swarm documentation

Release 0.2rc5

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

Aug 08, 2017

Contents

1 Introduction **3**
- 1.1 Background . . . 4
- 1.2 Basics . . . 4
- 1.3 About . . . 6
  - 1.3.1 This document . . . 6
  - 1.3.2 Status . . . 6
  - 1.3.3 License . . . 6
  - 1.3.4 Credits . . . 6
  - 1.3.5 Community . . . 7
  - 1.3.6 Reporting a bug and contributing . . . 7
  - 1.3.7 Roadmap and Resources . . . 7

2 Installation and Updates **9**
- 2.1 Installation . . . 9
- 2.2 Supported Platforms . . . 9
- 2.3 Prerequisites . . . 9
- 2.4 Ubuntu . . . 10
- 2.5 Installing from source . . . 10
- 2.6 Updating your client . . . 11

3 Connecting to Swarm (Simple) **13**
- 3.1 How do I connect? . . . 13
- 3.2 How do I upload and download? . . . 14

4 Connecting to Swarm (Advanced) **15**
- 4.1 Preparation . . . 15
- 4.2 Connecting to the swarm testnet . . . 16
  - 4.2.1 Connecting swarm only (no blockchain) . . . 16
  - 4.2.2 Using swarm together with the Ropsten testnet blockchain . . . 16
  - 4.2.3 Adding enodes manually . . . 17
- 4.3 Swarm in singleton mode . . . 17
- 4.4 Running a private swarm . . . 18
- 4.5 Testing SWAP . . . 20
  - 4.5.1 Testing SWAP on your private blockchain. . . . 20
    - 4.5.1.1 Custom genesis block . . . 20
    - 4.5.1.2 Mining on your private chain . . . 21
- 4.6 Configuration . . . 22
- 4.7 Command line options for swarm . . . 22
- 4.8 Configuration options . . . 22
  - 4.8.1 Main parameters . . . 22
  - 4.8.2 Storage parameters . . . 23
  - 4.8.3 Chunker/bzzhash parameters . . . 23
  - 4.8.4 Synchronisation parameters . . . 23
  - 4.8.5 Hive/Kademlia parameters . . . 23
  - 4.8.6 SWAP parameters . . . 24
  - 4.8.7 Setting up SWAP . . . 24

5 Usage **27**
5.1 Using swarm from the command line . . . 27
5.1.1 Uploading a file or directory to the swarm . . . 27
5.1.1.1 Example: uploading a file . . . 27
5.1.1.2 Example: Uploading a directory . . . 28
5.2 Content retrieval: hashes and manifests . . . 29
5.2.1 Retrieving content using the http proxy . . . 29
5.2.2 bzz url schemes . . . 29
5.2.2.1 bzz . . . 29
5.2.2.2 bzzi (immutable) . . . 30
5.2.2.3 bzzr (raw) . . . 30
5.2.3 Manifests . . . 30
5.2.4 Path Matching on Manifests . . . 32
5.3 Ethereum Name Service . . . 32
5.3.1 Content Retrieval using ENS . . . 33
5.3.2 Registering names for your swarm content . . . 33
5.3.2.1 Preparation . . . 33
5.3.2.2 Registering a .test domain . . . 33
5.3.2.3 Registering a .eth domain . . . 34
5.3.2.4 Setting up a resolver . . . 34
5.3.2.5 Registering a swarm hash on the publicResolver . . . 34
5.3.2.6 Looking up names in the ENS manually . . . 35
5.3.2.7 Updating your content . . . 35
5.4 The HTTP API . . . 35
5.5 Swarm IPC API . . . 35
5.5.1 Mounting Swarm . . . 36
5.5.1.1 Installing FUSE . . . 37
5.5.2 Chequebook RPC API . . . 37
5.5.3 Example: use of the console . . . 37
5.5.3.1 Uploading content . . . 37
5.5.3.2 Downloading content . . . 38

6 Architecture **39**
6.1 Swarm hash . . . 41
6.1.1 Introduction . . . 41
6.1.2 Description . . . 41
6.1.3 Strict interpretation . . . 41
6.1.4 Loose interpretations . . . 42
6.2 Chunker . . . 42
6.3 Distributed Preimage Archive . . . 42
6.3.1 High-level design . . . 42
6.3.2 Requests . . . 43
6.4 Syncing . . . 44
6.5 Peer management (hive, kademlia) . . . 45
6.5.1 Peer addresses . . . 45
6.5.2 Logarithmic distance and network topology . . . 46
6.5.3 Peer table format . . . 46
6.5.4 Peer table update . . . 47
6.6 The bzz protocol . . . 47
6.7 Incentivisation . . . 47
6.7.1 swap, swear & swindle . . . 47
6.7.2 SWAP – Swarm Accounting Protocol . . . 47
6.7.3 SWEAR – Storage With Enforced Archiving Rules or Swarm Enforcement And Registration . . . 48
6.7.4 SWINDLE – Secured With INsurance Deposit Litigation and Escrow . . . 48

7 Indices and tables **49**

Contents:

Chapter 1

Introduction

Swarm is a distributed storage platform and content distribution service, a native base layer service of the ethereum *web 3* stack. The primary objective of Swarm is to provide a sufficiently decentralized and redundant store of Ethereum's public record, in particular to store and distribute dapp code and data as well as block chain data. From an economic point of view, it allows participants to efficiently pool their storage and bandwidth resources in order to provide the aforementioned services to all participants.

From the end user's perspective, Swarm is not that different from WWW, except that uploads are not to a specific server. The objective is to offer a peer-to-peer storage and serving solution that is DDOS-resistant, zero-downtime, fault-tolerant and censorship-resistant as well as self-sustaining due to a built-in incentive system which uses peer-to-peer accounting and allows trading resources for payment. Swarm is designed to deeply integrate with the devp2p multiprotocol network layer of Ethereum as well as with the Ethereum blockchain for domain name resolution, service payments and content availability insurance (the latter is to be implemented in POC 0.4 by Q2 2017).

This document provides you with information on :

- how to run and configure a local swarm node
- how to connect to the test network
- how to store and access content on swarm
- swarm architecture and concepts such as chunk, hash and manifest
- command line tools relevant to swarm
- API documentation for the http swarm proxy
- API documentation for the bzz RPC module
- how to register swarm domains with the Ethereum Name Service
- how to test, manage logging, debug and report issues

Background

The primary objective of Swarm is to provide a sufficiently decentralized and redundant store of Ethereum's public record, in particular to store and distribute Đapp code and data as well as block chain data. [Note that the latter is not part of the current release].

From an economic point of view, it allows participants to efficiently pool their storage and bandwidth resources in order to provide the aforementioned services to all participants.

These objectives entail the following design requirements:

- distributed storage, inclusivity, long tail of power law
- flexible expansion of space without hardware investment decisions, unlimited growth
- zero downtime
- immutable, unforgeable, verifiable yet plausibly deniable storage
- no single point of failure, fault and attack resilience
- censorship resistance, universally accessible permanent public record
- sustainability due to a incentive system
- efficient market driven pricing. tradeable trade off of memory, persistent storage, bandwidth
- efficient use of the blockchain by the swarm accounting protocol
- deposit-challenge based guaranteed storage [planned for POC 0.4 by Q2 2017]

Basics

Swarm client is part of the Ethereum stack, the reference implementation is written in golang and found under the go-ethereum repository. Currently at POC (proof of concept) version 0.2 is running on all nodes.

Swarm defines the *bzz subprotocol* running on the ethereum devp2p network. The bzz subprotocol is in flux, the specification of the wire protocol is considered stable only with POC 0.4 expected in Q2 2017.

The swarm of Swarm is the collection of nodes of the devp2p network each of which run the bzz protocol on the same network id.

Swarm nodes are also connected to an ethereum blockchain. Nodes running the same network id are supposed to connect to the same blockchain. Such a swarm network is identified by its network id which is an arbitrary integer.

Swarm allows for *upload and disappear* which means that any node can just upload content to the swarm and then is allowed to go offline. As long as nodes do not drop out or become unavailable, the content will still be accessible due to the 'synchronization' procedure in which nodes continuously pass along available data between each other.

Note: Uploaded content is not guaranteed to persist until storage insurance is implemented (expected in POC 0.4 by Q2). All participating nodes should consider 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.

Note: Swarm POC 0.2 uses no encryption. Upload of sensitive and private data is highly discouraged as there is no way to undo an upload. Users should refrain from uploading unencrypted sensitive data, in other words

- no valuable personal content

- no illegal, controversial or unethical content

Swarm defines 3 crucial notions

chunks pieces of data (max 4K), the basic unit of storage and retrieval in the swarm

hash cryptographic hash of data that serves as its unique identifier and address

manifest data structure describing collections allow for url based access to content

In this guide, content is understood very broadly in a technical sense denoting any blob of data. Swarm defines a specific identifier for a piece of content. This identifier serves as the retrieval address for the content. Identifiers need 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 let us view the identifiers 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, i.e. Hash addressing therefore is immutable in the strong sense that you cannot even express mutable content: "changing the content changes the hash".

Users, however, usually use some discovery and or semantic access to data, which is implemented by the ethereum name service (ENS). The ENS enables content retrieval based on mnemonic (or branded) names, much like the DNS of the world wide web, but without servers.

Swarm nodes participating in the network also have their own *base address (also called bzzkey)* which is derived as the (keccak 256bit sha3) hash of an ethereum address, 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 defined by its swarm hash. The hashes of data chunks themselves are packaged into a chunk which in turn has its own hash. In this way the content gets mapped to a chunk 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).

The current version of swarm implements a *strictly content addressed distributed hash table* (DHT). Here 'strictly content addressed' means that the node(s) closest to the address of a chunk do not only serve information about the content but actually host the data. (Note that although it is part of the protocol, we cannot have any sort of guarantee that it will be preserved. this is a caveat worth stating again: no guarantee of permanence and persistence). In other words, in order to retrieve a piece of content (as a part of a larger collection/document) a chunk must reach its destination from the uploader to the storer when storing/uploading and must also be served back to a requester when retrieving/downloading. 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 offers (very low) constant time for lookups logarithmic to the network size.

Note: There is no such thing as delete/remove in swarm. Once data is uploaded there is no way you can initiate her 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 (to be implemented in POC 0.4 expected by Q2 of 2017) will be used to protect important content from this fate.

Swarm content access is centred around the notion of a manifest. A manifest file describes a document collection, e.g.,

- a filesystem directory
- an index of a database
- a virtual server

Manifests specify paths and corresponding content hashes allowing for url based content retrieval. Manifests can therefore define a routing table for (static) assets (including dynamic content using for instance static javascript). This offers the functionality of *virtual hosting*, storing entire directories or web(3)sites, similar to www but without servers.

You can read more about these components in Architecture.

About

This document

This document's source code is found at https://github.com/ethersphere/swarm-guide The most up-to-date swarm book in various formats is available on the old web http://ethersphere.org/swarm/docs as well as on swarm bzz://swarm/guide

Status

The status of swarm is proof of concept vanilla prototype tested on a toy network. This version is POC 0.2.5

Note: Swarm is experimental code and untested in the wild. Use with extreme care.

License

Credits

Swarm is code by Ethersphere (ΞTHΞRSPHΞΞ) *https://github.com/ethersphere*

the team behind swarm:

- Viktor Trón @zelig
- Dániel A. Nagy @nagydani
- Aron Fischer @homotopycolimit
- Nick Johnson @Arachnid
- Zsolt Felföldi @zsfelfoldi

Swarm is funded by the Ethereum Foundation.

Special thanks to

- Felix Lange, Alex Leverington for inventing and implementing devp2p/rlpx;
- Jeffrey Wilcke and the go team for continued support, testing and direction;
- Gavin Wood and Vitalik Buterin for the vision;
- Alex Van der Sande, Fabian Vogelsteller, Bas van Kervel and the Mist team
- Nick Savers, Alex Beregszaszi, Daniel Varga, Juan Benet for inspiring discussions and ideas

- Participants of the orange lounge research group
- Roman Mandeleil and Anton Nashatyrev for the java implementation
- Igor Sharudin for example dapps
- Community contributors for feedback and testing

Community

Daily development and discussions are ongoing in various gitter channels:

- https://gitter.im/ethereum/swarm: general public chatroom about swarm dev
- https://gitter.im/ethersphere/orange-lounge: our reading/writing/working group and R&D sessions
- https://gitter.im/ethereum/pss: about postal services on swarm - messaging with deterministic routing
- https://gitter.im/ethereum/swatch: variable bitrate media streaming and multicast/broadcast solution

Swarm discussions also on the Ethereum subreddit: http://www.reddit.com/r/ethereum

Reporting a bug and contributing

Issues are tracked on github and github only. Swarm related issues and PRs are labeled with swarm: https://github.com/ethereum/go-ethereum/labels/swarm

Please include the commit and branch when reporting an issue.

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

Roadmap and Resources

Swarm roadmap and tentative plan for features and POC series are found on the wiki: https://github.com/ethereum/go-ethereum/wiki/swarm-roadmap https://github.com/ethereum/go-ethereum/wiki/swarm---POC-series

the *swarm homepage* is accessible via swarm at bzz://swarm or the gateway http://swarm-gateways.net/bzz:/swarm/

The swarm page also contains a list of swarm-related talks (video recording and slides).

You can also find the (first 2) ethersphere orange papers there.

Public gateways are:

- http://swarm-gateways.net/
- http://web3.download/
- http://ethereum-swarm.net/

Swarm testnet monitor: http://stats.ens.domains/

Source code is at https://github.com/ethereum/go-ethereum/

Example dapps are at https://github.com/ethereum/swarm-dapps

This document source https://github.com/ethersphere/swarm-guide

Chapter 2

Installation and Updates

Installation

Swarm is part of the Ethereum stack, the reference implementation is currently at POC (proof of concept) version 0.2.

The source code is found on github: https://github.com/ethereum/go-ethereum/tree/master/

Supported Platforms

Geth runs on all major platforms (linux, MacOSX, Windows, also raspberry pi, android OS, iOS).

Note: This package has not been tested on platforms other than linux and OSX.

Prerequisites

building the swarm daemon **`swarm`** requires the following packages:

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

Grab the relevant prerequisites and build from source.

On linux (ubuntu/debian variants) use `apt` to install go and git

```
sudo apt install golang git
```

while on Mac OSX you'd use **`brew`**

```
brew install go git
```

Then you must prepare your go environment as follows

```
mkdir ~/go
export GOPATH="$HOME/go"
echo 'export GOPATH="$HOME/go"' >> ~/.profile
```

Ubuntu

The Ubuntu repositories carry an old version of Go.

Ubuntu users can use the 'gophers' PPA to install an up to date version of Go (version 1.7 or later is preferred). See https://launchpad.net/~gophers/+archive/ubuntu/archive for more information. Note that this PPA requires adding /usr/lib/go-1.X/bin to the executable PATH.

Other distros

Download the latest distribution

```
curl -O https://storage.googleapis.com/golang/go1.7.3.linux-amd64.tar.gz
```

Unpack it to the /usr/local (might require sudo)

```
tar -C /usr/local -xzf go1.7.3.linux-amd64.tar.gz
```

Set GOPATH and PATH

For Go to work properly, you need to set the following two environment variables:

Setup a go folder

```
mkdir -p ~/go; echo "export GOPATH=$HOME/go" >> ~/.bashrc
```

Update your path

```
echo "export PATH=$PATH:$HOME/go/bin:/usr/local/go/bin" >> ~/.bashrc
```

Read the environment variables into current session:

```
source ~/.bashrc
```

Installing from source

Once all prerequisites are met, download the go-ethereum source code

```
mkdir -p $GOPATH/src/github.com/ethereum
cd $GOPATH/src/github.com/ethereum
git clone https://github.com/ethereum/go-ethereum
cd go-ethereum
git checkout master
go get github.com/ethereum/go-ethereum
```

and finally compile the swarm daemon `swarm` and the main go-ethereum client `geth`.

```
go install -v ./cmd/geth
go install -v ./cmd/swarm
```

You can now run **`swarm`** to start your swarm node. Let's check *swarm*'s installation

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

Should give you some relevant information back

```
Swarm
Version: 0.2
Network Id: 0
Go Version: go1.7.4
OS: linux
```

```
GOPATH=/home/user/go
GOROOT=/usr/local/go
```

Updating your client

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

```
cd $GOPATH/src/github.com/ethereum/go-ethereum
git checkout master
git pull
go install -v ./cmd/geth
go install -v ./cmd/swarm
```

Chapter 3

Connecting to Swarm (Simple)

These instructions layout the simplest way to use swarm.

How do I connect?

To start a basic swarm node you must have both geth and swarm installed on your manchine. You can find the relevant instructions in the Installation section of the Swarm manual.

Note: You can find the relevant instructions in the Installation and Updates section of the Swarm manual.

If you do have not yet made your Ethereum account, start 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 (for example MYPASSWORD) the output will be your Ethereum address. This is also the base address for your Swarm node.

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

Since we need to use it later, save it into your ENV variables under the name `BZZKEY`

```
BZZKEY=2f1cd699b0bf461dcfbf0098ad8f5587b038f0f1
```

Next, start your geth node and establish connection with Ethereum main network with the following command

```
geth
```

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

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

How do I upload and download?

The best way to upload and download files to/from Swarm has to do with using curl.

To upload a single file, run this:

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

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

```
027e57bcbae76c4b6a1c5ce589be41232498f1af86e1b1a2fc2bdffd740e9b39
```

This is the address string of your file inside Swarm.

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

```
curl -s http://localhost:8500/bzz:/
↪027e57bcbae76c4b6a1c5ce589be41232498f1af86e1b1a2fc2bdffd740e9b39
```

And that's it.

Good luck, we hope you enjoyed using Swarm!

Chapter 4

Connecting to Swarm (Advanced)

These instructions will begin by laying out how to start a local, private, personal (singleton) swarm. Use this to familiarise yourself with the functioning of the client; upload and download and http proxy.

Preparation

To start a basic swarm node we must start geth with an empty data directory on a private network and then connect the swarm daemon to this instance of geth.

First set aside an empty temporary directory to be the data store

---

Note: If you followed the installation instructions from this guide, you will find your executables in the $GOPATH/bin directory. Make sure to move your files into an executable $PATH, or include $GOPATH/bin directory on it.

```
DATADIR=/tmp/BZZ/`date +%s`
```

then make a new account using this directory

```
geth --datadir $DATADIR 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 (for example MYPASSWORD) the output will be an address - the base address of the swarm node.

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

We save it under the name `BZZKEY`

```
BZZKEY=2f1cd699b0bf461dcfbf0098ad8f5587b038f0f1
```

With the preparations complete, we can now launch our swarm client. In what follows we detail a few ways you might want to use swarm.

- connecting to the swarm testnet without blockchain
- connecting to the swarm testnet and connecting to the Ropsten blockchain

- using swarm in singleton mode for local testing
- launching a private swarm
- testing SWAP accounting with a private Swarm

Connecting to the swarm testnet

Note: IMPORTANT: Automatic connection to the testnet is currently not working properly for all users. This issue is being fixed right now. In the meantime, please add a few enodes manually to bootstrap your node. See "Adding enodes manually" below.

Swarm needs an ethereum blockchain for

- domain name resolution using the Ethereum Name Service (ENS) contract.
- incentivisation (for example: SWAP)

If you do not care about domain resolution and run your swarm without SWAP (the default), then connecting to the blockchain is unnecessary. Hence `swarm` does not require either the `--ens-api` or `--swap-api` flags.

Connecting swarm only (no blockchain)

Set up you environment as seen above, ie., make sure you have a data directory.

Note: Even though you do not need the ethereum blockchain, you will need geth to generate a swarm account ($BZZKEY), since this account determines the base address that your swarm node is going to use.

```
swarm --bzzaccount $BZZKEY \
      --datadir $DATADIR \
      --ens-api '' \
      2>> $DATADIR/swarm.log < <(echo -n "MYPASSWORD") &
```

The `swarm` daemon will seek out and connect to other swarm nodes. It manages its own peer connections independent of `geth`.

Using swarm together with the Ropsten testnet blockchain

In case you don't yet have an account, run

```
geth --datadir $DATADIR --testnet account new
```

Run a geth node connected to the Ropsten testnet

```
nohup geth --datadir $DATADIR \
      --unlock 0 \
      --password <(echo -n "MYPASSWORD") \
      --testnet \
       2>> $DATADIR/geth.log &
```

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

```
swarm --bzzaccount $BZZKEY \
      --datadir $DATADIR \
      --keystore $DATADIR/testnet/keystore \
      --ens-api $DATADIR/testnet/geth.ipc \
      2>> $DATADIR/swarm.log < <(echo -n "MYPASSWORD") &
```

Adding enodes manually

Eventually automatic node discovery will be working for swarm nodes. Until then you can start off the connection process by adding a few peers manually using the `admin.addPeer` console command.

```
geth --exec='admin.addPeer("ENODE")' attach ipc:/path/to/bzzd.ipc
```

Where ENODE is one of the following:

```
enode://
↪01f7728a1ba53fc263bcfbc2acacc07f08358657070e17536b2845d98d1741ec2af00718c79827dfdbecf5cfcd77965
↪68.194.101:30400
enode://
↪6d9102dd1bebb823944480282c4ba4f066f8dcf15da513268f148890ddea42d7d8afa58c76b08c16b867a58223f2b56
↪68.194.101:30427
enode://
↪fca15e2e40788e422b6b5fc718d7041ce395ff65959f859f63b6e4a6fe5886459609e4c5084b1a036ceca43e3eec6a9
↪68.194.101:30428
enode://
↪b795d0c872061336fea95a530333ee49ca22ce519f6b9bf1573c31ac0b62c99fe5c8a222dbc83d4ef5dc9e2dfb816fd
↪68.194.101:30429
enode://
↪756f582f597843e630b35371fc080d63b027757493f00df91dd799069cfc6cb52ac4d8b1a56b973baf015dd0e9182ea
↪68.194.101:30430
enode://
↪d9ccde9c5a90c15a91469b865ffd81f2882dd8731e8cbcd9a493d5cf42d875cc2709ccbc568cf90128896a165ac7a0b
↪68.194.101:30431
enode://
↪65382e9cd2e6ffdf5a8fb2de02d24ac305f1cd014324b290d28a9fba859fcd2ed95b8152a99695a6f2780c342b9815d
↪68.194.101:30433
enode://
↪7e09d045cc1522e86f70443861dceb21723fad5e2eda3370a5e14747e7a8a61809fa6c11b37b2ecf1d5aab44976375b
↪68.194.101:30434
enode://
↪bd8c3421167f418ecbb796f843fe340550d2c5e8a3646210c9c9d747bbd34d29398b3e3716ee76aa3f2fc46d325eb68
↪68.194.101:30435
enode://
↪8bb7fb70b80f60962c8979b20905898f8f6172ae4f6a715b89712cb7e965bfaab9aa0abd74c7966ad68892860481507
↪68.194.101:30436
```

Swarm in singleton mode

To launch in singleton mode, start geth using `--maxpeers 0`

```
nohup geth --datadir $DATADIR \
      --unlock 0 \
      --password <(echo -n "MYPASSWORD") \
      --verbosity 4 \
      --networkid 322 \
      --nodiscover \
      --maxpeers 0 \
       2>> $DATADIR/geth.log &
```

and launch the swarm; connecting it to the geth node. For consistency, let's use the same network id 322 as geth.

```
swarm --bzzaccount $BZZKEY \
      --datadir $DATADIR \
      --ens-api $DATADIR/geth.ipc \
      --verbosity 4 \
      --maxpeers 0 \
      --bzznetworkid 322 \
      2>> $DATADIR/swarm.log < <(echo -n "MYPASSWORD") &
```

Note: In this example, running geth is optional, it is not strictly needed. To run without geth, simply change the ens-api flag to `--ens-api ''` (an empty string).

At this verbosity level you should see plenty(!) of output accumulating in the logfiles. You can keep an eye on the output by using the command `tail -f $DATADIR/swarm.log` and `tail -f $DATADIR/geth.log`. Note: if doing this from another terminal you will have to specify the path manually because $DATADIR will not be set.

You can change the verbosity level without restarting geth and swarm via the console:

```
geth --exec "web3.debug.verbosity(3)" attach ipc:$DATADIR/geth.ipc
geth --exec "web3.debug.verbosity(3)" attach ipc:$DATADIR/bzzd.ipc
```

Note: Following these instructions you are now running a single local swarm node, not connected to any other.

Running a private swarm

You can extend your singleton node into a private swarm. First you fire up a number of `swarm` instances, following the instructions above. You can keep the same datadir, since all node-specific into will reside under `$DATADIR/bzz-$BZZKEY/` Make sure that you create an account for each instance of swarm you want to run. For simplicity we can assume you run one geth instance and each swarm daemon process connects to that via ipc if they are on the same computer (or local network), otherwise you can use http or websockets as transport for the eth network traffic.

Once your `n` nodes are up and running, you can list all there enodes using `admin.nodeInfo.enode` (or cleaner: `console.log(admin.nodeInfo.enode)`) on the swarm console. With a shell one-liner:

```
geth --exec "console.log(admin.nodeInfo.enode)" attach /path/to/bzzd.ipc
```

Then you can for instance connect each node with one particular node (call it bootnode) by injecting `admin.addPeer(enode)` into the swarm console (this has the same effect as if you created a `static-nodes.json` file for devp2p:

```
geth --exec "admin.addPeer($BOOTNODE)" attach /path/to/bzzd.ipc
```

Fortunately there is also an easier short-cut for this, namely adding the `--bootnodes $BOOTNODE` flag when you start swarm.

These relatively tedious steps of managing connections needs to be performed only once. If you bring up the same nodes a second time, earlier peers are remembered and contacted.

Note: Note that if you run several swarm daemons locally on the same instance, you can use the same data directory ($DATADIR), each swarm will automatically use its own subdirectory corre-

sponding to the bzzaccount. This means that you can store all your keys in one keystore directory: $DATADIR/keystore.

In case you want to run several nodes locally and you are behind a firewall, connection between nodes using your external IP will likely not work. In this case, you need to substitute `[::]` (indicating localhost) for the IP address in the enode.

To list all enodes of a local cluster:

```
for i in `ls $DATADIR | grep -v keystore`; do geth --exec "console.log(admin.
↪nodeInfo.enode)" attach $DATADIR/$i/bzzd.ipc; done > enodes.lst
```

To change IP to localhost:

```
cat enodes.lst | perl -pe 's/@[\d\.]+/@[::]/' > local-enodes.lst
```

Note: Steps in this section are not necessary if you simply want to connect to the swarm testnet. Since a bootnode to the testnet is set by default, your node will have a way to bootstrap its connections.

If you want to run all these instructions in a single script, you can wrap them in something like

```
#!/bin/bash

# Working directory
cd /tmp

# Preparation
DATADIR=/tmp/BZZ/`date +%s`
mkdir -p $DATADIR
read -s -p "Enter Password. It will be stored in $DATADIR/my-password: "␣
↪MYPASSWORD && echo $MYPASSWORD > $DATADIR/my-password
echo
BZZKEY=$($GOPATH/bin/geth --datadir $DATADIR --password $DATADIR/my-password␣
↪account new | awk -F"{|}" '{print $2}')

echo "Your account is ready: "$BZZKEY

# Run geth in the background
nohup $GOPATH/bin/geth --datadir $DATADIR \
    --unlock 0 \
    --password <(cat $DATADIR/my-password) \
    --verbosity 6 \
    --networkid 322 \
    --nodiscover \
    --maxpeers 0 \
    2>> $DATADIR/geth.log &

echo "geth is running in the background, you can check its logs at "$DATADIR"/geth.
↪log"

# Now run swarm in the background
$GOPATH/bin/swarm \
    --bzzaccount $BZZKEY \
    --datadir $DATADIR \
    --ens-api $DATADIR/geth.ipc \
    --verbosity 6 \
    --maxpeers 0 \
    --bzznetworkid 322 \
    &> $DATADIR/swarm.log < <(cat $DATADIR/my-password) &
```

```
echo "swarm is running in the background, you can check its logs at "$DATADIR"/
↪swarm.log"

# Cleaning up
# You need to perform this feature manually
# USE THESE COMMANDS AT YOUR OWN RISK!
##
# kill -9 $(ps aux | grep swarm | grep bzzaccount | awk '{print $2}')
# kill -9 $(ps aux | grep geth | grep datadir | awk '{print $2}')
# rm -rf /tmp/BZZ
```

Testing SWAP

Note: Important! Please only test SWAP on a private network.

Testing SWAP on your private blockchain.

The SWarm Accounting Protocol (SWAP) is disabled by default. Use of the `--swap` flag to enable it. If it is set to true, then SWAP will be enabled. However, activating SWAP requires more than just adding the –swap flag. This is because it requires a chequebook contract to be deployed and for that we need to have ether in the main account. We can get some ether either through mining or by simply issuing ourselves some ether in a custom genesis block.

Custom genesis block

Open a text editor and write the following (be sure to include the correct BZZKEY)

```
{
"nonce": "0x0000000000000042",
  "mixhash": "0x0000000000000000000000000000000000000000000000000000000000000000",
  "difficulty": "0x4000",
  "alloc": {
    "THE BZZKEY address starting with 0x eg.␣
↪0x2f1cd699b0bf461dcfbf0098ad8f5587b038f0f1": {
    "balance": "1000000000000000000"
    }
  },
  "coinbase": "0x0000000000000000000000000000000000000000",
  "timestamp": "0x00",
  "parentHash": "0x0000000000000000000000000000000000000000000000000000000000000000
↪",
  "extraData": "Custom Ethereum Genesis Block to test Swarm with SWAP",
  "gasLimit": "0xffffffff"
}
```

Save the file as `$DATADIR/genesis.json`.

If you already have swarm and geth running, kill the processes

```
killall -s SIGKILL geth
killall -s SIGKILL swarm
```

and remove the old data from the $DATADIR and then reinitialise with the custom genesis block

```
rm -rf $DATADIR/geth $DATADIR/swarm
geth --datadir $DATADIR init $DATADIR/genesis.json
```

We are now ready to restart geth and swarm using our custom genesis block

```
nohup geth --datadir $DATADIR \
       --mine \
       --unlock 0 \
       --password <(echo -n "MYPASSWORD") \
       --verbosity 6 \
       --networkid 322 \
       --nodiscover \
       --maxpeers 0 \
        2>> $DATADIR/geth.log &
```

and launch the swarm (with SWAP); connecting it to the geth node. For consistency let's use the same network id 322 for the swarm private network.

```
swarm --bzzaccount $BZZKEY \
       --swap \
       --swap-api $DATADIR/geth.ipc \
       --datadir $DATADIR \
       --verbosity 6 \
       --ens-api $DATADIR/geth.ipc \
       --maxpeers 0 \
       --bzznetworkid 322 \
       2>> $DATADIR/swarm.log < <(echo -n "MYPASSWORD") &
```

If all is successful you will see the message "Deploying new chequebook" on the swarm.log. Once the transaction is mined, SWAP is ready.

Note: Astute readers will notice that enabling SWAP while setting maxpeers to 0 seems futile. These instructions will be updated soon to allow you to run a private swap testnet with several peers.

Mining on your private chain

The alternative to creating a custom genesis block is to earn your all your ether by mining on your private chain. You can start you geth node in mining mode using the `--mine` flag, or (in our case) we can start mining on an already running geth node by issuing the `miner.start()` command:

```
geth --exec 'miner.start()' attach ipc:$DATADIR/geth.ipc
```

There will be an initial delay while the necessary DAG is generated. You can see the progress in the geth.log file. After mining has started, you can see your balance increasing via `eth.getBalance()`:

```
geth --exec 'eth.getBalance(eth.coinbase)' attach ipc:$DATADIR/geth.ipc
# or
geth --exec 'eth.getBalance(eth.accounts[0])' attach ipc:$DATADIR/geth.ipc
```

Once the balance is greater than 0 we can restart `swarm` with swap enabled.

```
killall swarm
swarm --bzzaccount $BZZKEY \
     --swap \
     --swap-api $DATADIR/geth.ipc \
     --datadir $DATADIR \
     --verbosity 6 \
     --ens-api $DATADIR/geth.ipc \
```

```
    --maxpeers 0 \
    2>> $DATADIR/swarm.log < <(echo -n "MYPASSWORD") &
```

Note: without a custom genesis block the mining difficulty may be too high to be practical (depending on your system). You can see the current difficulty with `admin.nodeInfo`

```
geth --exec 'admin.nodeInfo' attach ipc:$DATADIR/geth.ipc | grep difficulty
```

Configuration

Command line options for swarm

The swarm swarm daemon has the following swarm specific command line options:

`--bzzconfig value` Swarm config file path (datadir/bzz) The swarm config file is a json encoded format, the setting in there are documented in the following section

`--swap` Swarm SWAP enabled (default false). The SWAP (Swarm accounting protocol) is switched on by default in the current release.

`--bzznosync` Swarm Syncing disabled (default false) This option will be deprecated. It is only for testing.

`--bzzport value` Swarm local http api port (default 8500) Useful if you run multiple swarm instances and want to expose their own http proxy.

`--bzzaccount value` Swarm account key The base account that determines the node's swarm base address. This address determines which chunks are stored and retrieved at the node and therefore must not to be changed across sessions.

`--chequebook value` chequebook contract address the chequebook contract is automatically deployed on the connected blockchain if it doesn't exist. it is recorded in the config file, hence specifying it is rarely needed.

The rest of the flags are not swarm specific.

Configuration options

This section lists all the options you can set in the swarm configuration file.

The default location for the swarm configuration file is `<datadir>/swarm/bzz-<baseaccount>/config.json`. Thus continuing from the previous section, the configuration file would be

```
$DATADIR/swarm/bzz-$BZZKEY/config.json
```

It is possible to specify a different config file when launching swarm by using the *–bzzconfig* flag.

Note: The status of this project warrants that there will be potentially a lot of changes to these options.

Main parameters

Path (`<datadir>/bzz-<$BZZKEY>/`) swarm data directory

Port (8500) port to run the http proxy server

PublicKey Public key of your swarm base account

BzzKey Swarm node base address ($hash(PublicKey)$). This is used to decide storage based on radius and routing by kademlia.

EnsRoot (0xd344889e0be3e9ef6c26b0f60ef66a32e83c1b69) Ethereum Name Service contract address

Storage parameters

ChunkDbPath (`<datadir>/bzz-<$BZZKEY>/chunks`) leveldb directory for persistent storage of chunks

DbCapacity (5000000) chunk storage capacity, number of chunks (5M is roughly 20-25GB)

CacheCapacity (5000) Number of recent chunks cached in memory

Radius (0) Storage Radius: minimum proximity order (number of identical prefix bits of address key) for chunks to warrant storage. Given a storage radius r and total number of chunks in the network n, the node stores $n * 2^{-r}$ chunks minimum. If you allow b bytes for guaranteed storage and the chunk storage size is c, your radius should be set to $int(log_2(nc/b))$

Chunker/bzzhash parameters

Branches (128) Number of branches in bzzhash merkle tree. $Branches * ByteSize(Hash)$ gives the datasize of chunks. This option will be removed in a later release

Hash (SHA3) The hash function used by the chunker (base hash algo of bzzhash): SHA3 or SHA256 This option will be removed in a later release.

Synchronisation parameters

These parameters are likely to change in POC 0.3

KeyBufferSize (1024) In-memory cache for unsynced keys

SyncBufferSize (128) In-memory cache for unsynced keys

SyncCacheSize (1024) In-memory cache for outgoing deliveries

SyncBatchSize (128) Maximum number of unsynced keys sent in one batch

SyncPriorities ([3, 3, 2, 1, 1]) Array of 5 priorities corresponding to 5 delivery types <delivery, propagation, deletion, history, backlog>. Specifying a monotonically decreasing list of priorities is highly recommended.

SyncModes ([true, true, true, true, false]) A boolean array specifying confirmation mode ON corresponding to 5 delivery types: <delivery, propagation, deletion, history, backlog>. Specifying true for a type means all deliveries will be preceeded by a confirmation roundtrip: the hash key is sent first in an unsyncedKeysMsg and delivered only if confirmed in a deliveryRequestMsg.

Hive/Kademlia parameters

These parameters are likely to change in POC 0.3

CallInterval (1s) Time elapsed before attempting to connect to the most needed peer

BucketSize (3) Maximum number of active peers in a kademlia proximity bin. If new peer is added, the worst peer in the bin is dropped.

MaxProx (10) Highest Proximity order (i.e., Maximum number of identical prefix bits of address key) considered distinct. Given the total number of nodes in the network N, MaxProx should be larger than $log_2(N/ProxBinSize)$), safely $log_2(N)$.

ProxBinSize (8) Number of most proximate nodes lumped together in the most proximate kademlia bin

KadDbPath (<datadir>/bzz/bzz-<BZZKEY>/bzz-peers.json) json file path storing the known bzz peers used to bootstrap kademlia table.

SWAP parameters

BuyAt ($2 * 10^{10}$ wei) highest accepted price per chunk in wei

SellAt ($2 * 10^{10}$ wei) offered price per chunk in wei

PayAt (100 chunks) Maximum number of chunks served without receiving a cheque. Debt tolerance.

DropAt (10000) Maximum number of chunks served without receiving a cheque. Debt tolerance.

AutoCashInterval ($3 * 10^{11}$, 5 minutes) Maximum Time before any outstanding cheques are cashed

AutoCashThreshold ($5 * 10^{13}$) Maximum total amount of uncashed cheques in Wei

AutoDepositInterval ($3 * 10^{11}$, 5 minutes) Maximum time before cheque book is replenished if necessary by sending funds from the baseaccount

AutoDepositThreshold ($5 * 10^{13}$) Minimum balance in Wei required before replenishing the cheque book

AutoDepositBuffer (10^{14}) Maximum amount of Wei expected as a safety credit buffer on the cheque book

PublicKey (PublicKey(bzzaccount)) Public key of your swarm base account use

Contract Address of the cheque book contract deployed on the Ethereum blockchain. If blank, a new chequebook contract will be deployed.

Beneficiary (Address(PublicKey)) Ethereum account address serving as beneficiary of incoming cheques

By default, the config file is sought under `<datadir>/bzz/bzz-<$BZZKEY>/config.json`. If this file does not exist at startup, the default config file is created which you can then edit (the directories on the path will be created if necessary). In this case or if `config.Contract` is blank (zero address), a new chequebook contract is deployed. Until the contract is confirmed on the blockchain, no outgoing retrieve requests will be allowed.

Setting up SWAP

SWAP (Swarm accounting protocol) is the system that allows fair utilisation of bandwidth (see Incentivisation, esp. SWAP – Swarm Accounting Protocol). In order for SWAP to be used, a chequebook contract has to have been deployed. If the chequebook contract does not exist when the client is launched or if the contract specified in the config file is invalid, then the client attempts to autodeploy a chequebook:

> [BZZ] SWAP Deploying new chequebook (owner: 0xe10536.. .5e491)

If you already have a valid chequebook on the blockchain you can just enter it in the config file `Contract` field.

You can set a separate account as beneficiary to which the cashed cheque payment for your services are to be credited. Set it on the `Beneficiary` field in the config file.

Autodeployment of the chequebook can fail if the baseaccount has no funds and cannot pay for the transaction. Note that this can also happen if your blockchain is not synchronised. In this case you will see the log message:

```
[BZZ] SWAP unable to deploy new chequebook: unable to send chequebook     creation␣
↪transaction: Account
 does not exist or account     balance too low..  .retrying in 10s

[BZZ] SWAP arrangement with <enode://23ae0e62..  ..  ..  8a4c6bc93b7d2aa4fb@195.
↪228.155.76:30301>: purchase from peer disabled; selling to peer disabled)
```

Since no business is possible here, the connection is idle until at least one party has a contract. In fact, this is only enabled for a test phase. If we are not allowed to purchase chunks, then no outgoing requests are allowed. If we still try to download content that we dont have locally, the request will fail (unless we have credit with other peers).

```
[BZZ] netStore.startSearch: unable to send retrieveRequest to peer [<addr>]:␣
↪[SWAP] <enode://23ae0e62..  ..  ..  8a4c6bc93b7d2aa4fb@195.228.155.76:30301> we␣
↪cannot have debt (unable to buy)
```

Once one of the nodes has funds (say after mining a bit), and also someone on the network is mining, then the autodeployment will eventually succeed:

```
[CHEQUEBOOK] chequebook deployed at 0x77de9813e52e3a..  .c8835ea7 (owner:␣
↪0xe10536ae628f7d6e319435ef9b429dcdc085e491)
[CHEQUEBOOK] new chequebook initialised from 0x77de9813e52e3a..  .c8835ea7 (owner:␣
↪0xe10536ae628f7d6e319435ef9b429dcdc085e491)
[BZZ] SWAP auto deposit ON for 0xe10536 -> 0x77de98: interval = 5m0s, threshold =␣
↪50000000000000, buffer = 100000000000000)
[BZZ] Swarm: new chequebook set: saving config file, resetting all connections in␣
↪the hive
[KΛÐ]: remove node enode://23ae0e6..  .aa4fb@195.228.155.76:30301 from table
```

Once the node deployed a new chequebook, its address is set in the config file and all connections are reset with the new conditions. Purchase in one direction should be enabled. The logs from the point of view of the peer with no valid chequebook:

```
[CHEQUEBOOK] initialised inbox (0x9585..  .3bceee6c -> 0xa5df94be..  .bbef1e5)␣
↪expected signer: 041e18592..  ..  ..  702cf5e73cf8d618
[SWAP] <enode://23ae0e62..  ..  ..  8a4c6bc93b7d2aa4fb@195.228.155.76:30301>   ␣
↪set autocash to every 5m0s, max uncashed limit: 50000000000000
[SWAP] <enode://23ae0e62..  ..  ..  8a4c6bc93b7d2aa4fb@195.228.155.76:30301>   ␣
↪autodeposit off (not buying)
[SWAP] <enode://23ae0e62..  ..  ..  8a4c6bc93b7d2aa4fb@195.228.155.76:30301>   ␣
↪remote profile set: pay at: 100, drop at: 10000,    buy at: 20000000000, sell␣
↪at: 20000000000
[BZZ] SWAP arrangement with <enode://23ae0e62..  ..  ..  8a4c6bc93b7d2aa4fb@195.
↪228.155.76:30301>: purchase from peer disabled;   selling to peer enabled at␣
↪20000000000 wei/chunk)
```

Depending on autodeposit settings, the chequebook will be regularly replenished:

```
[BZZ] SWAP auto deposit ON for 0x6d2c5b -> 0xefbb0c:
 interval = 5m0s, threshold = 50000000000000,
 buffer = 100000000000000)
 deposited 100000000000000 wei to chequebook (0xefbb0c0..  .16dea,  balance:␣
↪100000000000000, target: 100000000000000)
```

The peer with no chequebook (yet) should not be allowed to download and thus retrieve requests will not go out. The other peer however is able to pay, therefore this other peer can retrieve chunks from the first peer and pay for them. This in turn puts the first peer in positive, which they can then use both to (auto)deploy their own chequebook and to pay for retrieving data as well. If they do not deploy a chequebook for whatever reason, they can use their balance to pay for retrieving data, but only down to 0 balance; after that no more requests are allowed to go out. Again you will see:

```
[BZZ] netStore.startSearch: unable to send retrieveRequest to peer [aff89da0c6...
↪623e5671c01]: [SWAP]  <enode://23ae0e62...8a4c6bc93b7d2aa4fb@195.228.155.
↪76:30301> we cannot have debt (unable to buy)
```

If a peer without a chequebook tries to send requests without paying, then the remote peer (who can see that they have no chequebook contract) interprets this as adverserial behaviour resulting in the peer being dropped.

Following on in this example, we start mining and then restart the node. The second chequebook autodeploys, the peers sync their chains and reconnect and then if all goes smoothly the logs will show something like:

```
initialised inbox (0x95850c6..  .bceee6c -> 0xa5df94b..  .bef1e5) expected signer:␣
↪041e185925bb..  ..  ..  702cf5e73cf8d618
[SWAP] <enode://23ae0e62..  ..  ..  8a4c6bc93b7d2aa4fb@195.228.155.76:30301> set␣
↪autocash to every 5m0s, max uncashed limit: 50000000000000
[SWAP] <enode://23ae0e62..  ..  ..  8a4c6bc93b7d2aa4fb@195.228.155.76:30301> set␣
↪autodeposit to every 5m0s, pay at: 50000000000000, buffer: 100000000000000
[SWAP] <enode://23ae0e62..  ..  ..  8a4c6bc93b7d2aa4fb@195.228.155.76:30301>␣
↪remote profile set: pay at: 100, drop at: 10000, buy at: 20000000000, sell at:␣
↪20000000000
[SWAP] <enode://23ae0e62..  ..  ..  8a4c6bc93b7d2aa4fb@195.228.155.76:30301>␣
↪remote profile set: pay at: 100, drop at: 10000, buy at: 20000000000, sell at:␣
↪20000000000
[BZZ] SWAP arrangement with <node://23ae0e62...8a4c6bc93b7d2aa4fb@195.228.155.
↪76:30301>: purchase from peer enabled at 20000000000 wei/chunk; selling to peer␣
↪enabled at 20000000000 wei/chunk)
```

As part of normal operation, after a peer reaches a balance of `PayAt` (number of chunks), a cheque payment is sent via the protocol. Logs on the receiving end:

```
[CHEQUEBOOK] verify cheque: contract: 0x95850..  .eee6c, beneficiary:␣
↪0xe10536ae628..  .cdc085e491, amount: 868020000000000,signature: a7d52dc744b8.. ␣
↪..  ..  f1fe2001 - sum: 866020000000000
[CHEQUEBOOK] received cheque of 2000000000000 wei in inbox (0x95850..  .eee6c,␣
↪uncashed: 42000000000000)
```

The cheque is verified. If uncashed cheques have an outstanding balance of more than `AutoCashThreshold`, the last cheque (with a cumulative amount) is cashed. This is done by sending a transaction containing the cheque to the remote peer's cheuebook contract. Therefore in order to cash a payment, your sender account (baseaddress) needs to have funds and the network should be mining.

```
[CHEQUEBOOK] cashing cheque (total: 1040000000000000) on chequebook (0x95850c6..  .
↪eee6c) sending to 0xa5df94be..  .e5aaz
```

For further fine tuning of SWAP, see SWAP parameters.

Chapter 5

Usage

Using swarm from the command line

Uploading a file or directory to the swarm

Make sure you have compiled the swarm command

```
cd $GOPATH/src/github.com/ethereum/go-ethereum
go install ./cmd/swarm
```

The *swarm up* subcommand makes it easy to upload files and directories. Usage:

```
swarm up /path/to/file/or/directory
```

By default this assumes that you are running your own swarm node with a local http proxy on the default port (8500). See Running the swarm client to learn how to run a local node. It is possible to specify alternative proxy endpoints with the `--bzzapi` option.

You can use one of the public gateways as a proxy, in which case you can upload to swarm without even running a node.

Note: This treat is likely to disappear or be seriously restricted in the future.

```
swarm --bzzapi http://swarm-gateways.net up /path/to/file/or/directory
```

Example: uploading a file

Issue the following command to upload the go-ethereum README file to your swarm

```
swarm up  $GOPATH/src/github.com/ethereum/go-ethereum/README.md
```

It produces the following output

```
I1214 15:04:43.011654 upload.go:171] uploading file README.md (166 bytes)
I1214 15:04:48.167952 upload.go:180] uploading manifest
d1f25a870a7bb7e5d526a7623338e4e9b8399e76df8b634020d11d969594f24a
```

The first two lines are just debug logs, you can make them vanish by redirecting standard error to a file.

```
swarm up README.md 2> up.log
d1f25a870a7bb7e5d526a7623338e4e9b8399e76df8b634020d11d969594f24a
```

The hash returned is the swarm hash of a manifest that contains the README.md file as its only entry. So by default both the primary content and the manifest is uploaded. You can access this 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 may wish to only upload the content and maybe include it in a custom index, or it is handled as a datablob known and used only by some application that knows its mimetype. For this you can set *–manifest=false*:

```
swarm --manifest=false --bzzapi http://swarm-gateways.net/ up sw\^3.pdf 2> up.log
{
  "entries": [
    {
      "hash": "6a18222637cafb4ce692fa11df886a03e6d5e63432c53cbf7846970aa3e6fdf5",
      "contentType": "application/pdf"
    }
  ]
}
```

This option supresses automatic manifest upload, instead upon uploading the file, it displays the manifest. As you see the single entry is the file itself, while the manifest contains the content type that will be given to the browser. The manifest here acts as an extension of the primary content with metadata. In future, other HTTP headers will also be supported.

Example: Uploading a directory

Uploading directories is achieved with `swarm --recursive up`.

Let us create some test files

```
mkdir upload-test
echo "one" > upload-test/one.txt
echo "two" > upload-test/two
mkdir upload-test/three
echo "four" > upload-test/three/four
```

We can upload this directory with

```
swarm --recursive up upload-test/
```

The output should look something like

```
uploading file upload-test/one.txt (4 bytes)
uploading file upload-test/three/four (5 bytes)
uploading file upload-test/two (4 bytes)
uploading manifest
{
  "hash": "6c64ae708609be4cc34027b38b1104f0ea8dafd5164343117ce421f7714b5e98",
  "entries": [
    {
      "hash": "e57619a0be1101b948afc89dcfb9ce430f38fba9be19fd0a3ed7424d500340a4",
      "contentType": "text/plain; charset=utf-8",
      "path": "one.txt"
    },
    {
      "hash": "8cc6a12255e553fc8d8b25b309186981b1fd458d2be41bcc099f148c167839ec",
      "path": "three/four"
    },
```

```
    {
      "hash": "2940c27ab5409f9ffa0074c4c81c01ab6f165ac0ae973cd03212068013b3b6f3",
      "path": "two"
    }
  ]
}
```

You could then retrieve the files relative to the root manifest like so:

```
http://localhost:8500/bzz:/
↪6c64ae708609be4cc34027b38b1104f0ea8dafd5164343117ce421f7714b5e98/three/four
```

if you'd like to be able to access your content via a human readable name like 'mysite.eth' intead of the long hex string above, see the section on Ethereum Name Service below.

Content retrieval: hashes and manifests

Retrieving content using the http proxy

As indicated above, your local swarm instance has an http interface running on port 8500 (by default). Retrieving content is simple matter of pointing your browser to

```
GET http://localhost:8500/bzz:/HASH
```

where HASH is the id of a swarm manifest. This is the most common usecase whereby swarm can serve the web.

Disregarding the clunky proxy part, it looks like http transfering content from servers, but in fact it is using swarm's serverless architecture.

The general pattern is:

```
<HTTP proxy>/<URL SCHEME>:/<DOMAIN OR HASH>/<PATH>?<QUERY_STRING>
```

The http proxy part can be eliminated if you register the appropriate scheme handler with your browser or you use Mist.

Swarm offers 3 distinct url schemes:

bzz url schemes

bzz

Example:

```
GET http://localhost:8500/bzz:/theswarm.test
```

The bzz scheme assumes that the domain part of the url points to a manifest. When retrieving the asset adderessed 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.

This generic scheme supports name resolution for domains registered on the Ethereum Name Service (ENS, see Ethereum Name Service). This is a read-only scheme meaning that it only supports GET requests and serves to retrieve content from swarm.

bzzi (immutable)

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

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

bzzr (raw)

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

When responding to GET requests to the bzzr scheme, swarm does not assume a manifest just serves the asset addressed by the url directly.

The `content_type` query parameter can be supplied to specify the mime you want otherwise content is served as a default octet stream. 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/bzzr:/
↪6a18222637cafb4ce692fa11df886a03e6d5e63432c53cbf7846970aa3e6fdf5?content_
↪type=application/pdf
```

Importantly and somewhat unusually for generic schemes, the raw scheme supports POST and PUT requests. This is a crucially important way in which swarm is different from the internet as we know it.

The possibility to POST makes swarm an actual cloud service, bringing upload functionality to your browsing.

In fact the command line tool `swarm up` uses the http proxy with the bzz raw scheme under the hood.

Manifests

In general manifests declare a list of strings associated with swarm hashes. Before we get into generalities however, let us begin with an introductory example.

This is demonstrated by the following example. Let's create 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.

```
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 this manifest:

```
# swarm --manifest=false --recursive --defaultpath orange-papers/index.html --
↪bzzapi http://swarm-gateways.net/ up orange-papers/ 2> up.log
{
  "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"
    },
    {
      "hash": "6a18222637cafb4ce692fa11df886a03e6d5e63432c53cbf7846970aa3e6fdf5",
      "contentType": "application/pdf",
      "path": "sw^3.pdf"
    }
  ]
}
```

We can see the retrieve the manifest directly (instead of the files they refer to) by using the bzz-raw protocol `bzzr`:

```
wget -O - "http://localhost:8500/bzzr:/
↪2477cc8584cc61091b5cc084cdcdb45bf3c6210c263b0143f030cf7d750e894d"
```

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.

```
http://localhost:8500/bzzr:/
↪2477cc8584cc61091b5cc084cdcdb45bf3c6210c263b0143f030cf7d750e894d?content_type=
↪"text/plain"
```

Now you can also check that the manifest hashes to the content (in fact swarm does it for you):

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

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

Path Matching on Manifests

A useful feature of manifests is that urls can be matched on the paths. Directory trees, routing tables and database indexes all share this problem. In some sense this makes the manifest a routing table and so the manifest swarm entry acts as if it were 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 at the domain, which is the manifest above. The url path `sw^3` is matched against the entries. In this case a perfect match is found and the document at 6a182226... is served as a pdf.

As you 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`.

Ethereum Name Service

ENS is the system that Swarm uses to permit content to be referred to by a human-readable name, such as "orangepapers.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://orange-papers.eth/*.

If we take our earlier example and set the hash 2477cc85... as the content hash for the domain '' orangepapers.eth'', we can request:

```
GET http://localhost:8500/bzz:/orange-papers.eth/sw^3.pdf
```

and get served the same as with:

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

Full documentation on ENS is available here.

If you just want to set up ENS so that you can host your Swarm content on a domain, here's a quick set of steps to get you started.

Content Retrieval using ENS

The default configuration of swarm is to use names registered on the Ropsten testnet. In order for you to be able to resolve names to swarm hashes, all that needs to happen is that your swarm client is connected to a geth node synced on the Ropsten testnet. See section "Running the swarm client" here.

Registering names for your swarm content

There are several steps involved in registering a new name and assigning a swarm hash to it. To start off, you'll need to register a domain, then you need to assign a resolver to the domain and then you add the swarm hash to the resolver.

Note: The ENS system will let you register even invalid names - names with upper case characters, or prohibited unicode characters, for instance - but your browser will never resolve them. As a result, take care to make sure any domain you try to register is well-formed before registering it

Preparation

The first step to take is to download ensutils.js (direct link).

You should of course have geth running and connected to ropsten (*geth –testnet*). Connect to the geth console:

```
./geth attach ipc:/path/to/geth.ipc
```

Once inside the console, run:

```
loadScript('/path/to/ensutils.js')
```

Note: You can leave the console at any time by pressing ctrl+D

Registering a .test domain

The easiest option is to register a .test domain. These domains can be registered by anyone at any time, but they automatically expire after 28 days.

We will be sending transactions on Ropsten, so if you have not already done so, get yourself some ropsten testnet ether. You can get some for free here.

Before being able to send transaction, you will need to unlock your account using `personal.unlockAccount(account)` i.e.

```
personal.unlockAccount(eth.accounts[0])
```

Then, still inside the geth console (with ensutils.js loaded) type the following (replacing MYNAME with the name you wish to register):

```
testRegistrar.register(web3.sha3('MYNAME'), eth.accounts[0], {from: eth.
↪accounts[0]});
```

Note: Warning: do not register names with UPPER CASE letters. The ENS will let you register them, but your browser will never resolve them.

The output will be a transaction hash. Once this transaction is mined on the testnet you can verify that the name MYNAME.test belongs to you:

```
eth.accounts[0] == ens.owner(namehash('MYNAME.test'))
```

Registering a .eth domain

Registering a .eth domain is more involved. If you're just wanting to test things out quickly, start with a .test domain. The .eth domains take a while to register, as they use an auction system, (while .test domains can be registered instantly, but only persist for 28 days). Further, .eth domains are also restricted to being at least 7 characters long. For complete documentation see here.

Just as when registering a .test domain, you will need testnet ether and you must unlock your account. Then you may start bidding on a domain.

Quick Reference:

1. Prepare:

```
personal.unlockAccount(eth.accounts[0])
loadScript('/path/to/ensutils.js')
```

2. Make a bid:

```
bid = ethRegistrar.shaBid(web3.sha3('myname'), eth.accounts[0], web3.toWei(1,
↪'ether'), web3.sha3('secret'));
```

3. Reveal your bid:

```
ethRegistrar.unsealBid(web3.sha3('myname'), eth.accounts[0], web3.toWei(1, 'ether
↪'), web3.sha3('secret'), {from: eth.accounts[0], gas: 500000});
```

4. Finalise:

```
ethRegistrar.finalizeAuction(web3.sha3('myname'), {from: eth.accounts[0], gas:␣
↪500000});
```

For info on how to increase your bids, check the current highest bid, check when an auction ends, check if a name is available in the first place and more please consult the official documentation.

Setting up a resolver

The next step is to set up a resolver for your new domain name. While it's possible to write and deploy your own custom resolver, for everyday use with Swarm, a general purpose one is provided, and is already deployed on the testnet.

On the geth (testnet) console:

```
loadScript('/path/to/ensutils.js')
personal.unlockAccount(eth.accounts[0], "")
ens.setResolver(namehash('MYNAME.test'), publicResolver.address, {from: eth.
↪accounts[0], gas: 100000});
```

Registering a swarm hash on the publicResolver

Finally, after uploading your content to Swarm as detailed above, you can update your site with this command:

```
publicResolver.setContent(namehash('MYNAME.test'), 'HASH', {from: eth.accounts[0],␣
↪gas: 100000})
```

Again, replace 'MYNAME.test' with the name you registered, and replace 'HASH' with the hash you got when uploading your content to swarm, starting with 0x.

After this has executed successfully, anyone running a correctly configured and synchronised Swarm client will be able to access the current version of your site on *bzz://MYNAME.test/*.

```
http://localhost:8500/bzz:/MYNAME.test
```

Looking up names in the ENS manually

After registering your names and swarm hashes, you can check that everything is updated correctly by looking up the name manually.

Connect to the geth console and load ensutils.js just as before. Then type

```
getContent('MYNAME.test')
```

You can also check this in your swarm console with:

```
bzz.resolve('MYNAME.test')
```

If everything worked correctly, it will return the hash you specified when you called *setContent* earlier.

Updating your content

Each time you update your site's content afterwards, you only need to repeat the last step to update the mapping between the name you own and the content you want it to point to. Anyone visiting your site by its name will always see the version you most recently updated using *setHash*, above.

```
publicResolver.setContent(namehash('MYNAME.test'), 'NEWHASH', {from: eth.
↪accounts[0], gas: 100000})
```

The HTTP API

GET http://localhost:8500/bzz:/domain/some/path retrieve document at domain/some/path allowing domain to resolve via The Ethereum Name Service

GET http://localhost:8500/bzzi:/HASH/some/path retrieve document at HASH/some/path where HASH is a valid swarm hash

GET http://localhost:8500/bzzr:/domain/some/path retrieve the raw content at domain/some/path allowing domain to resolve via The Ethereum Name Service

POST http://localhost:8500/bzzr: The post request is the simplest upload method. Direct upload of files - no manifest is created. It returns the hash of the uploaded file

PUT http://localhost:8500/bzz:/HASH | domain/some/path The PUT request publishes the uploaded asset to the manifest. It looks for the manifest by domain or hash, makes a copy of it and updates its collection with the new asset. It returns the hash of the newly created manifest.

Swarm IPC API

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

Note: Note that this is not the recommended way for users or dapps to interact with swarm and is only meant for debugging ad testing purposes. Given that this module offers local filesystem access,

allowing dapps to use this module or exposing it via remote connections creates a major security risk. For this reason `swarm` only exposes this api via local ipc (unlike geth not allowing websockets or http).

The API offers the following methods:

`bzz.upload(localfspath, defaultfile)` uploads the file or directory at `localfspath`. The second optional argument specifies the path to the file which will be served when the empty path is matched. It is common to match the empty path to `index.html`

it returns content hash of the manifest which can then be used to download it.

`bzz.download(bzzpath, localdirpath)` it recursively downloads all the paths starting from the manifest at `bzzpath` and downloads them in a corresponding directory structure under `localdirpath` using the slashes in the paths to indicate subdirectories.

assuming `dirpath.orig` is the root of any aribitrary directory tree containing no soft links or special files, uploading and downloading will result in identical data on your filesystem:

bzz.download(bzz.upload(dirpath.orig), dirpath.replica) diff -r dirpath.orig dirpath.replica || echo "identical"

`bzz.put(content, contentType)` can be used to push a raw data blob to swarm. Creates a manifest with an entry. This entry has the empty path and specifies the content type given as second argument. It returns content hash of this manifest.

`bzz.get(bzzpath)` It downloads the manifest at `bzzpath` and returns a response json object with content, mime type, status code and content size. This should only be used for small pieces of data, since the content gets instantiated in memory.

`bzz.resolve(domain)` resolves the domain name to a content hash using ENS and returns that. If swarm is not connected to a blockchain it returns an error. Note that your eth backend needs to be syncronised in order to get uptodate domain resolution.

`bzz.info()` returns information about the swarm node

`bzz.hive()` outputs the kademlia table in a human-friendly table format

Mounting Swarm

Another way of intracting with Swarm is by mounting it as a local filesystem using Fuse (a.k.a swarmfs). 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.

`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.

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
```

Chequebook RPC API

Swarm also exposes an RPC API for the chequebook offering the followng methods:

`chequebook.balance()` Returns the balance of your swap chequebook contract in wei. It errors if no chequebook is set.

`chequebook.issue(beneficiary, value)` Issues a cheque to beneficiary (an ethereum address) in the amount of value (given in wei). The json structure returned can be copied and sent to beneficiary who in turn can cash it using `chequebook.cash(cheque)`. It errors if no chequebook is set.

`chequebook.cash(cheque)` Cashes the cheque issued. Note that anyone can cash a cheque. Its success only depends on the cheque's validity and the solvency of the issuers chequbook contract up to the amount specified in the cheque. The tranasction is paid from your bzz base account. Returns the transaction hash. It errors if no chequebook is set or if your account has insufficient funds to send the transaction.

`chequebook.deposit(amount)` Transfers funds of amount wei from your bzz base account to your swap chequebook contract. It errors if no chequebook is set or if your account has insufficient funds.

Example: use of the console

Uploading content

It is possible to upload files from the swarm console (without the need for swarm command or an http proxy). The console command is

```
bzz.upload("/path/to/file/or/directory", "filename")
```

The command returns the root hash of a manifest. The second argument is optional; it specifies what the empty path should resolve to (often this would be `index.html`). Proceeding as in the example above (Example: Uploading a directory). Prepare some files:

```
mkdir upload-test
echo "one" > upload-test/one.txt
echo "two" > upload-test/two
mkdir upload-test/three
echo "four" > upload-test/three/four
```

Then execute the `bzz.upload` command on the swarm console: (note `bzzd.ipc` instead of `geth.ipc`)

```
./geth --exec 'bzz.upload("upload-test/", "one.txt")' attach ipc:$DATADIR/bzzd.ipc
```

We get the output:

```
dec805295032e7b712ce4d90ff3b31092a861ded5244e3debce7894c537bd440
```

If we open this HASH in a browser

```
http://localhost:8500/bzz:/
↪dec805295032e7b712ce4d90ff3b31092a861ded5244e3debce7894c537bd440/
```

We see "one" because the empty path resolves to "one.txt". Other valid URLs are

```
http://localhost:8500/bzz:/
↪dec805295032e7b712ce4d90ff3b31092a861ded5244e3debce7894c537bd440/one.txt
http://localhost:8500/bzz:/
↪dec805295032e7b712ce4d90ff3b31092a861ded5244e3debce7894c537bd440/two
http://localhost:8500/bzz:/
↪dec805295032e7b712ce4d90ff3b31092a861ded5244e3debce7894c537bd440/three/four
```

We only recommend using this API for testing purposes or command line scripts. Since they save on http file upload, their performance is somewhat better than using the http API.

Downloading content

As an alternative to http to retrieve content, you can use `bzz.get(HASH)` or `bzz.download(HASH, /path/to/download/to)` on the swarm console (note `bzzd.ipc` instead of `geth.ipc`)

```
./geth --exec 'bzz.get(HASH)' attach ipc:$DATADIR/bzzd.ipc
./geth --exec 'bzz.download(HASH, "/path/to/download/to")' attach ipc:$DATADIR/
↪bzzd.ipc
```

Chapter 6

Architecture

This chapter is aimed at developers who want to understand the underlying concepts and design of swarm.

Contents

- *Architecture*
  - *Swarm hash*
    - *Introduction*
    - *Description*
    - *Strict interpretation*
    - *Loose interpretations*
  - *Chunker*
  - *Distributed Preimage Archive*
    - *High-level design*
    - *Requests*
  - *Syncing*
  - *Peer management (hive, kademlia)*
    - *Peer addresses*
    - *Logarithmic distance and network topology*
    - *Peer table format*
    - *Peer table update*
  - *The bzz protocol*
  - *Incentivisation*
    - *swap, swear & swindle*
    - *SWAP – Swarm Accounting Protocol*
    - *SWEAR – Storage With Enforced Archiving Rules or Swarm Enforcement And Registration*
    - *SWINDLE – Secured With INsurance Deposit Litigation and Escrow*

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

- *message*: p2p RLPx network layer. Messages are relevant for the bzz wire protocol The bzz proto-

col.

- *chunk*: fixed size data unit of storage, content-addressing, request/delivery and accounting: this is the level relevant to the entire storage layer (including localStore, DHT/netstore, bzz protocol, accounting protocol)
- *document*: in want of a better word, we call 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. This layer is handled by the DPA and its Chunker.
- *collection*: a mapping of paths to documents is represented by the *swarm manifest*. This layer has mapping to file system directory tree. Given trivial routing conventions, url can be mapped to documents in a standardised way, allowing manifests to mimic webservers on swarm. This layer is relevant to high level apis: the go API, HTTP proxy API, console and web3 JS APIs.

The actual storage layer of swarm consists of two main components, the *localstore (LOC)* and the *netstore (NET)*. The local store provides the interface to local computing resources utilised for storage. In particular we explicitly delineate an in-memory fast cache (*memory store (MEM)*) and a persistent disk storage (*dbstore (DBS)* possibly with its own cache system). The reason for this is that we can optimise the system by relying on certain properties of the memory store specific for our purpose, e.g., that keys are hashes, so no further hashing is needed, the keys can be directly mapped in a tree/trie structure.

For disk storage, leveldb is used. Both components can easily be swapped by alternative solutions with minimal work.

The netStore is the actual DHT (distributed hash table) implementation. It interacts with the bzz protocol as well as the hive, the network peer logistic manager. The netStore is really where the distributed storage logic is implemented.

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

The component that chunks the documents into the merkle tree is called the *chunker*. Our chunker implements the *bzzhash* algorithm which is parallelized tree hash based on keccak 256-bit SHA3. The DPA runs a storage loop which receives the chunks back from the chunker and dispatches them to the chunkstore for storage. This entry point is the netStore.

When a root key is handed to the DPA for document retrieval, the DPA calls the Chunker which hands back a seekable document reader to the caller. This is a *lazy reader* in the sense that it retrieves relevant parts of the underlying document only as they are actually read. This entails that partial reads (e.g., range requests on video) are supported on the lowest level. In other words this scheme provides an integrity protected random access storage of documents.

The swarm manifest is a structure that defines a mapping between arbitrary paths and documents to handle document collections. It also includes various metadata associated with the collection and the documents therein.

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.

API is the go implementation (and go API) for these high level functions. There is an http proxy interface as well as a RPC API for these functions. These all differ in their exact functionality due to inherent privilege differences or interface limitations. These are described in detail in Usage.

The SWAP – Swarm Accounting Protocol component keeps track of requests between peers and implements the accounting protocol. It is described in detail in Incentivisation.

In what follows we describe the components in more detail.

Swarm hash

Introduction

Swarm Hash (a.k.a. *bzzhash*) is a [Merkle tree](http://en.wikipedia.org/wiki/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 (including SMP and massively-parallel architectures such as GPU's) for faster calculation and verification, can be used to verify the integrity of partial content without having to transmit all of it. Proofs of security to the underlying hash function carry over to Swarm Hash.

Description

Swarm Hash is constructed using a regular hash function (in our case, Keccak 256 bit SHA3) with a generalization of Merkle's tree hash scheme. The basic unit of hashing is a *chunk*, that can be either a *leaf chunk* containing a section of the content to be hashed or an *inner chunk* containing hashes of its children, which can be of either variety.

Hashes of leaf 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](http://en.wikipedia.org/wiki/Length_extension_attack), even if the underlying hash function is not. Note that this "safety belt" measure is extensively used in the latest edition of [OpenPGP standard (IETF RFC4880)](https://tools.ietf.org/html/rfc4880). This said, Swarm Hash is still vulnerable to length extension attacks, but can be easily protected against them, when necessary, using similar measures in a higher layer. A possibly very profitable performance optimization (not currently implemented) is to initialize the hash calculation with the length of the standard chunk size (e.g. 4096 bytes), thus saving the repeated hashing thereof.

Hashes of inner chunks are defined as the hashes of the concatenation of the 64-bit length (in LSB-first order) of the content hashed by the entire (sub-) tree rooted on this chunk and the hashes of its children.

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 leaf chunk. If it is shorter than that, it is an inner chunk. Otherwise, it is not a valid Swarm Hash chunk.

Strict interpretation

A strict Swarm Hash is one where every chunk with the possible exception of those on the rightmost branch is of a specified length, i.e. 4 kilobytes. Those on the rightmost branch are no longer, but possibly shorter than this length. The hash tree must be balanced, meaning that all root-to-leaf branches are of the same length.

The strict interpretation is unique in that only one hash value matches a particular content. The strict interpretation is only vulnerable to length extension attacks if the length of the content is a multiple of the chunk size, and the number of leaf chunks is an integer power of branching size (the fix maximum chunk size divided by hash length).

Two [parallelized implementationd are available in Go](https://github.com/ethereum/go-ethereum/tree/develop/swarm/storage/) is available as well as [a command-line tool](https://github.com/ethereum/go-ethereum/tree/develop/cmd/bzzhash) for hashing files on the local filesystem using the strict interpretation.

Loose interpretations

Swarm Hash interpreted less strictly may allow for different tree structures, imposing fewer restrictions or none at all. In this way, different hash values can resolve to the same content, which might have some adverse security implications.

However, it might open the door for different applications where this does not constitute a vulnerability. For example, accepting single-leaf hashes in addition to strict Swarm hashes allows for referencing files without having to implement the whole thing.

Chunker

Chunker is the interface to a component that is responsible for disassembling and assembling larger data. It relies on the underlying chunking model. This module is pluggable, the current implementation uses the *Treechunker* which implements *bzzhash*. An alternative implementation is the *pyramid* chunker that is more memory efficient for larger data and does not need the size of the file, so in principle is able to encode live streams on the fly.

When *splitting* a document, the chunker pushes the resulting chunks to the DPA that delegates them to storage layers (implementing ChunkStore interface) and returns the *root hash* of the document. After getting notified that all the data has been split (the error channel is closed), the caller can safely read or save the root key. Otherwise it times out if not all chunks get stored or not the entire stream of data has been processed. By inspecting the errc channel the caller can check if any explicit errors (typically IO read/write failures) occured during splitting.

When *joining* a document, the chunker needs the root key and returns a *lazy reader*. While joining, the chunker pushes chunk requests to the DPA that delegates them to chunk stores and notify the chunker if the data has been delivered (i.e. retrieved from memory cache, disk-persisted db or cloud based swarm delivery). The chunker then puts these together on demand as and where the reader is read.

The chunker works in a simple way, it builds a tree out of the document so that each node either represents a chunk of real data or a chunk of data representing a branching non-leaf node of the tree. In particular each such non-leaf chunk will represent a concatenation of the hashes of its respective children. This scheme simultaneously guarantees data integrity as well as self addressing. The *maximum chunk size* is currently 4096 which comes from the multiple of configurable parameters :option:Branches and :option:Hash. In addition to the data, the chunk contains the size of the subtree it encodes. Abstract nodes are transparent since their represented size component is strictly greater than their maximum data size, since they encode a subtree. Since the size is represented by a 64bit integer (8 bytes), the stored size of a chunk is at most 4104 bytes.

Distributed Preimage Archive

DPA (distributed preimage archive) stores small pieces of information (preimage objects, arbitrary strings of bytes of limited length) retrievable by their (cryptographic) hash value. Thus, preimage objects stored in DPA have implicit integrity protection. The hash function used for key assignment is assumed to be collision-free, meaning that colliding keys for different preimage objects are assumed to be practically impossible.

DPA serves as a fast, redundant store optimized for speedy retrieval and long-term reliability. Since the key is derived from the preimage, there is no sense in which we can talk about multiple or alternative values for keys, the store is immutable.

High-level design

DPA is organized as a *DHT (Distributed Hash Table)*: each participating node has an address (resolved into a network address by the p2p layer) coming from the same value set as the range of the hash

function. In particular it is the hash of the ethereum address of the node's base account.

There is a *distance measure* defined over this value set that is a proper metric satisfying the triangle inequality. It is always possible to tell how far another node or another preimage object is from a given address or hash value. The distance from self is zero.

Each node is interested in being able to find preimages to hash values as fast as possible and therefore stores as many preimages as it can itself. Each node ends up storing preimage objects within a given radius limited by available storage capacity. The cryptographic hash function takes care of randomization and fair load balancing.

On a high level, nodes should provide the following services through a public network protocol:

- Inserting new preimages into DPA
- Retrieving preimages from their own storage, if they have it.
- Sharing routing information to a given node address

Requests

When receiving a preimage that is not already present in its local storage, the node stores it locally. If the storage allocated by the node for the archive is full, the object accessed the longest time ago is discarded. Note that this policy implicitly results in storing the objects closer to the node's address, as - all else being equal - those are the ones which are most likely to be requested from this particular node, due to the lookup strategy detailed below.

After storing the preimage, the store request is also forwarded to all the nodes in the corresponding row of the routing table. Note that the kademlia routing makes sure that the row in the close proximity of a node actually contains nodes further out than self thereby taking care of storage redundancy.

A retrieval request for a key arrives with a key recently unseen. It is looked up in local store and if not found, it is assessed if it is worth having, or if its proximity warrants its storage or not. If deemed too distant it can be forgotten, if within our storage radius then we open a request entry in the request pool. Further requests in the near future asking for the same key will check its status with this entry.

Immediately upon receiving the request, the target is mapped to its kademlia proximity bin and the peers in the bin are ordered by proximity to the target. The request is forwarded to the first connected peer.

Various fallback strategies and parallel request forwarding will be implemented as of POC 0.4.

From the set up of the first forward onwards, all retrieval requests of the same target are remembered in a request pool. If we do not receive the data within that window we move on to the next peer. If we receive no delivery within the lifecycle of the request (it is kept alive by the live timeouts of the incoming requests for the content), we consider the item nonexistent and may even keep a record of that.

After successful retrieval, the preimage is stored and the requests are answered by returning the preimage object to all requesting nodes that are active (in terms of being alive connected as well as interested based on their timeout) either they relayed or originated the request. In fact these two are not nececssarily distinguished, which allows quasi anonymous browsing.

The pool of requesting nodes then can be forgotten, since all further queries can be responded with chunk delivery.

Deliveries that are unexpected can be considered storage requests.

If a storage request appears for the first time we assess the key for proximity and if deemed too distant may be forgotten. If we want to keep it (which is probably 100%, we just do not forward) then we save it to persistent storage. If the key is found in the database, its expiry may be updated. Storage requests are forwarded to the peers in the same kademlia proximity bin. If we are sufficiently close, the bin might include peers more distant from the chunk than we are.

Syncing

Node syncronisation is the protocol that makes sure content ends up where it is queried. Since the swarm has a 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 capacity 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 your worth while storing. 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 while 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 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. In order to efficiently utilise such access 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.

Now with popularity it may well happen that a node closest to the target address is no longer motivated to keep a chunk. If all the neighbouring nodes have the content, the retrieval may never end up with the closest node and if they themselves happen not to ever retrieve that content, the chunk is deleted. This resembles a doughnot with a hole in the middle. Just as the doughnut grows if more mouths bite at it, need to make sure that it never breaks, no queries from outside end up with the closest nodes which do not have it. Elastic shrinking requires that when a node decides to delete a content it needs to forward it to all peers closer to the chunk than itself. This is in fact anindication to the receiving peer that subsequent queries may end up being routed to them so they will be rewarded for their delivery.

Smart syncronisation 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 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 syncronise with a peer.

When two peers connect, they establish their synchronisation state by exchanging information in the protocol handshake. When a connection is peer connection is opened the first time, syncronisation does not specify an index count, meaning that all content in the relevant address space no matter how long ago it was entered is offered to the peer. The address space relevant by default just designates all addresses that are closer to the receiving node than the source. Syncronisation goes both ways independently. Once all content up to the current index is syncronised, the receiving peer updates the syncornisation state with the current index given by the source node. The source providing a counter should mean that they have provided the recipient with all chunks they have upto that time.

All newly stored content during a live connection is also offered to the peer. Assuming enough bandwidth, peers are expected to be fully in sync meaning that the storage counter stored by the recipient about a source is not very far behind the source node's current storage count.

In practice all replication of content since the beginning of the peer session is persisted across sessions. This is needed anyway since propagation can overload the connection causing network buffer contention. For a dynamic response, the stream of outgoing store requests are buffered. This means that if there is a disconnection, the earlier backlog will be replayed upon reconnection, ie. offered again to the recipient. Therefore for all intents and purposes synronisation of content for the periods of active connection do not need to be requested. If the recipient updates the counter as given by the source then at disconnection, the syncstate containing this counter will be recorded. Next time the peers connect

the recipient receives all content stored between this index and the beginning of the session. Since syncronisation can be adjusted by the recipient, it is assumed that syncing state is persisted by the recipient and given in the protocol handshake.

The handshake also allows the recipient to specify an address range by default covering all addresses not further than the peers' proximity. Note that in the case of peers in the most proximate bin, the target range may contain chunks that are closer to the source than the recipient.

The syncing protocol as defined here subsumes all scenarios where content is pushed. Given all the scenarios a chunk needs to be pushed, we distinguish 5 types:

Delivery is the responses to a retrieve request (either from originator or forwarded, either locally found or delivered to by other peers). Delivery proceeds typically from nodes closer to the target towards nodes farther from the target.

Propagation new content pushed to us as a result of syncronising with other peers. Propagation typically proceeds from nodes farther from the target to nodes closer to the target.

Deletion if content is deleted, content must be pushed inwards, i.e., proceeds from nodes farther from the target to nodes closer to the target.

History Delayed propagation of existing chunks prompted by syncronisation in the narrow sense. proceeds from nodes farther from the target to nodes closer to the target.

backlog is the undelivered chunks buffered at previous sessions

These 5 types are roughly in order of decreasing importance/urgency. The implementation lets you assign independent priorities to these types however we strongly recommend a monotonically decreasing prioritisation. By default, delivery is high priority, propagation types are medium and backlog is low priority. Note that within that priority backlog is replayed respecting the original priorities preserved. Also historical syncing is lower priority than real time traffic so in the default case of propagation, historical syncing only kicks in if no real time high or medium priority chunks available.

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 *unsyncedKeys* message containing a batch of hashes offered to push to the recipient. Recipient responds with a *delivery request* which also contains a batch of hashes that recipient actually needs (does not have) out of the ones listed among the incoming unsynced keys. If no chunks are missing an empty response is possible. Unsynced keys is sent whenever a delivery request is received. If none received until a timeout period and there are outstanding content to push, an unsynced keys message is sent.

Peer management (hive, kademlia)

Hive is the logistic manager of the swarm. It uses a generic kademlia nodetable to find best peer list for any target. This is used by the netstore to search for content in the swarm. When the node receives peer suggestions (bzz protocol peersMsgData exchange), the hive relays the peer addresses obtained from the message to the Kademlia table for db storage and filtering. Hive also manages connections and disconnections that allows for bootstapping as well as keeping the routing table uptodate. When the p2p server connects with a node capable of bzz protocol, the hive registers the node in the kademlia table and sends a *self lookup*. A self lookup is basically just a retrieve request with intended target corresponding to the node's base address. The receiving node does not record self lookups as a request or forward it, just reply with peers. This can be improved by simply automatically sending all relevant peers to a connected peer at the time they become known. All peers sent to the connected node are cached so that no repeat sends happen during the session.

Peer addresses

Nodes in the network are identified by the hash the ethereum address of the swarm base account. The distance between two addresses is the MSB first numerical value of their XOR.

Logarithmic distance and network topology

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 random sample of items in the bins as connections in a network of interconnected nodes than 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 in 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.

Peer table format

The peer table consists of rows, initially only one, at most 255 (typically much less). Each row contains at most k peers (data structures containing information about said peer such as their peer address, network address, a timestamp, etc). The parameter k is called *bucket size* and specified as part of the node configuration.

Row numbering starts with 0. Each row number i contains peers whose address matches the first i bits of this node's address. The $i+1$ bit of the address must differ from this node's address in all rows except the last one.

As a matter of implementation, it might be worth internally representing all 255 rows from the outset (requiring that the $i + 1$ bit be different from our node in all rows); but then considering all of the rows at the end as if they were one row. That is, we look at empty rows at the end and treat the elements in them as if they belonged to row i where i is the lowest index such that the total number of all elements in row i and in all higher rows, together is at most k[1].

A peer is added to the row to which it belongs according to its proximity order (the length of the address prefix in common with the base address). If that would increase the length of the row in question beyond the bucket size, the *worst* peer (according to some, not necessarily global, peer quality metric) is dropped from the row, except if it is the last row.

Joining the network requires only one bootstrap peer, all nodes from its table are included in the node's peer table. Thereafter, it performs a lookup of a synthetic random address from the address range corresponding to rows with indices that are smaller than the row in which the bootstrap node ended up.

Nodes can still safely dump their full peer table and accept connections from naive nodes. Overwriting the entire peer table of a node requires significant computational effort even with relatively low bucket size. DoS attacks against non-naive nodes (as described in this page) require generating addresses with corresponding key pairs for each row, requiring quite a bit of hashing power.

[1] There is a difference here to the original Kademlia paper http://pdos.csail.mit.edu/~petar/papers/maymounkov-kademlia-lncs.pdf. The rows with a high i for us here are the rows with a low i in the paper. For us, high i means high number of bits agreeing, for them high i mean high xor distance.

Peer table update

The overlay topology (kademlia) is supposed to be able to return one node record with the highest priority for desired connection. This is used to pick candidates for live nodes that are most wanted for a higly connected low centrality network structure for Swarm which best suits for a Kademlia-style routing.

The candidate is chosen using the following strategy. We check for missing online nodes in the buckets for 1 upto Max BucketSize rounds. On each round we proceed from the low to high proximity order buckets. If the number of active nodes (=connected peers) is less than the current round, then start looking for a known candidate. To determine if there is a candidate to recommend the node record database row corresponding to the bucket is checked. If the row cursor is on position i, the ith element in the row is chosen. If the record is scheduled not to be retried before NOW, the next element is taken. If the record is can be retried, it is set as checked, scheduled for checking and is returned. The time of the next check is NOW + X (duration) such that X = ConnRetryExp * delta where delta is the time past since the last check and ConnRetryExp is constant obsoletion factor. (Note that when node records are added from peer messages, they are marked as checked and placed at the cursor, ie. given priority over older entries). Entries which were checked more than purgeInterval ago are deleted from the node db row. If no candidate is found after a full round of checking the next bucket up is considered. If no candidate is found when we reach the maximum-proximity bucket, the next round starts.

node record a is more favoured to b $a > b$ iff a is a passive node (record of offline past peer)

$$|proxBin(a)| < |proxBin(b)|$$

$$||(proxBin(a) < proxBin(b)$$
$$|proxBin(a)| == |proxBin(b)|)$$

$$||(proxBin(a) == proxBin(b)$$
$$lastChecked(a) < lastChecked(b))$$

This has double role. Starting as naive node with empty db, this implements Kademlia bootstrapping and as a mature node, it fills short lines. All on demand.

The bzz protocol

BZZ implements the bzz subprotocol, the wire protocol of swarm. The bzz protocol is implemented as a subprotocol of the ethereum devp2p system. The protocol instance is launched on each peer by the network layer if the BZZ protocol handler is registered on the p2p server.

The protocol takes care of actually communicating the bzz protocol encoding and decoding requests for storage and retrieval, handling the protocol handshake dispaTching to netstore for handling the DHT logic, registering peers in the Kademlia table via the hive logistic manager.

Note: the bzz protocol is in a flux, as the components on the roadmap get implemented and the protocol solifies, a detailed wire protocol spec will be provided

---

Incentivisation

swap, swear & swindle

SWAP – Swarm Accounting Protocol

Swarm Accounting Protocol, Secured With Automated Payments

SWEAR – Storage With Enforced Archiving Rules or Swarm Enforcement And Registration

SWINDLE – Secured With INsurance Deposit Litigation and Escrow

Chapter 7

Indices and tables

- genindex
- modindex
- search

Index

A

API, 40
autocash, 26
autocash interval, 26
autocash target credit buffer, 26
autocash threshold, 26
AutoCashBuffer, 26
AutoCashInterval, 26
AutoCashThreshold, 26
autodeploy (chequebook contract), 24
autodeposit, 25
AutoDepositBuffer, 26
AutoDepositThreshold: autodeposit threshold, 26

B

Beneficiary, 24
benefieciary (‘‘Beneficiary‘‘ configuration parameter), 24
bootstrapping
    network, 46
bucket size (‘‘BucketSize‘‘), 46
BuyAt, 24
bzzhash, 23, 41
BzzKey, 43

C

cheque, 26
chequebook, 24
chequebook contract address, 24
chunk, 40
chunk size, 42
chunker, 23, 42
Contract, 24
credit buffer, 26

D

delivery request message, 23
delivery types, 23
DHT, 43
distance measure, 43
DPA, 42

E

expiry, 43

H

hash, 41
hive, 45
HTTP proxy, 40

J

joining, 42

K

Kademlia, 23

M

manifest, 40
maximum accepted chunk price, 24
maximum accepted chunk price (‘‘BuyAt‘‘), 24
merkle tree, 42
message, 40

N

network
    bootstrapping, 46

O

offered chunk price, 24
offered chunk price (‘‘BuyAt‘‘), 24

R

recipient address for service payments, 24

S

self lookup, 45
SellAt, 24
smart sync, 23
splitting, 42
storage layer, 40
storage radius, 43
SyncModes, 45
SyncPriorities, 45
syncronisation, 23

U

unsynced keys message, 23