232498f1af86e1b1a2fc2bdffd740e9b39
```

This is the Swarm hash of the address string of your content inside Swarm. It is the same hash that would have been returned by using the :ref:`swarm up <swarmup>` command.

To download a file from Swarm, you just need the file's Swarm hash. Once you have it, the process is simple. Run:

```
$ curl http://localhost:8500/bzz:/
↪027e57bcbae76c4b6a1c5ce589be41232498f1af86e1b1a2fc2bdffd740e9b39/
```

The result should be your file:

```
some-data
```

And that's it.

Note: If you omit the trailing slash from the url then the request will result in a HTTP redirect. The semantically correct way to access the root path of a Swarm manifest is using the trailing slash.

Tar stream upload

Tar is a traditional unix/linux file format for packing a directory structure into a single file. Swarm provides a convenient way of using this format to make it possible to perform recursive uploads using the HTTP API.

```
# create two directories with a file in each
$ mkdir dir1 dir2
$ echo "some-data" > dir1/file.txt
$ echo "some-data" > dir2/file.txt

# create a tar archive containing the two directories (this will tar everything in␣
↪the working directory)
tar cf files.tar .

# upload the tar archive to Swarm to create a manifest
$ curl -X POST -H "Content-Type: application/x-tar" --data-binary @files.tar http:/
↪/localhost:8500/bzz:/
> 1e0e21894d731271e50ea2cecf60801fdc8d0b23ae33b9e808e5789346e3355e
```

You can then download the files using:

```
$ curl http://localhost:8500/bzz:/
↪1e0e21894d731271e50ea2cecf60801fdc8d0b23ae33b9e808e5789346e3355e/dir1/file.txt
> some-data

$ curl http://localhost:8500/bzz:/
↪1e0e21894d731271e50ea2cecf60801fdc8d0b23ae33b9e808e5789346e3355e/dir2/file.txt
> some-data
```

GET requests work the same as before with the added ability to download multiple files by setting *Accept: application/x-tar*:

```
$ curl -s -H "Accept: application/x-tar" http://localhost:8500/bzz:/
↪ccef599d1a13bed9989e424011aed2c023fce25917864cd7de38a761567410b8/ | tar t
> dir1/file.txt
  dir2/file.txt
```

Multipart form upload

```
$ curl -F 'dir1/file.txt=some-data;type=text/plain' -F 'dir2/file.txt=some-data;
↪type=text/plain' http://localhost:8500/bzz:/
> 9557bc9bb38d60368f5f07aae289337fcc23b4a03b12bb40a0e3e0689f76c177

$ curl http://localhost:8500/bzz:/
↪9557bc9bb38d60368f5f07aae289337fcc23b4a03b12bb40a0e3e0689f76c177/dir1/file.txt
> some-data

$ curl http://localhost:8500/bzz:/
↪9557bc9bb38d60368f5f07aae289337fcc23b4a03b12bb40a0e3e0689f76c177/dir2/file.txt
> some-data
```

Add files to an existing manifest using multipart form

```
$ curl -F 'dir3/file.txt=some-other-data;type=text/plain' http://localhost:8500/
↪bzz:/9557bc9bb38d60368f5f07aae289337fcc23b4a03b12bb40a0e3e0689f76c177
> ccef599d1a13bed9989e424011aed2c023fce25917864cd7de38a761567410b8

$ curl http://localhost:8500/bzz:/
↪ccef599d1a13bed9989e424011aed2c023fce25917864cd7de38a761567410b8/dir1/file.txt
> some-data
```

(continues on next page)

(continued from previous page)

```
$ curl http://localhost:8500/bzz:/
↪ccef599d1a13bed9989e424011aed2c023fce25917864cd7de38a761567410b8/dir3/file.txt
> some-other-data
```

Upload files using a simple HTML form

```
<form method="POST" action="/bzz:/" enctype="multipart/form-data">
  <input type="file" name="dir1/file.txt">
  <input type="file" name="dir2/file.txt">
  <input type="submit" value="upload">
</form>
```

Listing files

Note: The `jq` command mentioned below is a separate application that can be used to pretty-print the json data retrieved from the `curl` request

A *GET* request with `bzz-list` URL scheme returns a list of files contained under the path, grouped into common prefixes which represent directories:

```
$ curl -s http://localhost:8500/bzz-list:/
↪ccef599d1a13bed9989e424011aed2c023fce25917864cd7de38a761567410b8/ | jq .
> {
   "common_prefixes": [
     "dir1/",
     "dir2/",
     "dir3/"
   ]
 }
```

```
$ curl -s http://localhost:8500/bzz-list:/
↪ccef599d1a13bed9989e424011aed2c023fce25917864cd7de38a761567410b8/dir1/ | jq .
> {
  "entries": [
    {
      "path": "dir1/file.txt",
      "contentType": "text/plain",
      "size": 9,
      "mod_time": "2017-03-12T15:19:55.112597383Z",
      "hash": "94f78a45c7897957809544aa6d68aa7ad35df695713895953b885aca274bd955"
    }
  ]
}
```

Setting `Accept: text/html` returns the list as a browsable HTML document.

3.5.2.3 Using the CLI - Command Line Interface

Uploading a file to your local Swarm node

Note: Once a file is uploaded to your local Swarm node, your node will *sync* the chunks of data with other nodes on the network. Thus, the file will eventually be available on the network even when your

original node goes offline.

Important: According to your Swarm node's configuration different syncing mode will be used. If your Swarm node is configured to operate in push-sync mode only - your origin address will be leaked for the proof of custody receipts

The basic command for uploading to your local node is `swarm up FILE`. For example, let's create a file called example.md and issue the following command to upload the file example.md file to your local Swarm node.

```
$ echo "this is an example" > example.md
$ swarm up example.md
> d1f25a870a7bb7e5d526a7623338e4e9b8399e76df8b634020d11d969594f24a
```

In order to track progress of your upload you can use the `--progress` flag:

```
$ swarm up --progress example.md
  Swarm Hash: d1f25a870a7bb7e5d526a7623338e4e9b8399e76df8b634020d11d969594f24a
  Tag UID: 672245080
  Upload status:
  Syncing 23 chunks       12s [-------------------------------------------------
↪----------] 0 %
```

The hash returned is the hash of a swarm manifest. This manifest is a JSON file that contains the `example.md` file as its only entry. Both the primary content and the manifest are uploaded by default.

After uploading, you can access this example.md file from Swarm by pointing your browser to:

```
$ http://localhost:8500/bzz:/
↪d1f25a870a7bb7e5d526a7623338e4e9b8399e76df8b634020d11d969594f24a/
```

The manifest makes sure you could retrieve the file with the correct MIME type.

You can encrypt your file using the `--encrypt` flag. See the Encryption section for details.

Suppressing automatic manifest creation

You may wish to prevent a manifest from being created alongside with your content and only upload the raw content. You might want to include it in a custom index, or handle it as a data-blob known and used only by a certain application that knows its MIME type. For this you can set `--manifest=false`:

```
$ swarm --manifest=false up FILE
> 7149075b7f485411e5cc7bb2d9b7c86b3f9f80fb16a3ba84f5dc6654ac3f8ceb
```

This option suppresses automatic manifest upload. It uploads the content as-is. However, if you wish to retrieve this file, the browser can not be told unambiguously what that file represents. In the context, the hash `7149075b7f485411e5cc7bb2d9b7c86b3f9f80fb16a3ba84f5dc6654ac3f8ceb` does not refer to a manifest. Therefore, any attempt to retrieve it using the `bzz:/` scheme will result in a `404 Not Found` error. In order to access this file, you would have to use the bzz-raw scheme.

Downloading a single file

To download single files, use the `swarm down` command. Single files can be downloaded in the following different manners. The following examples assume `<hash>` resolves into a single-file manifest:

```
$ swarm down bzz:/<hash>            #downloads the file at <hash> to the current␣
↪working directory
$ swarm down bzz:/<hash> file.tmp   #downloads the file at <hash> as ``file.tmp``␣
↪in the current working dir
$ swarm down bzz:/<hash> dir1/      #downloads the file at <hash> to ``dir1/``
```

You can also specify a custom proxy with *–bzzapi*:

```
$ swarm --bzzapi http://localhost:8500 down bzz:/<hash>           #downloads the␣
↪file at <hash> to the current working directory using the localhost node
```

Downloading a single file from a multi-entry manifest can be done with (`<hash>` resolves into a multi-entry manifest):

```
$ swarm down bzz:/<hash>/index.html            #downloads index.html to the␣
↪current working directory
$ swarm down bzz:/<hash>/index.html file.tmp   #downloads index.html as file.tmp␣
↪in the current working directory
$ swarm down bzz:/<hash>/index.html dir1/      #downloads index.html to dir1/
```

..If you try to download from a multi-entry manifest without specifying the file, you will get a *got too many matches for this path* error. You will need to specify a *–recursive* flag (see below).

Uploading to a remote Swarm node

You can upload to a remote Swarm node using the `--bzzapi` flag. For example, you can use one of the public gateways as a proxy, in which case you can upload to Swarm without even running a node.

```
$ swarm --bzzapi https://swarm-gateways.net up example.md
```

Note: This gateway currently only accepts uploads of limited size. In future, the ability to upload to this gateways is likely to disappear entirely.

Uploading a directory

Uploading directories is achieved with the `--recursive` flag.

```
$ swarm --recursive up /path/to/directory
> ab90f84c912915c2a300a94ec5bef6fc0747d1fbaf86d769b3eed1c836733a30
```

The returned hash refers to a root manifest referencing all the files in the directory.

Directory with default entry

It is possible to declare a default entry in a manifest. In the example above, if `index.html` is declared as the default, then a request for a resource with an empty path will show the contents of the file `/index.html`

```
$ swarm --defaultpath /path/to/directory/index.html --recursive up /path/to/
↪directory
> ef6fc0747d1fbaf86d769b3eed1c836733a30ab90f84c912915c2a300a94ec5b
```

You can now access index.html at

```
$ http://localhost:8500/bzz:/
↪ef6fc0747d1fbaf86d769b3eed1c836733a30ab90f84c912915c2a300a94ec5b/
```

and also at

```
$ http://localhost:8500/bzz:/
↪ef6fc0747d1fbaf86d769b3eed1c836733a30ab90f84c912915c2a300a94ec5b/index.html
```

This is especially useful when the hash (in this case `ef6fc0747d1fbaf86d769b3eed1c836733a30ab90f84c912915` is given a registered name like `mysite.eth` in the Ethereum Name Service. In this case the lookup would be even simpler:

```
http://localhost:8500/bzz:/mysite.eth/
```

Note: You can toggle automatic default entry detection with the `SWARM_AUTO_DEFAULTPATH` environment variable. You can do so by a simple `$ export SWARM_AUTO_DEFAULTPATH=true`. This will tell Swarm to automatically look for `<uploaded directory>/index.html` file and set it as the default manifest entry (in the case it exists).

Downloading a directory

To download a directory, use the `swarm down --recursive` command. Directories can be downloaded in the following different manners. The following examples assume <hash> resolves into a multi-entry manifest:

```
$ swarm down --recursive bzz:/<hash>            #downloads the directory at <hash>␣
↪to the current working directory
$ swarm down --recursive bzz:/<hash> dir1/      #downloads the file at <hash> to␣
↪dir1/
```

Similarly as with a single file, you can also specify a custom proxy with `--bzzapi`:

```
$ swarm --bzzapi http://localhost:8500 down --recursive bzz:/<hash> #note the flag␣
↪ordering
```

Important: Watch out for the order of arguments in directory upload/download: it's `swarm --recursive up` and `swarm down --recursive`.

Adding entries to a manifest

The command for modifying manifests is `swarm manifest`.

To add an entry to a manifest, use the command:

```
$ swarm manifest add <manifest-hash> <path> <hash> [content-type]
```

To remove an entry from a manifest, use the command:

```
$ swarm manifest remove <manifest-hash> <path>
```

To modify the hash of an entry in a manifest, use the command:

```
$ swarm manifest update <manifest-hash> <path> <new-hash>
```

Reference table

upload	`swarm up <file>`
~ dir	`swarm --recursive up <dir>`
~ dir w/ default entry (here: index.html)	`swarm --defaultpath <dir>/index.html --recursive up <dir>`
~ w/o manifest	`swarm --manifest=false up`
~ to remote node	`swarm --bzzapi https://swarm-gateways.net up`
~ with encryption	`swarm up --encrypt`
download	`swarm down bzz:/<hash>`
~ dir	`swarm down --recursive bzz:/<hash>`
~ as file	`swarm down bzz:/<hash> file.tmp`
~ into dir	`swarm down bzz:/<hash> dir/`
~ w/ custom proxy	`swarm down --bzzapi http://<proxy address> down bzz:/<hash>`
manifest	
add ~	`swarm manifest add <manifest-hash> <path> <hash> [content-type]`
remove ~	`swarm manifest remove <manifest-hash> <path>`
update ~	`swarm manifest update <manifest-hash> <path> <new-hash>`

Up- and downloading in the CLI: example usage

Up/downloading

Let's create a dummy file and upload it to Swarm:

```
$ echo "this is a test" > myfile.md
$ swarm up myfile.md
> <reference hash>
```

We can download it using the `bzz:/` scheme and give it a name.

```
$ swarm down bzz:/<reference hash> iwantmyfileback.md
$ cat iwantmyfileback.md
> this is a test
```

We can also `curl` it using the HTTP API.

```
$ curl http://localhost:8500/bzz:/<reference hash>/
> this is a test
```

We can use the `bzz-raw` scheme to see the manifest of the upload.

```
$ curl http://localhost:8500/bzz-raw:/<reference hash>/
```

This returns the manifest:

```
{
  "entries": [
    {
      "hash": "<file hash>",
      "path": "myfile.md",
      "contentType": "text/markdown; charset=utf-8",
      "mode": 420,
      "size": 15,
```

(continues on next page)

(continued from previous page)

```
      "mod_time": "<timestamp>"
    }
  ]
}
```

Up/down as is

We can upload the file as-is:

```
$ echo "this is a test" > myfile.md
$ swarm --manifest=false up myfile.md
> <as-is reference hash>
```

We can retrieve it using the `bzz-raw` scheme in the HTTP API.

```
$ curl http://localhost:8500/bzz-raw:/<as-is reference hash>/
> this is a test
```

Manipulate manifests

Let's create a directory with a dummy file, and upload the directory to swarm.

```
$ mkdir dir
$ echo "this is a test" > dir/dummyfile.md
$ swarm --recursive up dir
> <dir hash>
```

We can look at the manifest using `bzz-raw` and the HTTP API.

```
$ curl http://localhost:8500/bzz-raw:/<dir hash>/
```

It will look something like this:

```
{
  "entries": [
    {
      "hash": "<file hash>",
      "path": "dummyfile.md",
      "contentType": "text/markdown; charset=utf-8",
      "mode": 420,
      "size": 15,
      "mod_time": "2018-11-11T16:52:07+01:00"
    }
  ]
}
```

We can remove the file from the manifest using `manifest remove`.

```
$ swarm manifest remove <dir hash> "dummyfile.md"
> <new dir hash>
```

When we check the new dir hash, we notice that it's empty – as it should be.

Let's put the file back in there.

```
$ swarm up dir/dummyfile.md
> <individual file hash>
$ swarm manifest add <new dir hash> "dummyfileagain.md" <individual file hash>
> <new dir hash 2>
```

We can check the manifest under <new dir hash 2> to see that the file is back there.

3.5.3 Example Dapps

- https://swarm-gateways.net/bzz://swarmapps.eth
- source code: https://github.com/ethersphere/swarm-dapps

3.5.4 Working with content

In this chapter, we demonstrate features of Swarm related to storage and retrieval. First we discuss how to solve mutability of resources in a content addressed system using the Ethereum Name Service on the blockchain, then using Feeds in Swarm. Then we briefly discuss how to protect your data by restricting access using encryption. We also discuss in detail how files can be organised into collections using manifests and how this allows virtual hosting of websites. Another form of interaction with Swarm, namely mounting a Swarm manifest as a local directory using FUSE. We conclude by summarizing the various URL schemes that provide simple HTTP endpoints for clients to interact with Swarm.

3.5.4.1 Ethereum Name System (ENS)

Nick Johnson on the Ethereum Name System

Note: In order to *resolve* ENS names, your Swarm node has to be connected to an Ethereum blockchain (mainnet, or testnet). See Getting Started for instructions. This section explains how you can register your content to your ENS name.

ENS is the system that Swarm uses to permit content to be referred to by a human-readable name, such as "theswarm.eth". It operates analogously to the DNS system, translating human-readable names into machine identifiers - in this case, the Swarm hash of the content you're referring to. By registering a name and setting it to resolve to the content hash of the root manifest of your site, users can access your site via a URL such as `bzz://theswarm.eth/`.

Note: Currently The *bzz* scheme is not supported in major browsers such as Chrome, Firefox or Safari. If you want to access the *bzz* scheme through these browsers, currently you have to either use an HTTP gateway, such as https://swarm-gateways.net/bzz:/theswarm.eth/ or use a browser which supports the *bzz* scheme, such as Mist <https://github.com/ethereum/mist>.

Suppose we upload a directory to Swarm containing (among other things) the file `example.pdf`.

```
$ swarm --recursive up /path/to/dir
>2477cc8584cc61091b5cc084cdcdb45bf3c6210c263b0143f030cf7d750e894d
```

If we register the root hash as the `content` for `theswarm.eth`, then we can access the pdf at

```
bzz://theswarm.eth/example.pdf
```

if we are using a Swarm-enabled browser, or at

```
http://localhost:8500/bzz:/theswarm.eth/example.pdf
```

via a local gateway. We will get served the same content as with:

```
http://localhost:8500/bzz:/
↪2477cc8584cc61091b5cc084cdcdb45bf3c6210c263b0143f030cf7d750e894d/example.pdf
```

Please refer to the official ENS documentation for the full details on how to register content hashes to ENS.

In short, the steps you must take are:

1. Register an ENS name.
2. Associate a resolver with that name.
3. Register the Swarm hash with the resolver as the `content`.

We recommend using https://manager.ens.domains/. This will make it easy for you to:

- Associate the default resolver with your name
- Register a Swarm hash.

Note: When you register a Swarm hash with https://manager.ens.domains/ you MUST prefix the hash with 0x. For example 0x2477cc8584cc61091b5cc084cdcdb45bf3c6210c263b0143f030cf7d750e894d

---

3.5.4.2 Feeds

Note: Feeds, previously known as *Mutable Resource Updates*, is an experimental feature, available since Swarm POC3. It is under active development, so expect things to change.

Since Swarm hashes are content addressed, changes to data will constantly result in changing hashes. Swarm Feeds provide a way to easily overcome this problem and provide a single, persistent, identifier to follow sequential data.

The usual way of keeping the same pointer to changing data is using the Ethereum Name Service (ENS). However, since ENS is an on-chain feature, it might not be suitable for each use case since:

1. Every update to an ENS resolver will cost gas to execute
2. It is not be possible to change the data faster than the rate that new blocks are mined
3. ENS resolution requires your node to be synced to the blockchain

Swarm Feeds provide a way to have a persistent identifier for changing data without having to use ENS. It is named Feeds for its similarity with a news feed.

If you are using *Feeds* in conjunction with an ENS resolver contract, only one initial transaction to register the "Feed manifest address" will be necessary. This key will resolve to the latest version of the Feed (updating the Feed will not change the key).

You can think of a Feed as a user's Twitter account, where he/she posts updates about a particular Topic. In fact, the Feed object is simply defined as:

```
type Feed struct {
  Topic Topic
  User  common.Address
}
```

That is, a specific user posting updates about a specific Topic.

Users can post to any topic. If you know the user's address and agree on a particular Topic, you can then effectively "follow" that user's Feed.

Important: How you build the Topic is entirely up to your application. You could calculate a hash of something and use that, the recommendation is that it should be easy to derive out of information that is accesible to other users.

For convenience, `feed.NewTopic()` provides a way to "merge" a byte array with a string in order to build a Feed Topic out of both. This is used at the API level to create the illusion of subtopics. This way of building topics allows using a random byte array (for example the hash of a photo) and merge it with a human-readable string such as "comments" in order to create a Topic that could represent

the comments about that particular photo. This way, when you see a picture in a website you could immediately build a Topic out of it and see if some user posted comments about that photo.

Feeds are not created, only updated. If a particular Feed (user, topic combination) has never posted to, trying to fetch updates will yield nothing.

Feed Manifests

A Feed Manifest is simply a JSON object that contains the `Topic` and `User` of a particular Feed (i.e., a serialized `Feed` object). Uploading this JSON object to Swarm in the regular way will return the immutable hash of this object. We can then store this immutable hash in an ENS Resolver so that we can have a ENS domain that "follows" the Feed described in the manifest.

Feeds API

There are 3 different ways of interacting with *Feeds* : HTTP API, CLI and Golang API.

HTTP API

Posting to a Feed

Since Feed updates need to be signed, and an update has some correlation with a previous update, it is necessary to retrieve first the Feed's current status. Thus, the first step to post an update will be to retrieve this current status in a ready-to-sign template:

1. Get Feed template

```
GET /bzz-feed:/?topic=<TOPIC>&user=<USER>&meta=1
```

```
GET /bzz-feed:/<MANIFEST OR ENS NAME>/?meta=1
```

Where:

- `user`: Ethereum address of the user who publishes the Feed
- `topic`: Feed topic, encoded as a hex string. Topic is an arbitrary 32-byte string (64 hex chars)

Note:

- If `topic` is omitted, it is assumed to be zero, 0x000...
- if `name=<name>` (optional) is provided, a subtopic is composed with that name
- A common use is to omit topic and just use `name`, allowing for human-readable topics

You will receive a JSON like the below:

```
{
  "feed": {
    "topic": "0x6a617669000000000000000000000000000000000000000000000000000000000000",
    "user": "0xdfa2db618eacbfe84e94a71dda2492240993c45b"
  },
  "epoch": {
    "level": 16,
    "time": 1534237239
  }
  "protocolVersion" : 0,
}
```

2. Post the update

Extract the fields out of the JSON and build a query string as below:

`POST /bzz-feed:/?topic=<TOPIC>&user=<USER>&level=<LEVEL>&time=<TIME>&signature=<SIGNATUR`

Where:

- `topic`: Feed topic, as specified above
- `user`: your Ethereum address
- `level`: Suggested frequency level retrieved in the JSON above
- `time`: Suggested timestamp retrieved in the JSON above
- `protocolVersion`: Feeds protocol version. Currently `0`
- `signature`: Signature, hex encoded. See below on how to calclulate the signature
- Request posted data: binary stream with the update data

Reading a Feed

To retrieve a Feed's last update:

`GET /bzz-feed:/?topic=<TOPIC>&user=<USER>`

`GET /bzz-feed:/<MANIFEST OR ENS NAME>`

Note:

- Again, if `topic` is omitted, it is assumed to be zero, 0x000...
- If `name=<name>` is provided, a subtopic is composed with that name
- A common use is to omit `topic` and just use `name`, allowing for human-readable topics, for example: `GET /bzz-feed:/?name=profile-picture&user=<USER>`

To get a previous update:

Add an addtional `time` parameter. The last update before that `time` (unix time) will be looked up.

`GET /bzz-feed:/?topic=<TOPIC>&user=<USER>&time=<T>`

`GET /bzz-feed:/<MANIFEST OR ENS NAME>?time=<T>`

Creating a Feed Manifest

To create a `Feed manifest` using the HTTP API:

`POST /bzz-feed:/?topic=<TOPIC>&user=<USER>&manifest=1`. With an empty body.

This will create a manifest referencing the provided Feed.

Note: This API call will be deprecated in the near future.

3.5.4.3 Go API

Query object

The `Query` object allows you to build a query to browse a particular `Feed`.

The default `Query`, obtained with `feed.NewQueryLatest()` will build a `Query` that retrieves the latest update of the given `Feed`.

You can also use `feed.NewQuery()` instead, if you want to build a `Query` to look up an update before a certain date.

Advanced usage of `Query` includes hinting the lookup algorithm for faster lookups. The default hint `lookup.NoClue` will have your node track Feeds you query frequently and handle hints automatically.

Request object

The `Request` object makes it easy to construct and sign a request to Swarm to update a particular Feed. It contains methods to sign and add data. We can manually build the `Request` object, or fetch a valid "template" to use for the update.

A `Request` can also be serialized to JSON in case you need your application to delegate signatures, such as having a browser sign a Feed update request.

Posting to a Feed

1. Retrieve a `Request` object or build one from scratch. To retrieve a ready-to-sign one:

```
func (c *Client) GetFeedRequest(query *feed.Query, manifestAddressOrDomain string)␣
↪(*feed.Request, error)
```

2. Use `Request.SetData()` and `Request.Sign()` to load the payload data into the request and sign it
3. Call `UpdateFeed()` with the filled `Request`:

```
func (c *Client) UpdateFeed(request *feed.Request, createManifest bool) (io.
↪ReadCloser, error)
```

Reading a Feed

To retrieve a Feed update, use *client.QueryFeed()*. `QueryFeed` returns a byte stream with the raw content of the Feed update.

```
func (c *Client) QueryFeed(query *feed.Query, manifestAddressOrDomain string) (io.
↪ReadCloser, error)
```

`manifestAddressOrDomain` is the address you obtained in `CreateFeedWithManifest` or an `ENS` domain whose Resolver points to that address. `query` is a Query object, as defined above.

You only need to provide either `manifestAddressOrDomain` or `Query` to `QueryFeed()`. Set to `""` or `nil` respectively.

Creating a Feed Manifest

Swarm client (package swarm/api/client) has the following method:

```
func (c *Client) CreateFeedWithManifest(request *feed.Request) (string, error)
```

`CreateFeedWithManifest` uses the `request` parameter to set and create a `Feed manifest`.

Returns the resulting `Feed manifest address` that you can set in an ENS Resolver (setContent) or reference future updates using `Client.UpdateFeed()`

Example Go code

```
// Build a `Feed` object to track a particular user's updates
f := new(feed.Feed)
f.User = signer.Address()
f.Topic, _ = feed.NewTopic("weather",nil)

// Build a `Query` to retrieve a current Request for this feed
query := feeds.NewQueryLatest(&f, lookup.NoClue)

// Retrieve a ready-to-sign request using our query
// (queries can be reused)
request, err := client.GetFeedRequest(query, "")
if err != nil {
    utils.Fatalf("Error retrieving feed status: %s", err.Error())
}

// set the new data
request.SetData([]byte("Weather looks bright and sunny today, we should merge this␣
↪PR and go out enjoy"))

// sign update
if err = request.Sign(signer); err != nil {
    utils.Fatalf("Error signing feed update: %s", err.Error())
}

// post update
err = client.UpdateFeed(request)
if err != nil {
    utils.Fatalf("Error updating feed: %s", err.Error())
}
```

Command-Line

The CLI API allows us to go through how Feeds work using practical examples. You can look up CL usage by typing `swarm feed` into your CLI.

In the CLI examples, we will create and update feeds using the bzzapi on a running local Swarm node that listens by default on port 8500.

Creating a Feed Manifest

The Swarm CLI allows creating Feed Manifests directly from the console.

`swarm feed create` is defined as a command to create and publish a `Feed manifest`.

The feed topic can be built in the following ways:

- use `--topic` to set the topic to an arbitrary binary hex string.
- **use `--name` to set the topic to a human-readable name.** For example, `--name` could be set to "profile-picture", meaning this feed allows to get this user's current profile picture.
- **use both `--topic` and `--name` to create named subtopics.** For example, *–topic* could be set to an Ethereum contract address and `--name` could be set to "comments", meaning this feed tracks a discussion about that contract.

The `--user` flag allows to have this manifest refer to a user other than yourself. If not specified, it will then default to your local account (`--bzzaccount`).

If you don't specify a name or a topic, the topic will be set to `0 hex` and name will be set to your username.

```
$ swarm --bzzapi http://localhost:8500 feed create --name test
```

creates a feed named "test". This is equivalent to the HTTP API way of

```
$ swarm --bzzapi http://localhost:8500 feed create --topic 0x74657374
```

since `test string == 0x74657374 hex`. Name and topic are interchangeable, as long as you don't specify both.

`feed create` will return the **feed manifest**.

You can also use `curl` in the HTTP API, but, here, you have to explicitly define the user (which, in this case, is your account) and the manifest.

```
$ curl -XPOST -d 'name=test&user=<your account>&manifest=1' http://localhost:8500/
↪bzz-Feed:/
```

is equivalent to

```
$ curl -XPOST -d 'topic=0x74657374&user=<your account>&manifest=1' http://
↪localhost:8500/bzz-Feed:/
```

Posting to a Feed

To update a Feed with the CLI, use `feed update`. The **update** argument has to be in `hex`. If you want to update your *test* feed with the update *hello*, you can refer to it by name:

```
$ swarm --bzzapi http://localhost:8500 feed update --name test 0x68656c6c6f203
```

You can also refer to it by topic,

```
$ swarm --bzzapi http://localhost:8500 feed update --topic 0x74657374␣
↪0x68656c6c6f203
```

or manifest.

```
$ swarm --bzzapi http://localhost:8500 feed update --manifest <manifest hash>␣
↪0x68656c6c6f203
```

Reading Feed status

You can read the feed object using `feed info`. Again, you can use the feed name, the topic, or the manifest hash. Below, we use the name.

```
$ swarm --bzzapi http://localhost:8500 feed info --name test
```

Reading Feed Updates

Although the Swarm CLI doesn't have the functionality to retrieve feed updates, we can use `curl` and the HTTP api to retrieve them. Again, you can use the feed name, topic, or manifest hash. To return the update `hello` for your `test` feed, do this:

```
$ curl 'http://localhost:8500/bzz-feed:/?user=<your address>&name=test'
```

Computing Feed Signatures

1. computing the digest:

The digest is computed concatenating the following:

- 1-byte protocol version (currently 0)
- 7-bytes padding, set to 0
- 32-bytes topic
- 20-bytes user address
- 7-bytes time, little endian
- 1-byte level
- payload data (variable length)

2. Take the SHA3 hash of the above digest
3. Compute the ECDSA signature of the hash
4. Convert to hex string and put in the `signature` field above

3.5.4.4 JavaScript example

```
var web3 = require("web3");

if (module !== undefined) {
  module.exports = {
    digest: feedUpdateDigest
  }
}

var topicLength = 32;
var userLength = 20;
var timeLength = 7;
var levelLength = 1;
var headerLength = 8;
var updateMinLength = topicLength + userLength + timeLength + levelLength +␣
↪headerLength;




function feedUpdateDigest(request /*request*/, data /*UInt8Array*/) {
  var topicBytes = undefined;
    var userBytes = undefined;
    var protocolVersion = 0;

    protocolVersion = request.protocolVersion

  try {
    topicBytes = web3.utils.hexToBytes(request.feed.topic);
  } catch(err) {
    console.error("topicBytes: " + err);
    return undefined;
  }

  try {
    userBytes = web3.utils.hexToBytes(request.feed.user);
  } catch(err) {
```

(continues on next page)

(continued from previous page)

```
    console.error("topicBytes: " + err);
    return undefined;
  }

  var buf = new ArrayBuffer(updateMinLength + data.length);
  var view = new DataView(buf);
    var cursor = 0;

    view.setUint8(cursor, protocolVersion) // first byte is protocol version.
    cursor+=headerLength; // leave the next 7 bytes (padding) set to zero

  topicBytes.forEach(function(v) {
    view.setUint8(cursor, v);
    cursor++;
  });

  userBytes.forEach(function(v) {
    view.setUint8(cursor, v);
    cursor++;
  });

  // time is little-endian
  view.setUint32(cursor, request.epoch.time, true);
  cursor += 7;

  view.setUint8(cursor, request.epoch.level);
  cursor++;

  data.forEach(function(v) {
    view.setUint8(cursor, v);
    cursor++;
    });
    console.log(web3.utils.bytesToHex(new Uint8Array(buf)))

  return web3.utils.sha3(web3.utils.bytesToHex(new Uint8Array(buf)));
}

// data payload
data = new Uint8Array([5,154,15,165,62])

// request template, obtained calling http://localhost:8500/bzz-feed:/?user=
↪<0xUSER>&topic=<0xTOPIC>&meta=1
request = {"feed":{"topic":
↪"0x1234123412341234123412341234123412341234123412341234123412341234","user":
↪"0xabcdefabcdefabcdefabcdefabcdefabcdefabcd"},"epoch":{"time":1538650124,"level
↪":25},"protocolVersion":0}

// obtain digest
digest = feedUpdateDigest(request, data)

console.log(digest)
```

3.5.4.5 Manifests

In general manifests declare a list of strings associated with Swarm hashes. A manifest matches to exactly one hash, and it consists of a list of entries declaring the content which can be retrieved through that hash. This is demonstrated by the following example:

Let's create a directory containing the two orange papers and an html index file listing the two pdf documents.

```
$ ls -1 orange-papers/
index.html
smash.pdf
sw^3.pdf

$ cat orange-papers/index.html
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8">
  </head>
  <body>
    <ul>
      <li>
        <a href="./sw^3.pdf">Viktor Trón, Aron Fischer, Dániel Nagy A and Zsolt␣
↪Felföldi, Nick Johnson: swap, swear and swindle: incentive system for swarm.</a>␣
↪ May 2016
      </li>
      <li>
        <a href="./smash.pdf">Viktor Trón, Aron Fischer, Nick Johnson: smash-
↪proof: auditable storage for swarm secured by masked audit secret hash.</a> May␣
↪2016
      </li>
    </ul>
  </body>
</html>
```

We now use the `swarm up` command to upload the directory to Swarm to create a mini virtual site.

Note: In this example we are using the public gateway through the *bzz-api* option in order to upload. The examples below assume a node running on localhost to access content. Make sure to run a local node to reproduce these examples.

```
$ swarm --recursive --defaultpath orange-papers/index.html --bzzapi http://swarm-
↪gateways.net/ up orange-papers/ 2> up.log
> 2477cc8584cc61091b5cc084cdcdb45bf3c6210c263b0143f030cf7d750e894d
```

The returned hash is the hash of the manifest for the uploaded content (the orange-papers directory):

We now can get the manifest itself directly (instead of the files they refer to) by using the bzz-raw protocol `bzz-raw`:

```
$ wget -O- "http://localhost:8500/bzz-raw:/
↪2477cc8584cc61091b5cc084cdcdb45bf3c6210c263b0143f030cf7d750e894d"

> {
  "entries": [
    {
      "hash": "4b3a73e43ae5481960a5296a08aaae9cf466c9d5427e1eaa3b15f600373a048d",
      "contentType": "text/html; charset=utf-8"
    },
    {
      "hash": "4b3a73e43ae5481960a5296a08aaae9cf466c9d5427e1eaa3b15f600373a048d",
      "contentType": "text/html; charset=utf-8",
      "path": "index.html"
    },
    {
      "hash": "69b0a42a93825ac0407a8b0f47ccdd7655c569e80e92f3e9c63c28645df3e039",
      "contentType": "application/pdf",
      "path": "smash.pdf"
```

(continues on next page)

(continued from previous page)

```
    },
    {
      "hash": "6a18222637cafb4ce692fa11df886a03e6d5e63432c53cbf7846970aa3e6fdf5",
      "contentType": "application/pdf",
      "path": "sw^3.pdf"
    }
  ]
}
```

Note: macOS users can install wget via homebrew (or use curl).

Manifests contain content_type information for the hashes they reference. In other contexts, where content_type is not supplied or, when you suspect the information is wrong, it is possible to specify the content_type manually in the search query. For example, the manifest itself should be *text/plain*:

```
http://localhost:8500/bzz-raw:/
↪2477cc8584cc61091b5cc084cdcdb45bf3c6210c263b0143f030cf7d750e894d?content_type=
↪"text/plain"
```

Now you can also check that the manifest hash matches the content (in fact, Swarm does this for you):

```
$ wget -O- http://localhost:8500/bzz-raw:/
↪2477cc8584cc61091b5cc084cdcdb45bf3c6210c263b0143f030cf7d750e894d?content_type=
↪"text/plain" > manifest.json

$ swarm hash manifest.json
> 2477cc8584cc61091b5cc084cdcdb45bf3c6210c263b0143f030cf7d750e894d
```

Path Matching

A useful feature of manifests is that we can match paths with URLs. In some sense this makes the manifest a routing table and so the manifest acts as if it was a host.

More concretely, continuing in our example, when we request:

```
GET http://localhost:8500/bzz:/
↪2477cc8584cc61091b5cc084cdcdb45bf3c6210c263b0143f030cf7d750e894d/sw^3.pdf
```

Swarm first retrieves the document matching the manifest above. The url path `sw^3` is then matched against the entries. In this case a perfect match is found and the document at 6a182226... is served as a pdf.

As you can see the manifest contains 4 entries, although our directory contained only 3. The extra entry is there because of the `--defaultpath orange-papers/index.html` option to `swarm up`, which associates the empty path with the file you give as its argument. This makes it possible to have a default page served when the url path is empty. This feature essentially implements the most common webserver rewrite rules used to set the landing page of a site served when the url only contains the domain. So when you request

```
GET http://localhost:8500/bzz:/
↪2477cc8584cc61091b5cc084cdcdb45bf3c6210c263b0143f030cf7d750e894d/
```

you get served the index page (with content type `text/html`) at `4b3a73e43ae5481960a5296a08aaae9cf466c9d5427e1eaa3b15f600373a048d`.

Paths and directories

Swarm manifests don't "break" like a file system. In a file system, the directory matches at the path separator (/ in linux) at the end of a directory name:

```
-- dirname/
----subdir1/
------subdir1file.ext
------subdir2file.ext
----subdir2/
------subdir2file.ext
```

In Swarm, path matching does not happen on a given path separator, but **on common prefixes**. Let's look at an example: The current manifest for the `theswarm.eth` homepage is as follows:

```
wget -O- "http://swarm-gateways.net/bzz-raw:/theswarm.eth/ > manifest.json

> {"entries":[{"hash":
↪"ee55bc6844189299a44e4c06a4b7fbb6d66c90004159c67e6c6d010663233e26","path":
↪"LICENSE","mode":420,"size":1211,"mod_time":"2018-06-12T15:36:29Z"},
            {"hash":
↪"57fc80622275037baf4a620548ba82b284845b8862844c3f56825ae160051446","path":
↪"README.md","mode":420,"size":96,"mod_time":"2018-06-12T15:36:29Z"},
            {"hash":
↪"8919df964703ccc81de5aba1b688ff1a8439b4460440a64940a11e1345e453b5","path":"Swarm_
↪files/","contentType":"application/bzz-manifest+json","mod_time":"0001-01-
↪01T00:00:00Z"},
            {"hash":
↪"acce5ad5180764f1fb6ae832b624f1efa6c1de9b4c77b2e6ec39f627eb2fe82c","path":"css/",
↪"contentType":"application/bzz-manifest+json","mod_time":"0001-01-01T00:00:00Z"},
            {"hash":
↪"0a000783e31fcf0d1b01ac7d7dae0449cf09ea41731c16dc6cd15d167030a542","path":
↪"ethersphere/orange-papers/","contentType":"application/bzz-manifest+json","mod_
↪time":"0001-01-01T00:00:00Z"},
            {"hash":
↪"b17868f9e5a3bf94f955780e161c07b8cd95cfd0203d2d731146746f56256e56","path":"f",
↪"contentType":"application/bzz-manifest+json","mod_time":"0001-01-01T00:00:00Z"},
            {"hash":
↪"977055b5f06a05a8827fb42fe6d8ec97e5d7fc5a86488814a8ce89a6a10994c3","path":"i",
↪"contentType":"application/bzz-manifest+json","mod_time":"0001-01-01T00:00:00Z"},
            {"hash":
↪"48d9624942e927d660720109b32a17f8e0400d5096c6d988429b15099e199288","path":"js/",
↪"contentType":"application/bzz-manifest+json","mod_time":"0001-01-01T00:00:00Z"},
            {"hash":
↪"294830cee1d3e63341e4b34e5ec00707e891c9e71f619bc60c6a89d1a93a8f81","path":"talks/
↪","contentType":"application/bzz-manifest+json","mod_time":"0001-01-01T00:00:00Z
↪"},
            {"hash":
↪"12e1beb28d86ed828f9c38f064402e4fac9ca7b56dab9cf59103268a62a2b35f","contentType":
↪"text/html; charset=utf-8","mode":420,"size":31371,"mod_time":"2018-06-
↪12T15:36:29Z"}
  ]}
```

Note the `path` for entry `b17868...`: It is `f`. This means, there are more than one entries for this manifest which start with an *f*, and all those entries will be retrieved by requesting the hash `b17868...` and through that arrive at the matching manifest entry:

```
$ wget -O- http://localhost:8500/bzz-raw:/
↪b17868f9e5a3bf94f955780e161c07b8cd95cfd0203d2d731146746f56256e56/

{"entries":[{"hash":
↪"25e7859eeb7366849f3a57bb100ff9b3582caa2021f0f55fb8fce9533b6aa810","path":
↪"avicon.ico","mode":493,"size":32038,"mod_time":"2018-06-12T15:36:29Z"},
```

(continues on next page)

(continued from previous page)

```
            {"hash":
↪"97cfd23f9e36ca07b02e92dc70de379a49be654c7ed20b3b6b793516c62a1a03","path":"onts/
↪glyphicons-halflings-regular.","contentType":"application/bzz-manifest+json",
↪"mod_time":"0001-01-01T00:00:00Z"}
 ]}
```

So we can see that the `f` entry in the root hash resolves to a manifest containing `avicon.ico` and `onts/glyphicons-halflings-regular`. The latter is interesting in itself: its `content_type` is `application/bzz-manifest+json`, so it points to another manifest. Its `path` also does contain a path separator, but that does not result in a new manifest after the path separator like a directory (e.g. at `onts/`). The reason is that on the file system on the hard disk, the `fonts` directory only contains *one* directory named `glyphicons-halflings-regular`, thus creating a new manifest for just `onts/` would result in an unnecessary lookup. This general approach has been chosen to limit unnecessary lookups that would only slow down retrieval, and manifest "forks" happen in order to have the logarythmic bandwidth needed to retrieve a file in a directory with thousands of files.

When requesting `wget -O- "http://swarm-gateways.net/bzz-raw:/theswarm.eth/favicon.ico`, Swarm will first retrieve the manifest at the root hash, match on the first `f` in the entry list, resolve the hash for that entry and finally resolve the hash for the `favicon.ico` file.

For the `theswarm.eth` page, the same applies to the `i` entry in the root hash manifest. If we look up that hash, we'll find entries for `mages/` (a further manifest), and `ndex.html`, whose hash resolves to the main `index.html` for the web page.

Paths like `css/` or `js/` get their own manifests, just like common directories, because they contain several files.

Note: If a request is issued which Swarm can not resolve unambiguosly, a `300 "Multiplce Choices"` HTTP status will be returned. In the example above, this would apply for a request for `http://swarm-gateways.net/bzz:/theswarm.eth/i`, as it could match both `images/` as well as `index.html`

3.5.4.6 Encryption

Introduced in POC 0.3, symmetric encryption is now readily available to be used with the `swarm up` upload command. The encryption mechanism is meant to protect your information and make the chunked data unreadable to any handling Swarm node.

Swarm uses Counter mode encryption to encrypt and decrypt content. When you upload content to Swarm, the uploaded data is split into 4 KB chunks. These chunks will all be encrypted with a separate randomly generated encryption key. The encryption happens on your local Swarm node, unencrypted data is not shared with other nodes. The reference of a single chunk (and the whole content) will be the concatenation of the hash of encoded data and the decryption key. This means the reference will be longer than the standard unencrypted Swarm reference (64 bytes instead of 32 bytes).

When your node syncs the encrypted chunks of your content with other nodes, it does not share the full references (or the decryption keys in any way) with the other nodes. This means that other nodes will not be able to access your original data, moreover they will not be able to detect whether the synchronized chunks are encrypted or not.

When your data is retrieved it will only get decrypted on your local Swarm node. During the whole retrieval process the chunks traverse the network in their encrypted form, and none of the participating peers are able to decrypt them. They are only decrypted and assembled on the Swarm node you use for the download.

More info about how we handle encryption at Swarm can be found here.

Note: Swarm currently supports both encrypted and unencrypted `swarm up` commands through

usage of the `--encrypt` flag. This might change in the future as we will refine and make Swarm a safer network.

Important: The encryption feature is non-deterministic (due to a random key generated on every upload request) and users of the API should not rely on the result being idempotent; thus uploading the same content twice to Swarm with encryption enabled will not result in the same reference.

Example usage:

First, we create a simple test file.

```
$ echo "testfile" > mytest.txt
```

We upload the test file **without** encryption,

```
$ swarm up mytest.txt
> <file reference>
```

and **with** encryption.

```
$ swarm up --encrypt mytest.txt
> <encrypted reference>
```

Note that the reference of the encrypted upload is **longer** than that of the unencrypted upload. Note also that, because of the random encryption key, repeating the encrypted upload results in a different reference:

```
$ swarm up --encrypt mytest.txt
<another encrypted reference>
```

3.5.4.7 Access Control

Swarm supports restricting access to content through several access control strategies:

- Password protection - where a number of undisclosed parties can access content using a shared secret `(pass, act)`
- Selective access using Elliptic Curve key-pairs:
  - For an undisclosed party - where only one grantee can access the content `(pk)`
  - For a number of undisclosed parties - where every grantee can access the content `(act)`

Creating access control for content is currently supported only through CLI usage.

Accessing restricted content is available through CLI and HTTP. When accessing content which is restricted by a password HTTP Basic access authentication can be used out-of-the-box.

Important: When accessing content which is restricted to certain EC keys - the node which exposes the HTTP proxy that is queried must be started with the granted private key as its `bzzaccount` CLI parameter.

Password protection

The simplest type of credential is a passphrase. In typical use cases, the passphrase is distributed by off-band means, with adequate security measures. Any user that knows the passphrase can access the content.

When using password protection, a given content reference (e.g.: a given Swarm manifest address or, alternatively, a Mutable Resource address) is encrypted using scrypt with a given passphrase and a random salt. The encrypted reference and the salt are then embedded into an unencrypted manifest which can be freely distributed but only accessed by undisclosed parties that posses knowledge of the passphrase.

Password protection can also be used for selective access when using the `act` strategy - similarly to granting access to a certain EC key access can be also given to a party identified by a password. In fact, one could also create an `act` manifest that solely grants access to grantees through passwords, without the need to know their public keys.

Example usage:

Important: Restricting access to content on Swarm is a 2-step process - you first upload your content, then wrap the reference with an access control manifest. **We recommend that you always upload your content with encryption enabled**. In the following examples we will refer the uploaded content hash as `reference hash`

First, we create a simple test file. We upload it to Swarm (with encryption).

```
$ echo "testfile" > mytest.txt
$ swarm up --encrypt mytest.txt
> <reference hash>
```

Then, for the sake of this example, we create a file with our password in it.

```
$ echo "mypassword" > mypassword.txt
```

This password will protect the access-controlled content that we upload. We can refer to this password using the *–password* flag. The password file should contain the password in plaintext.

The `swarm access` command sets a new password using the `new pass` argument. It expects you to input the password file and the uploaded Swarm content hash you'd like to limit access to.

```
$ swarm access new pass --password mypassword.txt <reference hash>
> <reference of access controlled manifest>
```

The returned hash is the hash of the access controlled manifest.

When requesting this hash through the HTTP gateway you should receive an `HTTP Unauthorized 401` error:

```
$ curl http://localhost:8500/bzz:/<reference of access controlled manifest>/
> Code: 401
> Message: cant decrypt - forbidden
> Timestamp: XXX
```

You can retrieve the content in three ways:

1. The same request should make an authentication dialog pop-up in the browser. You could then input the password needed and the content should correctly appear. (Leave the username empty.)
2. Requesting the same hash with HTTP basic authentication would return the content too. `curl` needs you to input a username as well as a password, but the former can be an arbitrary string (here, it's `x`).

```
$ curl http://x:mypassword@localhost:8500/bzz:/<reference of access controlled␣
↪manifest>/
```

3. You can also use `swarm down` with the `--password` flag.

```
$ swarm  --password mypassword.txt down bzz:/<reference of access controlled␣
↪manifest>/ mytest2.txt
$ cat mytest2.txt
> testfile
```

Selective access using EC keys

A more sophisticated type of credential is an Elliptic Curve private key, identical to those used throughout Ethereum for accessing accounts.

In order to obtain the content reference, an Elliptic-curve Diffie–Hellman *(ECDH)* key agreement needs to be performed between a provided EC public key (that of the content publisher) and the authorized key, after which the undisclosed authorized party can decrypt the reference to the access controlled content.

Whether using access control to disclose content to a single party (by using the `pk` strategy) or to multiple parties (using the `act` strategy), a third unauthorized party cannot find out the identity of the authorized parties. The third party can, however, know the number of undisclosed grantees to the content. This, however, can be mitigated by adding bogus grantee keys while using the `act` strategy in cases where masking the number of grantees is necessary. This is not the case when using the `pk` strategy, as it as by definition an agreement between two parties and only two parties (the publisher and the grantee).

Important: Accessing content which is access controlled is enabled only when using a *local* Swarm node (e.g. running on *localhost*) in order to keep your data, passwords and encryption keys safe. This is enforced through an in-code guard.

Danger: NEVER (EVER!) use an external gateway to upload or download access controlled content as you will be putting your privacy at risk! You have been fairly warned!

Protecting content with Elliptic curve keys (single grantee):

The `pk` strategy requires a `bzzaccount` to encrypt with. The most comfortable option in this case would be the same `bzzaccount` you normally start your Swarm node with - this will allow you to access your content seamlessly through that node at any given point in time.

Grantee public keys are expected to be in an *secp256 compressed* form - 66 characters long string (an example would be `02e6f8d5e28faaa899744972bb847b6eb805a160494690c9ee7197ae9f619181db`). Comments and other characters are not allowed.

```
$ swarm --bzzaccount <your account> access new pk --grant-key <your public key>
↪<reference hash>
> <reference of access controlled manifest>
```

The returned hash `4b964a75ab19db960c274058695ca4ae21b8e19f03ddf1be482ba3ad3c5b9f9b` is the hash of the access controlled manifest.

The only way to fetch the access controlled content in this case would be to request the hash through one of the nodes that were granted access and/or posses the granted private key (and that the requesting node has been started with the appropriate `bzzaccount` that is associated with the relevant key) - either the local node that was used to upload the content or the node which was granted access through its public key.

Protecting content with Elliptic curve keys and passwords (multiple grantees):

The `act` strategy also requires a `bzzaccount` to encrypt with. The most comfortable option in this case would be the same `bzzaccount` you normally start your Swarm node with - this will allow you

to access your content seamlessly through that node at any given point in time

---

Note: the `act` strategy expects a grantee public-key list and/or a list of permitted passwords to be communicated to the CLI. This is done using the `--grant-keys` flag and/or the `--password` flag. Grantee public keys are expected to be in an *secp256 compressed* form - 66 characters long string (e.g. `02e6f8d5e28faaa899744972bb847b6eb805a160494690c9ee7197ae9f619181db`). Each grantee should appear in a separate line. Passwords are also expected to be line-separated. Comments and other characters are not allowed.

```
swarm --bzzaccount 2f1cd699b0bf461dcfbf0098ad8f5587b038f0f1 access new act --grant-
↪keys /path/to/public-keys/file --password /path/to/passwords/file  <reference␣
↪hash>
4b964a75ab19db960c274058695ca4ae21b8e19f03ddf1be482ba3ad3c5b9f9b
```

The returned hash `4b964a75ab19db960c274058695ca4ae21b8e19f03ddf1be482ba3ad3c5b9f9b` is the hash of the access controlled manifest.

As with the `pk` strategy - the only way to fetch the access controlled content in this case would be to request the hash through one of the nodes that were granted access and/or posses the granted private key (and that the requesting node has been started with the appropriate `bzzaccount` that is associated with the relevant key) - either the local node that was used to upload the content or one of the nodes which were granted access through their public keys.

HTTP usage

Accessing restricted content on Swarm through the HTTP API is, as mentioned, limited to your local node due to security considerations. Whenever requesting a restricted resource without the proper credentials via the HTTP proxy, the Swarm node will respond with an `HTTP 401 Unauthorized` response code.

When accessing password protected content:

When accessing a resource protected by a passphrase without the appropriate credentials the browser will receive an `HTTP 401 Unauthorized` response and will show a pop-up dialog asking for a username and password. For the sake of decrypting the content - only the password input in the dialog matters and the username field can be left blank.

The credentials for accessing content protected by a password can be provided in the initial request in the form of: `http://x:<password>@localhost:8500/bzz:/<hash or ens name>` (`curl` needs you to input a username as well as a password, but the former can be an arbitrary string (here, it's `x`).)

Important: Access controlled content should be accessed through the `bzz://` protocol

---

When accessing EC key protected content:

When accessing a resource protected by EC keys, the node that requests the content will try to decrypt the restricted content reference using its **own** EC key which is associated with the current *bzz account* that the node was started with (see the `--bzzaccount` flag). If the node's key is granted access - the content will be decrypted and displayed, otherwise - an `HTTP 401 Unauthorized` error will be returned by the node.

Access control in the CLI: example usage

Passwords

First, we create a simple test file. We upload it to Swarm using encryption.

```
$ echo "testfile" > mytest.txt
$ swarm up  --encrypt mytest.txt
> <reference hash>
```

Then, we define a password file and use it to create an access-controlled manifest.

```
$ echo "mypassword" > mypassword.txt
$ swarm access new pass --password mypassword.txt <reference hash>
> <reference of access controlled manifest>
```

We can create a passwords file with one password per line in plaintext (`password1` is probably not a very good password).

```
$ for i in {1..3}; do echo -e password$i; done > mypasswords.txt
$ cat mypasswords.txt
> password1
> password2
> password3
```

Then, we point to this list while wrapping our manifest.

```
$ swarm access new act --password mypasswords.txt <reference hash>
> <reference of access controlled manifest>
```

We can access the returned manifest using any of the passwords in the password list.

```
$ echo password1 > password1.txt
$ swarm --password1.txt down bzz:/<reference of access controlled manifest>
```

We can also *curl* it.

```
$ curl http://:password1@localhost:8500/bzz:/<reference of access controlled␣
↪manifest>/
```

Elliptic curve keys

1. `pk` strategy

First, we create a simple test file. We upload it to Swarm using encryption.

```
$ echo "testfile" > mytest.txt
$ swarm up --encrypt mytest.txt
> <reference hash>
```

Then, we draw an EC key pair and use the public key to create the access-controlled manifest.

```
$ swarm access new pk --grant-key <public key> <reference hash>
> <reference of access controlled manifest>
```

We can retrieve the access-controlled manifest via a node that has the private key. You can add a private key using `geth` (see here).

```
$ swarm --bzzaccount <address of node with granted private key> down bzz:/
↪<reference of access controlled manifest> out.txt
$ cat out.txt
> "testfile"
```

2. `act` strategy

We can also supply a list of public keys to create the access-controlled manifest.

```
$ swarm access new act --grant-keys <public key list> <reference hash>
> <reference of access controlled manifest>
```

Again, only nodes that possess the private key will have access to the content.

```
$ swarm --bzzaccount <address of node with a granted private key> down bzz:/
↪<reference of access controlled manifest> out.txt
$ cat out.txt
> "testfile"
```

3.5.4.8 FUSE

Another way of interacting with Swarm is by mounting it as a local filesystem using FUSE (Filesystem in Userspace). There are three IPC API's which help in doing this.

Note: FUSE needs to be installed on your Operating System for these commands to work. Windows is not supported by FUSE, so these command will work only in Linux, Mac OS and FreeBSD. For installation instruction for your OS, see "Installing FUSE" section below.

Installing FUSE

1. Linux (Ubuntu)

```
$ sudo apt-get install fuse
$ sudo modprobe fuse
$ sudo chown <username>:<groupname> /etc/fuse.conf
$ sudo chown <username>:<groupname> /dev/fuse
```

2. Mac OS

   Either install the latest package from https://osxfuse.github.io/ or use brew as below

```
$ brew update
$ brew install caskroom/cask/brew-cask
$ brew cask install osxfuse
```

CLI Usage

The Swarm CLI now integrates commands to make FUSE usage easier and streamlined.

Note: When using FUSE from the CLI, we assume you are running a local Swarm node on your machine. The FUSE commands attach to the running node through *bzzd.ipc*

Mount

One use case to mount a Swarm hash via FUSE is a file sharing feature accessible via your local file system. Files uploaded to Swarm are then transparently accessible via your local file system, just as if they were stored locally.

To mount a Swarm resource, first upload some content to Swarm using the `swarm up <resource>` command. You can also upload a complete folder using `swarm --recursive up <directory>`. Once you get the returned manifest hash, use it to mount the manifest to a mount point (the mount point should exist on your hard drive):

```
$ swarm fs mount <manifest-hash> <mount-point>
```

For example:

```
$ swarm fs mount <manifest-hash> /home/user/swarmmount
```

Your running Swarm node terminal output should show something similar to the following in case the command returned successfuly:

```
Attempting to mount /path/to/mount/point
Serving 6e4642148d0a1ea60e36931513f3ed6daf3deb5e499dcf256fa629fbc22cf247 at /path/
↪to/mount/point
Now serving swarm FUSE FS
↪manifest=6e4642148d0a1ea60e36931513f3ed6daf3deb5e499dcf256fa629fbc22cf247
↪mountpoint=/path/to/mount/point
```

You may get a "Fatal: had an error calling the RPC endpoint while mounting: context deadline exceeded" error if it takes too long to retrieve the content.

In your OS, via terminal or file browser, you now should be able to access the contents of the Swarm hash at `/path/to/mount/point`, i.e. `ls /home/user/swarmmount`

Access

Through your terminal or file browser, you can interact with your new mount as if it was a local directory. Thus you can add, remove, edit, create files and directories just as on a local directory. Every such action will interact with Swarm, taking effect on the Swarm distributed storage. Every such action also will result **in a new hash** for your mounted directory. If you would unmount and remount the same directory with the previous hash, your changes would seem to have been lost (effectively you are just mounting the previous version). While you change the current mount, this happens under the hood and your mount remains up-to-date.

Unmount

To unmount a `swarmfs` mount, either use the List Mounts command below, or use a known mount point:

```
$ swarm fs unmount <mount-point>
> 41e422e6daf2f4b32cd59dc6a296cce2f8cce1de9f7c7172e9d0fc4c68a3987a
```

The returned hash is the latest manifest version that was mounted. You can use this hash to remount the latest version with the most recent changes.

List Mounts

To see all existing swarmfs mount points, use the List Mounts command:

```
$ swarm fs list
```

Example Output:

```
Found 1 swarmfs mount(s):
0:
        Mount point: /path/to/mount/point
        Latest Manifest:
↪6e4642148d0a1ea60e36931513f3ed6daf3deb5e499dcf256fa629fbc22cf247
        Start Manifest:
↪6e4642148d0a1ea60e36931513f3ed6daf3deb5e499dcf256fa629fbc22cf247
```

3.5.4.9 Pinning Content

When content is uploaded in Swarm, it gets chunked and scattered over the network for storage. In the original Swarm design, the nodes that store the chunks gets incentivised for by the SWAP, SWEAR and SWINDLE protocols. For now, the incentivisation protocols are yet to be implemented. As a result of that, nodes does not have use preference in storing content. The default model used now is First Come First Serve (FIFO). The effect of the above model is that the uploaded contents will disappear after few days depending upon the overall network storage capacity. This is a undesirable property of the network today. To overcome this issue until the incentivisation layer is fully operational, pinning contents is implemented now. Aanyone can now pin a content (Swarm collection or a RAW file) on a Swarm node. i.e. a copy of the pinned content will be stored locally permenantly. Even if the content in the network disappears, it can be accessed from the pinned server always.

Note: Pinned content will be available only from the Swarm node where it is pinned.

Pinning Content

Content can be pinned in two different ways. One is during the content upload and the other is thereafter.

1. Pinning during upload

   Add a header "x-swarm-pin" and set it to "true" when uploading content. This will upload the file and then pin it too. This method can be used for Tar, Multipart and RAW file uploads too.

```
curl -H "Content-Type: application/x-tar" -H "x-swarm-pin: true"  --data-binary␣
↪@files.tar http://localhost:8500/bzz:/
```

2. Pinning after upload

   If an already uploaded content needs to be pinned, the following HTTP API should be used.

```
# to pin a Swarm collection
POST /bzz-pin:/<MANIFEST OR ENS NAME>

# to pin a RAW file in Swarm
POST /bzz-pin:/<SWARM RAW FILE HASH>/?raw=true
```

Note: When pinning a already uploaded file, make sure that the entire file content is available locally by issuing a download once.

Unpinning Content

An already pinned file can be unpinned at anytime. Once the collection is unpinned, the contents will follow the FIFO rule and may be garbage collected in future.

```
DELETE /bzz-pin:/<MANIFEST OR ENS NAME OR SWARM RAW FILE HASH>
```

Listing Pinning Info

Pinned contents and their information can be viewed at any point using this API. Information includes The pinned hash, wether the pinned content is a collection or RAW file, the pinned content size in bytes and the no of time the content is pinned.

```
GET /bzz-pin:/

[
 { "Address"    :
↪"0x94f78a45c7897957809544aa6d68aa7ad35df695713895953b885aca274bd955",
   "IsRaw"      : "false",
   "FileSize"   : "12046",
   "PinCounter" : "2",
 },
 { "Address"    :
↪"0xccef599d1a13bed9989e424011aed2c023fce25917864cd7de38a761567410b8",
   "IsRaw"      : "true",
   "FileSize"   : "146",
   "PinCounter" : "5",
 },
]
```

Note: The information will be returned in json format shown above

3.5.4.10 Tags

Tags are meant as a complementary component to track the state of an upload on Swarm. Tags consist mainly of counters and their sole purpose is to track and expose information necessary to display the progress of your uploads. Whenever you upload content, a tag is automatically created in order to allow tracking of your upload in its various stages.

The tag API is supported through two different interfaces: CLI and HTTP.

CLI

The tag API can will displayed through the command line interface through the `swarm up` command. When uploading a file, using the `--progress` flag, the client will periodically use the tags API to retrieve the status of the upload:

```
$ echo "this is an example" > example.md
$ swarm up --progress example.md
  Swarm Hash: 730c96f6de2b5b3b961b3cf1ca0916efe2543a13a6da31e1083c61b08adc3602
  Tag UID: 672245080
  Upload status:
  Syncing 23 chunks       0s␣
↪[============================================================] 100 %
  Done! Your file is now retrievable from other Swarm nodes
```

HTTP

The tag API can be accessed by HTTP using the `GET` verb on the `bzz-tag:/` locator.

`bzz-tag` can track a tag by two parameters - either the Swarm hash or the tag UID of the upload.

You can find the tag UID of the upload in the data returned alongside the progress bar while using `swarm up`. Tag retrieval by UID is meant for a future implementation that would allow tracking an upload that has not returned a Swarm hash yet; i.e. it is still being split and stored on the local node.

When uploading to Swarm via HTTP `POST`, the tag UID will be returned as a header named `x-swarm-tag`. This allows for programmatic access to tags for Dapp developers that would like to display upload progress to their users.

The tag associated with a Swarm hash can be retrieved with the hash inlined after `bzz-tag:/` while the tag associated with the tag UID can be retrieved using the UID as a query variable.

Using the Swarm Hash:

```
$ curl localhost:8500/bzz-tag:/
↪730c96f6de2b5b3b961b3cf1ca0916efe2543a13a6da31e1083c61b08adc3602
{
 "Uid": 12210768,
 "Name": "Some upload",
 "Address": "730c96f6de2b5b3b961b3cf1ca0916efe2543a13a6da31e1083c61b08adc3602",
 "Total": 2,
 "Split": 2,
 "Seen": 2,
 "Stored": 2,
 "Sent": 0,
 "Synced": 0,
 "StartedAt": "2019-09-30T12:20:11.176316707+05:30"
}
```

Using the tag UID:

```
$ curl localhost:8500/bzz-tag:/&Id=12210768
{
 "Uid": 12210768,
 "Name": "Some upload",
 "Address": "730c96f6de2b5b3b961b3cf1ca0916efe2543a13a6da31e1083c61b08adc3602",
 "Total": 2,
 "Split": 2,
 "Seen": 2,
 "Stored": 2,
 "Sent": 0,
 "Synced": 0,
 "StartedAt": "2019-09-30T12:20:11.176316707+05:30"
}
```

3.5.4.11 BZZ URL schemes

Swarm offers 6 distinct URL schemes:

bzz

The bzz scheme assumes that the domain part of the url points to a manifest. When retrieving the asset addressed by the URL, the manifest entries are matched against the URL path. The entry with the longest matching path is retrieved and served with the content type specified in the corresponding manifest entry.

Example:

```
GET http://localhost:8500/bzz:/
↪2477cc8584cc61091b5cc084cdcdb45bf3c6210c263b0143f030cf7d750e894d/readme.md
```

returns a readme.md file if the manifest at the given hash address contains such an entry.

```
$ ls
readme.md
$ swarm --recursive up .
c4c81dbce3835846e47a83df549e4cad399c6a81cbf83234274b87d49f5f9020
$ curl http://localhost:8500/bzz-raw:/
↪c4c81dbce3835846e47a83df549e4cad399c6a81cbf83234274b87d49f5f9020/readme.md
```

(continues on next page)

(continued from previous page)

```
## Hello Swarm!

Swarm is awesome
```

If the manifest does not contain an file at `readme.md` itself, but it does contain multiple entries to which the URL could be resolved, e.g. in the example above, the manifest has entries for `readme.md.1` and `readme.md.2`, the API returns an HTTP response "300 Multiple Choices", indicating that the request could not be unambiguously resolved. A list of available entries is returned via HTTP or JSON.

```
$ ls
readme.md.1 readme.md.2
$ swarm --recursive up .
679bde3ccb6fb911db96a0ea1586c04899c6c0cc6d3426e9ee361137b270a463
$ curl -H "Accept:application/json" http://localhost:8500/bzz:/
↪679bde3ccb6fb911db96a0ea1586c04899c6c0cc6d3426e9ee361137b270a463/readme.md
{"Msg":"\u003ca href='/bzz:/
↪679bde3ccb6fb911db96a0ea1586c04899c6c0cc6d3426e9ee361137b270a463/readme.md.1
↪'\u003ereadme.md.1\u003c/a\u003e\u003cbr/\u003e\u003ca href='/bzz:/
↪679bde3ccb6fb911db96a0ea1586c04899c6c0cc6d3426e9ee361137b270a463/readme.md.2
↪'\u003ereadme.md.2\u003c/a\u003e\u003cbr/\u003e","Code":300,"Timestamp":"Fri, 15␣
↪Jun 2018 14:48:42 CEST","Details":""}
$ curl -H "Accept:application/json" http://localhost:8500/bzz:/
↪679bde3ccb6fb911db96a0ea1586c04899c6c0cc6d3426e9ee361137b270a463/readme.md | jq
{
    "Msg": "<a href='/bzz:/
↪679bde3ccb6fb911db96a0ea1586c04899c6c0cc6d3426e9ee361137b270a463/readme.md.1'>
↪readme.md.1</a><br/><a href='/bzz:/
↪679bde3ccb6fb911db96a0ea1586c04899c6c0cc6d3426e9ee361137b270a463/readme.md.2'>
↪readme.md.2</a><br/>",
    "Code": 300,
    "Timestamp": "Fri, 15 Jun 2018 14:49:02 CEST",
    "Details": ""
}
```

`bzz` scheme also accepts POST requests to upload content and create manifest for them in one go:

```
$ curl -H "Content-Type: text/plain" --data-binary "some-data" http://
↪localhost:8500/bzz:/
635d13a547d3252839e9e68ac6446b58ae974f4f59648fe063b07c248494c7b2%
$ curl http://localhost:8500/bzz:/
↪635d13a547d3252839e9e68ac6446b58ae974f4f59648fe063b07c248494c7b2/
some-data%
$ curl -H "Accept:application/json" http://localhost:8500/bzz-raw:/
↪635d13a547d3252839e9e68ac6446b58ae974f4f59648fe063b07c248494c7b2/ | jq .
{
    "entries": [
        {
            "hash":
↪"379f234c04ed1a18722e4c76b5029ff6e21867186c4dfc101be4f1dd9a879d98",
            "contentType": "text/plain",
            "mode": 420,
            "size": 9,
            "mod_time": "2018-06-15T15:46:28.835066044+02:00"
        }
    ]
}
```

bzz-raw

```
GET http://localhost:8500/bzz-raw:/
↪2477cc8584cc61091b5cc084cdcdb45bf3c6210c263b0143f030cf7d750e894d
```

When responding to GET requests with the bzz-raw scheme, Swarm does not assume that the hash resolves to a manifest. Instead it just serves the asset referenced by the hash directly. So if the hash actually resolves to a manifest, it returns the raw manifest content itself.

E.g. continuing the example in the `bzz` section above with `readme.md.1` and `readme.md.2` in the manifest:

```
$ curl http://localhost:8500/bzz-raw:/
↪679bde3ccb6fb911db96a0ea1586c04899c6c0cc6d3426e9ee361137b270a463/ | jq
{
    "entries": [
        {
        "hash": "efc6d4a7d7f0846973a321d1702c0c478a20f72519516ef230b63baa3da18c22",
        "path": "readme.md.",
        "contentType": "application/bzz-manifest+json",
        "mod_time": "0001-01-01T00:00:00Z"
        }
    ]
    }
$ curl http://localhost:8500/bzz-raw:/
↪efc6d4a7d7f0846973a321d1702c0c478a20f72519516ef230b63baa3da18c22/ | jq
{
    "entries": [
        {
            "hash":
↪"d0675100bc4580a0ad890b5d6f06310c0705d4ab1e796cfa1a8c597840f9793f",
            "path": "1",
            "mode": 420,
            "size": 33,
            "mod_time": "2018-06-15T14:21:32+02:00"
        },
        {
            "hash":
↪"f97cf36ac0dd7178c098f3661cd0402fcc711ff62b67df9893d29f1db35adac6",
            "path": "2",
            "mode": 420,
            "size": 35,
            "mod_time": "2018-06-15T14:42:06+02:00"
        }
    ]
    }
```

The `content_type` query parameter can be supplied to specify the MIME type you are requesting, otherwise content is served as an octet-stream per default. For instance if you have a pdf document (not the manifest wrapping it) at hash `6a182226...` then the following url will properly serve it.

```
GET http://localhost:8500/bzz-raw:/
↪6a18222637cafb4ce692fa11df886a03e6d5e63432c53cbf7846970aa3e6fdf5?content_
↪type=application/pdf
```

`bzz-raw` also supports POST requests to upload content to Swarm, the response is the hash of the uploaded content:

```
$ curl --data-binary "some-data" http://localhost:8500/bzz-raw:/
379f234c04ed1a18722e4c76b5029ff6e21867186c4dfc101be4f1dd9a879d98%
$ curl http://localhost:8500/bzz-raw:/
↪379f234c04ed1a18722e4c76b5029ff6e21867186c4dfc101be4f1dd9a879d98/
```

(continues on next page)

(continued from previous page)

```
some-data%
```

bzz-list

```
GET http://localhost:8500/bzz-list:/
↪2477cc8584cc61091b5cc084cdcdb45bf3c6210c263b0143f030cf7d750e894d/path
```

Returns a list of all files contained in <manifest> under <path> grouped into common prefixes using / as a delimiter. If no path is supplied, all files in manifest are returned. The response is a JSON-encoded object with `common_prefixes` string field and `entries` list field.

```
$ curl http://localhost:8500/bzz-list:/
↪679bde3ccb6fb911db96a0ea1586c04899c6c0cc6d3426e9ee361137b270a463/ | jq
{
    "entries": [
        {
            "hash":
↪"d0675100bc4580a0ad890b5d6f06310c0705d4ab1e796cfa1a8c597840f9793f",
            "path": "readme.md.1",
            "mode": 420,
            "size": 33,
            "mod_time": "2018-06-15T14:21:32+02:00"
        },
        {
            "hash":
↪"f97cf36ac0dd7178c098f3661cd0402fcc711ff62b67df9893d29f1db35adac6",
            "path": "readme.md.2",
            "mode": 420,
            "size": 35,
            "mod_time": "2018-06-15T14:42:06+02:00"
        }
    ]
    }
```

bzz-hash

```
GET http://localhost:8500/bzz-hash:/theswarm.eth/
```

Swarm accepts GET requests for bzz-hash url scheme and responds with the hash value of the raw content, the same content returned by requests with bzz-raw scheme. Hash of the manifest is also the hash stored in ENS so bzz-hash can be used for ENS domain resolution.

Response content type is *text/plain*.

```
$ curl http://localhost:8500/bzz-hash:/theswarm.eth/
7a90587bfc04ac4c64aeb1a96bc84f053d3d84cefc79012c9a07dd5230dc1fa4%
```

bzz-immutable

```
GET http://localhost:8500/bzz-immutable:/
↪2477cc8584cc61091b5cc084cdcdb45bf3c6210c263b0143f030cf7d750e894d
```

The same as the generic scheme but there is no ENS domain resolution, the domain part of the path needs to be a valid hash. This is also a read-only scheme but explicit in its integrity protection. A particular bzz-immutable url will always necessarily address the exact same fixed immutable content.

```
$ curl http://localhost:8500/bzz-immutable:/
↪679bde3ccb6fb911db96a0ea1586c04899c6c0cc6d3426e9ee361137b270a463/readme.md.1
## Hello Swarm!

Swarm is awesome
$ curl -H "Accept:application/json" http://localhost:8500/bzz-immutable:/theswarm.
↪eth/ | jq .
{
    "Msg": "cannot resolve theswarm.eth: immutable address not a content hash: \
↪"theswarm.eth\"",
    "Code": 404,
    "Timestamp": "Fri, 15 Jun 2018 13:22:27 UTC",
    "Details": ""
}
```

3.5.5 Swarm Messaging for DAPP-Developers

pss (Postal Service over Swarm) is a messaging protocol over Swarm with strong privacy features. The pss API is exposed through a JSON RPC interface described in the API Reference, here we explain the basic concepts and features.

With `pss` you can send messages to any node in the Swarm network. The messages are routed in the same manner as retrieve requests for chunks. Instead of chunk hash reference, `pss` messages specify a destination in the overlay address space independently of the message payload. This destination can describe a *specific node* if it is a complete overlay address or a *neighbourhood* if it is partially specified one. Up to the destination, the message is relayed through devp2p peer connections using *forwarding kademlia* (passing messages via semi-permanent peer-to-peer TCP connections between relaying nodes using kademlia routing). Within the destination neighbourhood the message is broadcast using gossip.

Since `pss` messages are encrypted, ultimately *the recipient is whoever can decrypt the message*. Encryption can be done using asymmetric or symmetric encryption methods.

The message payload is dispatched to *message handlers* by the recipient nodes and dispatched to subscribers via the API.

Important: `pss` does not guarantee message ordering (Best-effort delivery) nor message delivery (e.g. messages to offline nodes will not be cached and replayed) at the moment.

Thanks to end-to-end encryption, pss caters for private communication.

Due to forwarding kademlia, `pss` offers sender anonymity.

Using partial addressing, `pss` offers a sliding scale of recipient anonymity: the larger the destination neighbourhood (the smaller prefix you reveal of the intended recipient overlay address), the more difficult it is to identify the real recipient. On the other hand, since dark routing is inefficient, there is a trade-off between anonymity on the one hand and message delivery latency and bandwidth (and therefore cost) on the other. This choice is left to the application.

Forward secrecy is provided if you use the *Handshakes* module.

3.5.5.1 PSS Usage

Registering a recipient

Intended recipients first need to be registered with the node. This registration includes the following data:

1. `Encryption key` - can be a ECDSA public key for asymmetric encryption or a 32 byte symmetric key.

2. `Topic` - an arbitrary 4 byte word.
3. `Address`- destination (fully or partially specified Swarm overlay address) to use for deterministic routing.

   The registration returns a key id which is used to refer to the stored key in subsequent operations.

After you associate an encryption key with an address they will be checked against any message that comes through (when sending or receiving) given it matches the topic and the destination of the message.

Sending a message

There are a few prerequisites for sending a message over `pss`:

1. `Encryption key id` - id of the stored recipient's encryption key.
2. `Topic` - an arbitrary 4 byte word (with the exception of `0x0000` to be reserved for `raw` messages).
3. `Message payload` - the message data as an arbitrary byte sequence.

Note: The Address that is coupled with the encryption key is used for routing the message. This does *not* need to be a full address; the network will route the message to the best of its ability with the information that is available. If *no* address is given (zero-length byte slice), routing is effectively deactivated, and the message is passed to all peers by all peers.

Upon sending the message it is encrypted and passed on from peer to peer. Any node along the route that can successfully decrypt the message is regarded as a recipient. If the destination is a neighbourhood, the message is passed around so ultimately it reaches the intended recipient which also forwards the message to their peers, recipients will continue to pass on the message to their peers, to make it harder for anyone spying on the traffic to tell where the message "ended up."

After you associate an encryption key with a destination they will be checked against any message that comes through (when sending or receiving) given it matches the topic and the address in the message.

Important: When using the internal encryption methods, you MUST associate keys (whether symmetric or asymmetric) with an address space AND a topic before you will be able to send anything.

Sending a raw message

It is also possible to send a message without using the builtin encryption. In this case no recipient registration is made, but the message is sent directly, with the following input data:

1. `Message payload` - the message data as an arbitrary byte sequence.
2. `Address`- the Swarm overlay address to use for the routing.

Receiving messages

You can subscribe to incoming messages using a topic. Since subscription needs push notifications, the supported RPC transport interfaces are websockets and IPC.

Important: `pss` does not guarantee message ordering (Best-effort delivery) nor message delivery (e.g. messages to offline nodes will not be cached and replayed) at the moment.

Protocols

A framework is also in place for making `devp2p` protocols available using `pss` connections. This feature is only available using the internal golang API, read more in the GoDocs or the codes.

3.5.6 API reference

3.5.6.1 HTTP

Name	Method	Descriptors	
bzz	GET	Purpose	retrieve document at domain/some/path allowing domain to resolve
		Locator	bzz:/<domain_part>/<resource_path>
		Locator Parts	domain part: mandatory - ENS name or a valid Swarm hash. path par
		HTTP Codes	200; 300; 404; 500
		Responds with	The content stored at the resolved ENS entry (or the matched path) wi
		Example	
	POST	Purpose	post an application/x-tar or multipart/form-data (or any other Conter
		Locator	bzz:/<manifest_hash?>/<resource_path?>/<encrypt?>
		Locator Parts	manifest hash - optional - an existing manifest address to update a reso
		HTTP Codes	200
		Responds with	a hash of a newly created manifest
		Example	
	DELETE	Purpose	delete a resource from a manifest by unlinking it from the existing mar
		Locator	bzz:/<domain>/<path>
		Locator Parts	domain part - mandatory - a valid ENS hash or a valid Swarm manifes
		HTTP Codes	200; 404; 500
		Responds with	the hash of the new manifest which does not have component `path`
		Example	
bzz-immutable	GET	Purpose	The same as the generic scheme but there is no ENS domain resolution
		Locator	bzz-immutable:/<hash>
		Locator Parts	hash part - a valid Swarm hash that points to a manifest
		HTTP Codes	200; 404; 500
		Responds with	the resolved content at the specified address with a valid content-type
		Example	use bzz-hash to resolve the ens name into a hash then use it with immu
bzz-raw	GET	Purpose	When responding to GET requests with the bzz-raw scheme swarm do
		Locator	bzz-raw:/<content_hash>?content_type=<mime>
		Locator Parts	content hash - mandatory - a valid Swarm content hash. content type -
		HTTP Codes	200; 404; 500
		Responds with	
		Example	a pdf document (not the manifest wrapping it) resides at hash 6a18222
	POST	Purpose	
		Locator	
		Locator Parts	
		HTTP Codes	200; 404; 500
		Responds with	
		Example	
bzz-list	GET	Purpose	Returns a list of all files contained in <manifest> under <path> groupe
		Locator	bzz-list:/<domain>/<path>
		Locator Parts	domain part - mandatory - a valid ENS entry that points to a valid mar
		HTTP Codes	200; 404; 500
		Responds with	
		Example	
bzz-hash	GET	Purpose	responds with the hash value of the raw content - the same content ret

Name	Method	Descriptors	
		Locator	bzz-hash:/<domain>
		Locator Parts	domain part - mandatory. a valid ENS name
		HTTP Codes	200; 404; 500
		Responds with	text/plain
		Example	
bzz-feed	GET	Purpose	Retrieve a Feed update
		Locator	bzz-feed:/<hash>?user=<user>&topic=<topic>&name=<name>&time
		Locator Parts	hash - optional - manifest hash of the Feed, otherwise user param requ
		HTTP Codes	200; 400; 404; 500
		Responds with	The content stored in the requested Feed update
		Example	
	GET	Purpose	Get Feed metadata, used to help publishing updates
		Locator	bzz-feed:/<hash>?user=<user>&topic=<topic>&name=<name>&met
		Locator Parts	hash - optional - manifest hash of the Feed, otherwise user param requ
		HTTP Codes	200; 400; 500
		Responds with	application/json
		Example	
	POST	Purpose	Post an update to a Feed
		Locator	bzz-feed:/<hash>?user=<user>&topic=<topic>&level=<level>&time=
		Locator Parts	hash - optional - manifest hash of the Feed, otherwise user param requ
		HTTP Codes	200; 400; 500
		Responds with	
		Example	

3.5.6.2 JavaScript

Swarm currently supports a Javascript API through a few packages:

erebos

erebos is available through NPM by issuing the following command:

```
npm install @erebos/swarm-browser # browser only
npm install @erebos/swarm-node # node only
npm install @erebos/swarm # universal
```

Note: Full documentation is available on the documentation website.

swarm-js

swarm-js is available through NPM by issuing the following command:

```
npm install swarm-js
```

Note: Full documentation is available on the GitHub page.

swarmgw

swarmgw is available through NPM by issuing the following command:

```
npm install swarmgw
```

When installed globally, it can also be used directly from the CLI:

```
npm install -g swarmgw
```

Note: Full documentation is available on the GitHub page.

3.5.6.3 RPC

Swarm exposes an IPC API under the `bzz` namespace.

FUSE

`swarmfs.mount(HASH|domain, mountpoint))` mounts Swarm contents represented by a Swarm hash or a ens domain name to the specified local directory. The local directory has to be writable and should be empty. Once this command is succesfull, you should see the contents in the local directory. The HASH is mounted in a rw mode, which means any change insie the directory will be automatically reflected in Swarm. Ex: if you copy a file from somewhere else in to mountpoint, it is equvivalent of using a `swarm up <file>` command.

`swarmfs.unmount(mountpoint)` This command unmounts the HASH|domain mounted in the specified mountpoint. If the device is busy, unmounting fails. In that case make sure you exit the process that is using the directory and try unmounting again.

`swarmfs.listmounts()` For every active mount, this command display three things. The mountpoint, start HASH supplied and the latest HASH. Since the HASH is mounted in rw mode, when ever there is a change to the file system (adding file, removing file etc), a new HASH is computed. This hash is called the latest HASH.

PSS

`pss` methods are by default exposed via IPC. If websockets are activated on the node, they will also be available there *if* the `pss` module is explicitly specified.

All parameters are hex-encoded bytes or strings unless otherwise noted.

`pss.getPublicKey()` Retrieves the public key of the node, in hex format

`pss.baseAddr()` Retrieves the Swarm overlay address of the node, in hex format

`pss.stringToTopic(name)` Creates a deterministic 4 byte topic value from an input name, returned in hex format

`pss.setPeerPublicKey(publickey, topic, address)` Register a peer's public key. This is done once for every topic that will be used with the peer. Address can be anything from 0 to 32 bytes inclusive of the peer's swarm address. The method has no return value.

`pss.sendAsym(publickey, topic, message)` Encrypts the message using the provided public key, and signs it using the node's private key. It then wraps it in an envelope containing the topic, and sends it to the network. The method has no return value.

`pss.setSymmetricKey(symkey, topic, address, bool decryption)` Register a symmetric key shared with a peer. This is done once for every topic that will be used with the peer. Address can be anything from 0 to 32 bytes inclusive of the peer's Swarm overlay address. If the fourth parameter is false, the key will not be added to the list of symmetric keys used for decryption attempts. The method returns an id used to reference the symmetric key in consecutive calls.

pss.sendSym(symkeyid, topic, message) Encrypts the message using the provided symmetric key, wraps it in an envelope containing the topic, and sends it to the network. The method has no return value.

pss.GetSymmetricAddressHint(topic, symkeyid) Return the Swarm address associated with the peer registered with the given symmetric key and topic combination. If a match is found it returns the address data in hex format.

pss.GetAsymmetricAddressHint(topic, publickey) Return the Swarm address associated with the peer registered with the given asymmetric key and topic combination. If a match is found it returns the address data in hex format.

3.6 Swarm for Node-Operators

This section is about how to run your swarm node, or deploy a seperate private or public swarm network.

3.6.1 Installation and Updates

Swarm runs on all major platforms (Linux, macOS, Windows, Raspberry Pi, Android, iOS).

Swarm was written in golang and requires the go-ethereum client **geth** to run.

Note: The swarm package has not been extensively tested on platforms other than Linux and macOS.

3.6.2 Download pre-compiled Swarm binaries

Pre-compiled binaries for Linux, macOS and Windows are available to download via our official homepage.

3.6.3 Setting up Swarm in Docker

You can run Swarm in a Docker container. The official Swarm Docker image including documentation on how to run it can be found on Github or pulled from Docker.

You can run it with optional arguments, e.g.:

```
$ docker run -it ethersphere/swarm --debug --verbosity 4
```

In order to up/download, you need to expose the HTTP api port (here: to localhost:8501) and set the HTTP address:

```
$ docker run -p 8501:8500/tcp -it ethersphere/swarm  --httpaddr=0.0.0.0 --debug --
↪verbosity 4
```

In this example, you can use `swarm --bzzapi http://localhost:8501 up testfile.md` to upload `testfile.md` to swarm using the Docker node, and you can get it back e.g. with `curl http://localhost:8501/bzz:/<hash>`.

Note that if you want to use a pprof HTTP server, you need to expose the ports and set the address (with `--pprofaddr=0.0.0.0`) too.

In order to attach a Geth Javascript console, you need to mount a data directory from a volume:

```
$ docker run -p 8501:8500/tcp -v /tmp/hostdata:/data -it --name swarm1 ethersphere/
↪swarm --httpaddr=0.0.0.0 --datadir /data --debug --verbosity 4
```

Then, you can attach the console with:

```
$ docker exec -it swarm1 geth attach /data/bzzd.ipc
```

You can also open a terminal session inside the container:

```
$ docker exec -it swarm1 /bin/sh
```

3.6.4 Installing Swarm from source

The Swarm source code for can be found on https://github.com/ethersphere/swarm

3.6.4.1 Prerequisites: Go and Git

Building the Swarm binary requires the following packages:

- go: https://golang.org
- git: http://git.org

Grab the relevant prerequisites and build from source.

Ubuntu / Debian

```
$ sudo apt install git

$ sudo add-apt-repository ppa:gophers/archive
$ sudo apt-get update
$ sudo apt-get install golang-1.11-go

// Note that golang-1.11-go puts binaries in /usr/lib/go-1.11/bin. If you want␣
↪them on your PATH, you need to make that change yourself.

$ export PATH=/usr/lib/go-1.11/bin:$PATH
```

Archlinux

```
$ pacman -S git go
```

Generic Linux

The latest version of Go can be found at https://golang.org/dl/

To install it, download the tar.gz file for your architecture and unpack it to `/usr/local`

macOS

```
$ brew install go git
```

Windows

Take a look here at installing go and git and preparing your go environment under Windows.

3.6.4.2 Configuring the Go environment

You should then prepare your Go environment.

Linux

```
$ mkdir $HOME/go
$ echo 'export GOPATH=$HOME/go' >> ~/.bashrc
$ echo 'export PATH=$GOPATH/bin:$PATH' >> ~/.bashrc
$ source ~/.bashrc
```

macOS

```
$ mkdir $HOME/go
$ echo 'export GOPATH=$HOME/go' >> $HOME/.bash_profile
$ echo 'export PATH=$GOPATH/bin:$PATH' >> $HOME/.bash_profile
$ source $HOME/.bash_profile
```

3.6.4.3 Download and install Geth

Once all prerequisites are met, download and install Geth from https://github.com/ethereum/go-ethereum

3.6.4.4 Compiling and installing Swarm

Once all prerequisites are met, and you have `geth` on your system, clone the Swarm git repo and build from source:

```
$ git clone https://github.com/ethersphere/swarm
$ cd swarm
$ make swarm
```

Alternatively you could also use the Go tooling and download and compile Swarm from *master* via:

```
$ go get -d github.com/ethersphere/swarm
$ go install github.com/ethersphere/swarm/cmd/swarm
```

You can now run `swarm` to start your Swarm node. Let's check if the installation of `swarm` was successful:

```
swarm version
```

If your `PATH` is not set and the `swarm` command cannot be found, try:

```
$ $GOPATH/bin/swarm version
```

This should return some relevant information. For example:

```
Swarm
Version: 0.3
Network Id: 0
Go Version: go1.10.1
OS: linux
GOPATH=/home/user/go
GOROOT=/usr/local/go
```

3.6.4.5 Updating your client

To update your client simply download the newest source code and recompile.

3.6.5 Running the Swarm Go-Client

To start a basic Swarm node you must have both `geth` and `swarm` installed on your machine. You can find the relevant instructions in the Installation and Updates section. `geth` is the go-ethereum client, you can read up on it in the Ethereum Homestead documentation.

To start Swarm you need an Ethereum account. You can create a new account in `geth` by running the following command:

```
$ geth account new
```

You will be prompted for a password:

```
Your new account is locked with a password. Please give a password. Do not forget␣
↪this password.
Passphrase:
Repeat passphrase:
```

Once you have specified the password, the output will be the Ethereum address representing that account. For example:

```
Address: {2f1cd699b0bf461dcfbf0098ad8f5587b038f0f1}
```

Using this account, connect to Swarm with

```
$ swarm --bzzaccount <your-account-here>
# in our example
$ swarm --bzzaccount 2f1cd699b0bf461dcfbf0098ad8f5587b038f0f1
```

(You should replace `2f1cd699b0bf461dcfbf0098ad8f5587b038f0f1` with your account address key).

Important: Remember your password. There is no *forgot my password* option for `swarm` and `geth`.

3.6.5.1 Verifying that your local Swarm node is running

When running, `swarm` is accessible through an HTTP API on port 8500. Confirm that it is up and running by pointing your browser to http://localhost:8500 (You should see a Swarm search box.)

3.6.6 Interacting with Swarm

The easiest way to access Swarm through the command line, or through the Geth JavaScript Console by attaching the console to a running swarm node. `$BZZKEY$` refers to your account address key.

Linux

```
$ swarm --bzzaccount $BZZKEY
```

And, in a new terminal window:

```
$ geth attach $HOME/.ethereum/bzzd.ipc
```

macOS

```
$ swarm --bzzaccount $BZZKEY
```

And, in a new terminal window:

```
$ geth attach $HOME/Library/Ethereum/bzzd.ipc
```

Windows

```
$ swarm --bzzaccount $BZZKEY
```

And, in a new terminal window:

```
$ geth attach \\.\pipe\bzzd.ipc
```

Swarm is fully compatible with Geth Console commands. For example, you can list your peers using `admin.peers`, add a peer using `admin.addPeer`, and so on.

You can use Swarm with CLI flags and environment variables. See a full list in the Configuration .

3.6.7 How do I enable ENS name resolution?

The Ethereum Name Service (ENS) is the Ethereum equivalent of DNS in the classic web. It is based on a suite of smart contracts running on the *Ethereum mainnet*.

In order to use **ENS** to resolve names to swarm content hashes, `swarm` has to connect to a `geth` instance that is connected to the *Ethereum mainnet*. This is done using the `--ens-api` flag.

First you must start your geth node and establish connection with Ethereum main network with the following command:

```
$ geth
```

for a full geth node, or

```
$ geth --syncmode=light
```

for light client mode.

Note: Syncing might take a while. When you use the light mode, you don't have to sync the node before it can be used to answer ENS queries. However, please note that light mode is still an experimental feature.

After the connection is established, open another terminal window and connect to Swarm:

Linux

```
$ swarm --ens-api $HOME/.ethereum/geth.ipc \
--bzzaccount $BZZKEY
```

macOS

```
$ swarm --ens-api $HOME/Library/Ethereum/geth.ipc \
--bzzaccount $BZZKEY
```

Windows

```
$ swarm --ens-api \\.\pipe\geth.ipc \
--bzzaccount $BZZKEY
```

Verify that this was successful by pointing your browser to http://localhost:8500/bzz:/theswarm.eth/

3.6.7.1 Using Swarm together with the testnet ENS

It is also possible to use the Ropsten ENS test registrar for name resolution instead of the Ethereum main .eth ENS on mainnet.

Run a geth node connected to the Ropsten testnet

```
$ geth --testnet
```

Then launch the `swarm`; connecting it to the geth node (`--ens-api`).

Linux

```
$ swarm --ens-api $HOME/.ethereum/geth/testnet/geth.ipc \
--bzzaccount $BZZKEY
```

macOS

```
$ swarm --ens-api $HOME/Library/Ethereum/geth/testnet/geth.ipc \
--bzzaccount $BZZKEY
```

Windows

```
$ swarm --ens-api \\.\pipe\geth.ipc \
--bzzaccount $BZZKEY
```

Swarm will automatically use the ENS deployed on Ropsten.

For other ethereum blockchains and other deployments of the ENS contracts, you can specify the contract addresses manually. For example the following command:

```
$ swarm --ens-api eth:<contract 1>@/home/user/.ethereum/geth.ipc \
         --ens-api test:<contract 2>@ws:<address 1> \
         --ens-api <contract 3>@ws:<address 2>
```

Will use the `geth.ipc` to resolve `.eth` names using the contract at address `<contract 1>` and it will use `ws:<address 1>` to resolve `.test` names using the contract at address `<contract 2>`. For all other names it will use the ENS contract at address `<contract 3>` on `ws:<address 2>`.

3.6.7.2 Using an external ENS source

Important: Take care when using external sources of information. By doing so you are trusting someone else to be truthful. Using an external ENS source may make you vulnerable to man-in-the-middle attacks. It is only recommended for test and development environments.

Maintaining a fully synced Ethereum node comes with certain hardware and bandwidth constraints, and can be tricky to achieve. Also, light client mode, where syncing is not necessary, is still experimental.

An alternative solution for development purposes is to connect to an external node that you trust, and that offers the necessary functionality through HTTP.

If the external node is running on IP 12.34.56.78 port 8545, the command would be:

```
$ swarm --ens-api http://12.34.45.78:8545
```

You can also use `https`. But keep in mind that Swarm *does not validate the certificate*.

3.6.8 Connect to the SWAP-enabled testnet

The Swarm project now runs a SWAP-enabled (incentivized) testnet. It uses the same binary as the normal network node, but has incentivization switched on.

Important: The testnet is highly experimental and may be updated continuously until we release a mainnet version. All tokens used on this testnet are fictitious. Do NOT use this testnet with real value tokens. Funds can be lost.

The first public incentives-enabled Swarm network runs on the Ropsten network. It runs with BZZ network id **5**.

3.6.8.1 Prerequisites

1. Create an account on **Ropsten** and get Ropsten Ethers (to pay transaction fees from a Ropsten Faucet).
2. Start Swarm with `--swap-skip-deposit` flag and note the address of your newly deployed chequebook.
3. Fill in the address of the chequebook here:.

3.6.8.2 Network setup

Currently we provide 30 nodes in a kubernetes cluster on the public swap-enabled network.

3.6.8.3 Bootnodes

The bootnodes for the public swap-enabled Swarm cluster run in a separate VMs. The addresses are

```
enode://
↪7f4d606c91d50d91fd09cb44f8b3d8033f1ca87e977a881e91d77ff6af98b6a52245ba9aeba13a39024ae8bdf3a
↪122.203.99:40301
enode://
↪3d58e0cf0a057e71388dd15719cb8f7c94f732dd4f3e5f7a6e3f2185db68ed10ac352080b81811d17bf3f65873c
↪35.212.179:40301
```

Add these addresses with the *–bootnodes* when starting your node (see example below).

3.6.8.4 Run your own swap-enabled node and connect to the cluster

All SWAP-related configuration options start with the prefix *swap*. Check the configuration chapter below to consult available options. To enable a node to run with the incentivized layer switched on, add the *–swap* flag. However, the *–swap-backend-url* also has to be provided. This flag tells the node to which blockchain it will connect and through which provider (e.g. a *geth* node or via *infura*).

Get Tokens Getting tokens via the [faucet](https://ropsten.etherscan.io/address/0x49bf80bdee2684580966e476aee0dc3d773ffaf5#writeContract) is only possible *after* your start Swarm. To get tokens, first start Swarm with `swap-skip-deposit`, note down the address of the chequebook and call the *drip* function via the interface of etherscan, with the address of your chequebook as argument. Add the *swap-deposit-amount* flag in this case and set it to zero.

Note: the current faucet only allows to call the drip function one time per deployed chequebook contract.

Start Swarm An example of how to start the SWAP-enabled Swarm node (other configuration options for the Swarm binary apply here too: keystore, datadirectory, and all the other options. Refer to the configuration chapters below):

```
swarm --swap --swap-backend-url=https://ropsten.infura.io/v3/
↪4f7e7287d52447ab8865dbdcf7c203e1 \
   --swap-skip-deposit --ws --wsaddr=0.0.0.0 --wsorigins=* --wsapi=admin,
↪net,debug,bzz,stream,accounting,swap \
   --bzznetworkid 5 --bzzkeyhex␣
↪0C03CAE29D0D25A0DCF254E2AFAE7A8C137F887748AF21C53DBBF163CA367509 --
↪verbosity 3 \
   --bootnodes "enode://
↪7f4d606c91d50d91fd09cb44f8b3d8033f1ca87e977a881e91d77ff6af98b6a52245ba9aeba13a39024ae8bdf3a
↪122.203.99:40301,enode://
↪3d58e0cf0a057e71388dd15719cb8f7c94f732dd4f3e5f7a6e3f2185db68ed10ac352080b81811d17bf3f65873c
↪35.212.179:40301"
```

Check balance on your local node

The balances (remember that the node maintains an independent balance with each peer it had interactions with) can be queried via RPC. In the above snippet, we started the binary with the *–ws* flag(s), which allows the node to be queried via Websockets. Here is an example of how this can be done via JSON RPC (the values here are from an example, your node may return totally different values):

```
>$ echo '{"jsonrpc":"2.0","method":"swap_balances","params":[], "id":104}
↪' | websocat ws://localhost:8546/ -n --one-message --origin localhost |␣
↪jq
  {
    "jsonrpc": "2.0",
    "id": 104,
    "result": {
      "10e14c08a7c873adb30516807c138781da76d284dbb7f12d27d95919f9da3d24":␣
↪0,
      "1913db38b3a9d8cf2a5110f4ab1fbd0b882f729063e456e76c874bf3bda49788":␣
↪-488096051628,
      "292fb3f158a8313c0606df126bef393c0d49f1879f35a813e07fc370a95f318c":␣
↪-81349341938,
      "2cf4f38451678a6276ab1ad7039f91c7ebe99beca2342132d80566148157702f":␣
↪-744251831714,
      "4069823dc97610784f237c5189bb0047fa44fdb58050408186e76dfc678117e4":␣
↪0,
      "50b3e7aecb9348770b28fe846e6ccd4a65dc6a7e6fad7f32f386226fca0fbb05":␣
↪-732144077442,
      "5e2e42c0a2476cd3a5df0ba0f361a077202d0a05d8885940f5731f11bd06e314":␣
↪-81349341938,
      "817a6498af5f83c8b9bd7991523696ac82af0b19c68dd20c44dee128b04c24b4":␣
↪-569445393566,
      "8449ac2043f5eb12b859bb909ae7632fc2c64da8f5c5b0e24726bfc2b3b73683":␣
↪0,
      "8e14175898416068388ee88ac1b51b03c64c0f32ffd02bcfc5c40dc38a8273f4":␣
↪-162698683876,
      "9e2d5a87d089d32996ccc650e29caf6827dd31349ae779d643438eb6a64bae57":␣
↪0,
      "ac370a71c55dbae1eda0c41406f36db04967230f5a6d53a8ca00a96d560b25e9":␣
↪-650794735504,
      "b0b4283f1b00d9b12f9b57639175505d1e2db46ff553d393c6b79fbaa1eb5877":␣
↪0,
      "c13d2f0b212b2b2b7f4414be742ed361d63725fa571c7b04c562b980242372e2":␣
↪0,
      "c3faaf2ff61edbe4bece896467aa517989c82bcddcc9e4a0b7cc91ef991adc82":␣
↪-1233143154257,
      "d196803f49ac650598a606a3417b19c80d234e71a79c98881b9bb75359aebe6b":␣
↪0,
      "e75b3b14877058c9f4d6884fb0be474f9d42864aa0b430fac62c46d3058ba38d":␣
↪-4636912490466,
      "ed57d9d24f898731b928466ad05aeedeb698a82ad5e7f4a86a217b37a1190bbc":␣
↪-416804432270
```

(continues on next page)

(continued from previous page)

```
    }
  }
```

3.6.9 Running a SWAP-enabled node on an alternative blockchain

If you want to run Swarm with a different blockchain platform (for example, RIF Storage runs a Swarm network on their own RSK network, check [https://www.rifos.org/blog/rif-storage-testnet-launch], or you may want to run experiments and tests within your own development environment, say using *ganache*), you need to deploy the factory contract and provide its address to the options via *-swap-chequebook-factory*

3.6.10 Alternative modes

Below are examples on ways to run `swarm` beyond just the default network. You can instruct Swarm using the geth command line interface or use the geth javascript console.

3.6.10.1 Swarm in singleton mode (no peers)

If you **don't** want your swarm node to connect to any existing networks, you can provide it with a custom network identifier using `--bzznetworkid` with a random large number.

Linux

```
$ swarm --bzzaccount $BZZKEY \
--datadir $HOME/.ethereum \
--ens-api $HOME/.ethereum/geth.ipc \
--bzznetworkid <random number between 15 and 256>
```

macOS

```
$ swarm --bzzaccount $BZZKEY \
--datadir $HOME/Library/Ethereum/ \
--ens-api $HOME/Library/Ethereum/geth.ipc \
--bzznetworkid <random number between 15 and 256>
```

Windows

```
$ swarm --bzzaccount $BZZKEY \
--datadir %HOMEPATH%\AppData\Roaming\Ethereum \
--ens-api \\.\pipe\geth.ipc \
--bzznetworkid <random number between 15 and 256>
```

3.6.10.2 Adding enodes manually

By default, Swarm will automatically seek out peers in the network.

Additionally you can manually start off the connection process by adding one or more peers using the `admin.addPeer` console command.

Linux

```
$ geth --exec='admin.addPeer("ENODE")' attach $HOME/.ethereum/bzzd.ipc
```

macOS

```
$ geth --exec='admin.addPeer("ENODE")' attach $HOME/Library/Ethereum/bzzd.ipc
```

Windows

```
$ geth --exec='admin.addPeer("ENODE")' attach \\.\pipe\bzzd.ipc
```

(You can also do this in the Geth Console, as seen in Section 3.2.)

Note: When you stop a node, all peer connections will be saved. When you start again, the node will try to reconnect to those peers automatically.

Where ENODE is the enode record of a swarm node. Such a record looks like the following:

```
enode://
↪01f7728a1ba53fc263bcfbc2acacc07f08358657070e17536b2845d98d1741ec2af00718c79827dfdbecf5cfcd77965
↪2.3.4:30399
```

The enode of your swarm node can be accessed using `geth` connected to `bzzd.ipc`

Linux

```
$ geth --exec "admin.nodeInfo.enode" attach $HOME/.ethereum/bzzd.ipc
```

macOS

```
$ geth --exec "admin.nodeInfo.enode" attach $HOME/Library/Ethereum/bzzd.ipc
```

Windows

```
$ geth --exec "admin.nodeInfo.enode" attach \\.\pipe\bzzd.ipc
```

Note: Note how `geth` is used for two different purposes here: You use it to run an Ethereum Mainnet node for ENS lookups. But you also use it to "attach" to the Swarm node to send commands to it.

3.6.10.3 Connecting to the public Swarm cluster

By default Swarm connects to the public Swarm testnet operated by the Ethereum Foundation and other contributors.

The nodes the team maintains function as a free-to-use public access gateway to Swarm, so that users can experiment with Swarm without the need to run a local node. To download data through the gateway use the `https://swarm-gateways.net/bzz:/<address>/` URL.

3.6.10.4 Metrics reporting

Swarm uses the *go-metrics* library for metrics collection. You can set your node to collect metrics and push them to an influxdb database (called *metrics* by default) with the default settings. Tracing is also supported. An example of a default configuration is given below:

```
$ swarm --bzzaccount <bzzkey> \
--debug \
--metrics \
--metrics.influxdb.export \
--metrics.influxdb.endpoint "http://localhost:8086" \
--metrics.influxdb.username "user" \
--metrics.influxdb.password "pass" \
```

(continues on next page)

(continued from previous page)

```
--metrics.influxdb.database "metrics" \
--metrics.influxdb.host.tag "localhost" \
--verbosity 4 \
--tracing \
--tracing.endpoint=jaeger:6831 \
--tracing.svc myswarm
```

3.6.11 Go-Client Command line options Configuration

The `swarm` executable supports the following configuration options:

- Configuration file
- Environment variables
- Command line

Options provided via command line override options from the environment variables, which will override options in the config file. If an option is not explicitly provided, a default will be chosen.

In order to keep the set of flags and variables manageable, only a subset of all available configuration options are available via command line and environment variables. Some are only available through a TOML configuration file.

Note: Swarm reuses code from ethereum, specifically some p2p networking protocol and other common parts. To this end, it accepts a number of environment variables which are actually from the `geth` environment. Refer to the geth documentation for reference on these flags.

This is the list of flags inherited from `geth`:

```
--identity
--bootnodes
--datadir
--keystore
--port
--nodiscover
--v5disc
--netrestrict
--nodekey
--nodekeyhex
--maxpeers
--nat
--ipcdisable
--ipcpath
--password
```

3.6.12 Config File

Note: `swarm` can be executed with the `dumpconfig` command, which prints a default configuration to STDOUT, and thus can be redirected to a file as a template for the config file.

A TOML configuration file is organized in sections. The below list of available configuration options is organized according to these sections. The sections correspond to *Go* modules, so need to be respected in order for file configuration to work properly. See https://github.com/naoina/toml for the TOML parser and encoder library for Golang, and https://github.com/toml-lang/toml for further information on TOML.

To run Swarm with a config file, use:

```
$ swarm --config /path/to/config/file.toml
```

3.6.13 General configuration parameters

Config file	Command line flag	Environment variable	Default value	Description
n/a	–config	n/a	n/a	Path to config file in TOML format
n/a	–bzzapi	n/a	http://127.0.0.1:8500	Swarm HTTP endpoint
BootNodes	–bootnodes	SWARM_BOOTNODES		Boot nodes
BzzAccount	–bzzaccount	SWARM_ACCOUNT		Swarm account key
BzzKey	n/a	n/a	n/a	Swarm node base address ($hash(PublicKey)hash(PublicKey)$). This is used to decide storage based on radius and routing by kademlia.
Cors	–corsdomain	SWARM_CORS		Domain on which to send Access-Control-Allow-Origin header (multiple domains can be supplied separated by a ‘,’)
n/a	–debug	n/a	n/a	Prepends log messages with call-site location (file and line number)
n/a	–defaultpath	n/a	n/a	path to file served for empty url path (none)
n/a	–delivery-skip-check	SWARM_DELIVERY_SKIP_CHECK	false	Skip chunk delivery check (default false)
EnsApi	–ens-api	SWARM_ENS_API	<$GETH_DATADIR>/geth.ipc	Ethereum Name Service API address
EnsRoot	–ens-addr	SWARM_ENS_ADDR	ens.test.addr	Ethereum Name Service contract address
ListenAddr	–httpaddr	SWARM_LISTEN_ADDR	127.0.0.1	Swarm listen address
n/a	–manifest value	n/a	true	Automatic manifest upload (default true)
n/a	–mime value	n/a	n/a	Force mime type on upload
NetworkId	–bzznetworkid	SWARM_NETWORK_ID	322	Network ID
Path	–datadir	GETH_DATADIR	<$GETH_DATADIR>/swarm	Path to swarm configuration directory
Port	–bzzport	SWARM_PORT	8500	Port to run the http proxy server
PublicKey	n/a	n/a	n/a	Public key of swarm base account
n/a	–recursive	n/a	false	Upload directories recursively (default false)
n/a	–stdin	n/a	n/a	Reads data to be uploaded from stdin
n/a	–storepath value	SWARM_STORE_PATH	<$GETH_DATADIR>/swarm/bzz-<$BZZ_KEY>/chunks	Path to swarm chunk DB
n/a	–storesize value	SWARM_STORE_CAPACITY	5000000	Number of chunks (5M is roughly 20-25GB) (default 5000000)]
n/a	–storecache value	SWARM_STORE_CACHE_CAPACITY	5000	Number of recent chunks cached in memory (default 5000)
n/a	–sync-update-delay value	SWARM_ENV_SYNC_UPDATE_DELAY	15s	Duration for sync subscriptions update after no new peers are added (default 15s)
SyncDisabled	–nosync	SWARM_ENV_SYNC_DISABLE	false	Disable swarm node synchronization

Continued on next page

Table 2 – continued from previous page

Config file	Command line flag	Environment variable	Default value	Description
SwapBackendURL	–swap-backend-url	SWARM_SWAP_BACKEND_URL		URL of the Ethereum API provider (access to the blockchain) to use to settle SWAP payments
SwapEnabled	–swap	SWARM_SWAP_ENABLE	false	Enable SWAP. If present, the node starts with accounting enabled. Only works if the backend URL is provided as well.
SwapPaymentThreshold	–swap-payment-threshold	SWARM_SWAP_PAYMENT_THRESHOLD	1000000	Honey amount at which payment is triggered
SwapDisconnectThreshold	–swap-disconnect-threshold	SWARM_SWAP_DISCONNECT_THRESHOLD	1500000	Honey amount at which a peer disconnects
SwapDepositAmount	–swap-deposit-amount	SWARM_SWAP_DEPOSIT_AMOUNT		Deposit amount in Honey for swap chequebook
SwapSkipDeposit	–swap-skip-deposit	SWARM_SWAP_SKIP_DEPOSIT	false	Don't deposit during boot sequence
SwapLogPath	–swap-audit-logpath	SWARM_SWAP_LOG_PATH	n/a	Write execution logs of swap audit to the given directory
SwapChequebookFactory	–swap-chequebook-factory	SWARM_SWAP_CHEQUEBOOK_FACTORY_ADDR		Swap Chequebook factory contract address (default value 0x878Ccb2e3c2973767e431bAec86D1EFd809480d5). Prevents fraudulent contract address creation.
Contract	–swap-chequebook	SWARM_CHEQUEBOOK_ADDR	0x0	Swap chequebook contract address
n/a	–verbosity value	n/a	3	Logging verbosity: 0=silent, 1=error, 2=warn, 3=info, 4=debug, 5=detail
n/a	–ws	n/a	false	Enable the WS-RPC server
n/a	–wsaddr value	n/a	localhost	WS-RPC server listening interface
n/a	–wsport value	n/a	8546	WS-RPC server listening port
n/a	–wsapi value	n/a	n/a	API's offered over the WS-RPC interface
n/a	–wsorigins value	n/a	n/a	Origins from which to accept websockets requests
n/a	n/a	SWARM_AUTO_DEFAULTPATH	false	Toggle automatic manifest default path on recursive uploads (looks for index.html)

3.7 Swarm for Go-Client Contributors

All you need to know to contribute to the Swarm Go-Client implementation.

Source code is located at https://github.com/ethersphere/swarm/.

3.7.1 Introduction

Swarm nodes can also connect with one (or several) Ethereum blockchains for domain name resolution and one ethereum blockchain for bandwidth and storage compensation.

Nodes running the same network id are supposed to connect to the same blockchain for payments. A Swarm network is identified by its network id which is an arbitrary integer.

Swarm supports encryption. Upload of unencrypted sensitive and private data is highly discouraged as **there is no way to undo an upload**. Users should refrain from uploading illegal, controversial or unethical content.

Always use encryption for sensitive content. For encrypted content, uploaded data is 'protected', i.e. only those that know the reference to the root chunk (the Swarm hash of the file) as well as the decryption key can access the content. Since publishing this reference (on ENS or with Feeds) requires an extra step, users are mildly protected against careless publishing as long as they use encryption. Even though there is no guarantees for removal, unaccessed content that is not explicitly insured will eventually disappear from the Swarm, as nodes will be incentivised to garbage collect it in case of storage capacity limits.

Swarm is a Persistent Data Structure, therefore there is no notion of delete/remove action in Swarm. This is because content is disseminated to Swarm nodes who are incentivised to serve it.

Important: It is not possible to **delete or remove** content uploaded to Swarm. **Always encrypt** sensitive content using the integrated Swarm encryption.

3.7.2 Reporting a bug and contributing

Issues are tracked on github and github only. Swarm related issues and PRs have labels prefixed with *swarm*:

- https://github.com/ethersphere/swarm/issues
- Good first issues

Please include the commit and branch when reporting an issue.

Pull requests should by default commit on the *master* branch.

Prospective contributors please read the *Contributing* section from our readme: https://github.com/ethersphere/swarm#contributing.

3.7.3 High level component description

In this chapter we introduce the internal software architecture of the go swarm code. It is only a high level description of the most important components. The code is documented in finer detail as comments in the codebase itself.

3.7.3.1 Interfaces to Swarm

There are currently three entry points for communicating with a swarm node, and there is a command line interface.

Wire Protocol The p2p entry point - this is the way the peers talk to each other. Protocol structure adheres to the devp2p standard and the transport is being done RLPx over TCP.

HTTP Proxy The user entry point to Swarm. User operations over dapps and CLI that interact with Swarm are proxied through the HTTP interface. The API exposes methods to interact with content on Swarm.

RPC Another user interface mainly used for development purposes. The user facing side of this is to be deprecated.

CLI The CLI is a wrapper for the HTTP interface allowing users easy access to basic up-download functionality, content management, and it also implements some administrative tasks.

3.7.3.2 Structural components and key processes

Chunker When a file is submitted to the system, the input data stream is then transformed into chunks, encrypted, then hashed and stored. This results in a single root chunk reference of the data.

Syncing process Syncing is the process that deals with changes in the network when nodes join and leave, and when new content is uploaded. Push and pull syncing work together to get chunks to where they are supposed to be stored (to the local neighbourhood where they belong).

Push Sync A process initiated by the uploader of content to make sure that the chunks get to the areas in the network from which they can be retrieved. Combining push-sync with tags allows users to track the status of their uploads. Push syncing is initiated upon upload of new content.

Pull Sync Pull syncing is initiated by all participating nodes in order to fill up their local storage allocation in order to keep redundancy by replicating the local storage of their neighbouring peers. Pull syncing caters the need for chunk propagation towards the nearest neighbourhood. This process is responsible for maintaining a minimal redundancy level for the stored chunks.

3.7.3.3 Storage module

LocalStore Provides persistent storage on each node. Provides indexes, iterators and metric storage to other components.

NetStore Extends local storage with network fetching. The net store is exposed internally between the APIs in order to transparently resolve any chunk dependencies that might be needed to be satisfied from the network in order to accomodate different operations on content.

Kademlia Kademlia in the sense of Swarm has two different meanings. Firstly, Kademlia is the type of the network topology that Swarm builds upon. Secondly, within the Swarm codebase the component which manages the connections to peers over the devp2p network in order to form the Kademlia topology. Peers exchange the neccessary information about each other through a discovery protocol (which does not build upon the devp2p discovery protocol).

Feeds Swarm Feeds allows a user to build an update feed about a particular topic without resorting to ENS on each update. The update scheme is built on swarm chunks with chunk keys following a predictable, versionable pattern. A Feed is defined as the series of updates of a specific user about a particular topic.

3.7.3.4 Communication layer

PSS A messaging subsystem which builds upon the Kademlia topology to provide application level messaging (eg. chat dapps) and is also used for Push-sync.

3.7.4 Simulation Framework

Find everything you need to run Swarm network simulations for testing and debugging.

Index

A

API, 14

B

bzzhash, 15

C

chunk, 13
chunk size, 15
chunker, 15

H

hash, 15
HTTP proxy, 14

J

joining, 15

M

manifest, 13, 14
merkle tree, 15
message, 13

S

splitting, 15
storage layer, 13

U

URL schemes, 14