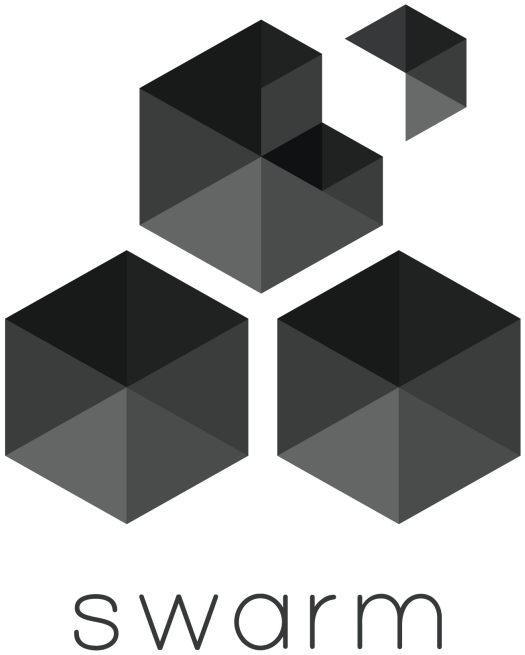


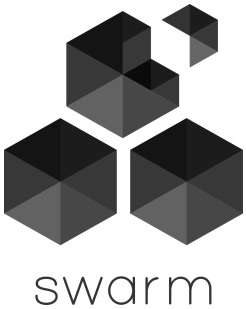
swarm documentation

Release 0.3

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

Dec 03, 2018

Contents



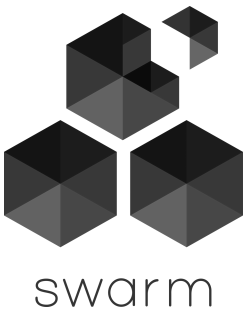
Swarm is a distributed storage platform and content distribution network, a native base layer service of the Ethereum **web3** stack.

This documentation was created with curious end-users, third web enterpreneurs and developers in mind. It should make you thoroughly educated about Swarm within a reasonable amount of time.

Table of contents:

Chapter 1

Introduction



Swarm is a distributed storage platform and content distribution service, a native base layer service of the ethereum *web3* 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 blockchain data. From an economic point of view, it allows participants to efficiently pool their storage and bandwidth resources in order to provide these services to all participants of the network, all while being incentivised by Ethereum.

1.1 Objective

Swarm's broader objective is to provide infrastructure services for developers of decentralised web applications (dapps), notably: messaging, data streaming, peer to peer accounting, mutable resource updates, storage insurance, proof of custody scan and repair, payment channels and database services.

From the end user's perspective, Swarm is not that different from the world wide web, with the exception that uploads are not hosted on a specific server. Swarm offers a peer-to-peer storage and serving solution that is DDoS-resistant, has 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 (using ENS), service payments and content availability insurance.

Please refer to our development roadmap to stay informed with our progress.

1.2 Overview

Swarm is set out to provide base layer infrastructure for a new decentralised internet. Swarm is a peer-to-peer network of nodes providing distributed digital services by contributing resources (storage, message forwarding, payment processing) to each other. These contributions are accurately accounted for

on a peer to peer basis, allowing nodes to trade resource for resource, but offering monetary compensation to nodes consuming less than they serve.

The Ethereum Foundation operates a Swarm testnet that can be used to test out functionality in a similar manner to the Ethereum testnet (ropsten). Everyone can join the network by running the Swarm client node on their server, desktop, laptop or mobile device. See *Getting started* for how to do this. The 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 0.3 is running on all nodes.

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

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

Swarm nodes can also connect with one (or several) ethereum blockchains for domain name resolution and one ethereum blockchain for bandwidth and storage compensation. Nodes running the same network id are supposed to connect to the same blockchain for payments. A Swarm network is identified by its network id which is an arbitrary integer.

Swarm 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 on the testnet until storage insurance is implemented (see Roadmap for more details). All participating nodes should consider participation a voluntary service with no formal obligation whatsoever and should be expected to delete content at their will. Therefore, users should under no circumstances regard Swarm as safe storage until the incentive system is functional.

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

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

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

1.3 Available APIs

Swarm offers several APIs:

- CLI
- JSON-RPC - using web3 bindings over Geth's IPC
- HTTP interface - every Swarm node exposes a local HTTP proxy that implements the bzz protocol suite
- Javascript - available through the erebos, swarm-js or swarmgw packages

1.4 Code

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

1.5 Roadmap

Roadmap is located at https://github.com/ethersphere/Swarm/wiki/roadmap

Note: Swarm is experimental code and untested in the wild. Use with extreme care. We encourage developers to connect to the testnet with their permanent nodes and give us feedback.

1.6 Public gateways

Swarm offers a local HTTP proxy API that Dapps can use to interact with Swarm. The Ethereum Foundation is hosting a public gateway, which allows free access so that people can try Swarm without running their own node.

The Swarm public gateway can be found at https://swarm-gateways.net and is always running the latest *stable* Swarm release.

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

1.7 Example Dapps

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

1.8 Reporting a bug and contributing

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

- https://github.com/ethersphere/go-ethereum/issues
- Good first issues

Please include the commit and branch when reporting an issue.

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

Prospective contributors please read the *Developers' Guide <https://github.com/ethereum/go-ethereum/wiki/Developers'-Guide>*

1.9 Credits

Swarm is funded by the Ethereum Foundation and industry sponsors.

1.9.1 The Core team

- Viktor Trón - @zelig
- Daniel A. Nagy - @nagydani
- Aron Fischer - @homotopycolimit
- Louis Holbrook - @nolash
- Lewis Marshal - @lmars
- Fabio Barone - @holisticode
- Anton Evangelatov - @nonsense
- Janoš Guljaš - @janos
- Balint Gabor - @gbalint
- Elad Nachmias - @justelad

were on the core team:

- Zahoor Mohamed - @jmozah
- Zsolt Felföldi - @zsfelfoldi
- Nick Johnson - @Arachnid

1.9.2 Sponsors and collaborators

- http://status.im
- http://livepeer.org
- http://jaak.io
- http://datafund.io
- http://mainframe.com
- http://wolk.com
- http://riat.at
- http://datafund.org
- http://216.com
- http://cofound.it
- http://iconomi.net
- http://infura.io
- http://epiclabs.io
- http://asseth.fr

1.9.3 Special thanks

- Felix Lange, Alex Leverington for inventing and implementing devp2p/rlpx
- Jeffrey Wilcke, Peter Szilagyi and the entire ethereum foundation go team for continued support, testing and direction
- Gavin Wood and Vitalik Buterin for the holy trinity vision of web3
- Nick Johnson for ENS and ENS Swarm integration
- Alex Van der Sande, Fabian Vogelsteller, Bas van Kervel, Victor Maia, Everton Fraga and the Mist team
- Elad Verbin for his continued technical involvement as an advisor and ideator
- Nick Savers for his unrelenting support and meticulous reviews of our papers
- Gregor Zavcer, Alexei Akhunov, Alex Beregszaszi, Daniel Varga, Julien Boutloup for inspiring discussions and ideas
- Juan Benet and the IPFS team for continued inspiration
- Carl Youngblood, Shane Howley, Paul Le Cam, Doug Leonard and the mainframe team for their contribution to PSS and MRU
- Sourabh Niyogi and the entire Wolk team for the inspiring collaboration on databases
- Ralph Pilcher for implementing the swap swear and swindle contract suite in solidity/truffle and Oren Sokolowsky for the initial version
- Javier Peletier from Epiclabs (ethergit) for his contribution to MRUs
- Jarrad Hope and Carl Bennet (Status) for their support
- Participants of the orange lounge research group and the Swarm orange summits
- Roman Mandeleil and Anton Nashatyrev for an early java implementation of swarm
- Igor Sharudin, Dean Vaessen for example dapps
- Community contributors for feedback and testing
- Daniel Kalman, Benjamin Kampmann, Daniel Lengyel, Anand Jaisingh for contributing to the swarm websites
- Felipe Santana, Paolo Perez and Paratii team for filming at the 2017 swarm summit and making the summit website

Chapter 2

Installation and Updates

Swarm is part of the Ethereum stack, the reference implementation is currently at POC3 (proof of concept 3), or version 0.3.x

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

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

2.1 Installing Swarm on Ubuntu via PPA

The simplest way to install Swarm on Ubuntu distributions is via the built in launchpad PPAs (Personal Package Archives). We provide a single PPA repository that contains our stable releases for Ubuntu versions trusty, xenial, bionic and cosmic.

To enable our launchpad repository please run:

```
sudo apt-get install software-properties-common
sudo add-apt-repository -y ppa:ethereum/ethereum
```

After that you can install the stable version of Swarm:

```
sudo apt-get update
sudo apt-get install ethereum-swarm
```

2.2 Installing Swarm from source

The Swarm source code for can be found on https://github.com/ethereum/go-ethereum

2.2.1 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.

Ubuntu / Debian

```
sudo apt install git
sudo apt install golang
```

Archlinux

```
pacman -S git go
```

Generic Linux

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

To install it, download the tar.gz file for your architecture

```
curl -O https://dl.google.com/go/go1.10.1.linux-amd64.tar.gz
```

Unpack it to the /usr/local

```
sudo tar -C /usr/local -xzf go1.10.1.linux-amd64.tar.gz
```

macOS

```
brew install go git
```

2.2.2 Configuration

You should then prepare your Go environment, for example:

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

2.2.3 Compiling and installing

Once all prerequisites are met, download and install packages and dependencies for go-ethereum.

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

This will download the master source code branch.

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

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

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

```
swarm version
```

or, if your *PATH* is not set and the *swarm* command can not be found, try:

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

This should return some relevant information. For example:

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

2.2.4 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 ./cmd/geth
go install ./cmd/swarm
```

Chapter 3

Getting started

The first thing to do is to start up your Swarm node and connect it to the Swarm.

3.1 Running Swarm

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

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

```
geth account new
```

You will be prompted for a password:

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

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

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

Using this account, connect to Swarm with

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

(You should replace 2f1cd699b0bf461dcfbf0098ad8f5587b038f0f1 with your address).

3.1.1 Verifying that your local Swarm node is running

When running, `swarm` is accessible through an HTTP API on port 8500. Confirm that it is up and running by pointing your browser to http://localhost:8500

3.2 How do I enable ENS name resolution?

Note: **ENS** is based on a suite of smart contracts running on the *Ethereum mainnet*.

The Ethereum Name Service is the Ethereum equivalent of DNS in the classic web. In order to use **ENS** to resolve names to swarm content hashes, `swarm` has to connect to a `geth` instance that is connected to the *Ethereum mainnet*. This is done using the `--ens-api` flag.

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

```
geth
```

for a full geth node, or

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

for light client mode.

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

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

```
swarm --ens-api '$HOME/.ethereum/geth.ipc' \
  --bzzaccount 2f1cd699b0bf461dcfbf0098ad8f5587b038f0f1
```

Note: For Mac OS, replace "$HOME/.ethereum/" with "~/Library/Ethereum/"

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

3.2.1 Using Swarm together with the testnet ENS

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

Run a geth node connected to the Ropsten testnet

```
geth --testnet
```

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

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

Swarm will automatically use the ENS deployed on Ropsten.

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

```
swarm --ens-api eth:314159265dD8dbb310642f98f50C066173C1259b@/home/user/.ethereum/
↪geth.ipc \
        --ens-api test:0x112234455C3a32FD11230C42E7Bccd4A84e02010@ws:1.2.3.4:5678␣
↪\
        --ens-api 0x230C42E7Bccd4A84e02010112234455C3a32FD11@ws:8.9.0.1:2345
```

Will use the `geth.ipc` to resolve `.eth` names using the contract at `314159265dD8dbb310642f98f50C066173C1259b` and it will use `ws:1.2.3.4:5678` to resolve `.test` names using the contract at `0x112234455C3a32FD11230C42E7Bccd4A84e02010`. For

all other names it will use the ENS contract at `0x230C42E7Bccd4A84e02010112234455C3a32FD11` on `ws:8.9.0.1:2345`.

3.2.2 Using an external ENS source

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

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

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

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

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

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

3.3 Alternative modes

Below are examples on ways to run swarm beyond just the default network.

3.3.1 Swarm in singleton mode (no peers)

To launch in singleton mode, use the `--maxpeers 0` flag.

```
swarm --bzzaccount $BZZKEY \
      --datadir $DATADIR \
      --ens-api $DATADIR/geth.ipc \
      --maxpeers 0
```

3.3.2 Adding enodes manually

By default, swarm will automatically seek out peers in the network. This can be suppressed using the `--nodiscover` flag:

```
swarm --bzzaccount $BZZKEY \
      --datadir $DATADIR \
      --ens-api $DATADIR/geth.ipc \
      --nodiscover
```

Without discovery, it is possible to manually start off the connection process by adding one or more peers using the `admin.addPeer` console command.

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

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

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

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

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

```
geth --exec "admin.nodeInfo.enode" attach ipc:/path/to/bzzd.ipc
```

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

3.3.3 Connecting to the public Swarm cluster

If you would like to join the public Swarm cluster operated by the Ethereum Foundation and other contributors, you can use one of the bootnodes avaible from this list:

https://gist.github.com/homotopycolimit/db446fa3269a199762e67b2ca037dbeb

The cluster functions as a free-to-use public access gateway to Swarm, without the need to run a local node. To download data through the gateway use the `https://swarm-gateways.net/bzz:/<address>/` URL.

Chapter 4

Uploading and downloading

Contents

4.1 Introduction

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

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

of your local Swarm node querying its peers in the network for the relevant chunks of data and then reassembling the content locally.

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

4.2 Using CLI

4.2.1 Uploading a file to your local Swarm node

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

The basic command for uploading to your local node is `swarm up FILE`. For example, issue the following command to upload the file example.md file to your local Swarm node

```
swarm up /path/to/example.md
> d1f25a870a7bb7e5d526a7623338e4e9b8399e76df8b634020d11d969594f24a
```

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

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

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

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

Suppressing automatic manifest creation

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

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

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

4.2.2 Downloading a single file

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

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

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

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

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

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

4.2.3 Uploading to a remote Swarm node

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

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

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

4.2.4 Uploading a directory

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

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

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

Directory with default entry

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

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

You can now access index.html at

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

and also at

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

This is especially useful when the hash (in this case `ef6fc0747d1fbaf86d769b3eed1c836733a30ab90f84c91291`5 is given a registered name like `mysite.eth` in the Ethereum Name Service. In this case the lookup would be even simpler:

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

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

4.2.5 Downloading a directory

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

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

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

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

4.2.6 Adding entries to a manifest

The command for modifying manifests is `swarm manifest`.

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

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

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

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

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

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

4.3 Using HTTP

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

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

To upload a single file, run this:

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

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

```
027e57bcbae76c4b6a1c5ce589be41232498f1af86e1b1a2fc2bdffd740e9b39
```

This is the address string of your content inside Swarm. It is the same hash that would have been returned by using the *swarm up* command

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

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

The result should be your file:

```
some-data
```

And that's it.

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

4.3.1 Tar stream upload

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

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

# create a tar archive containing the two directories
tar cf files.tar .

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

You can then download the files using:

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

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

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

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

4.3.2 Multipart form upload

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

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

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

4.3.3 Add files to an existing manifest using multipart form

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

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

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

4.3.4 Upload files using a simple HTML form

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

4.3.5 Listing files

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

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

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

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

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

Chapter 5

Working with content

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

5.1 Using ENS names

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

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

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

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

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

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

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

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

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

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

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

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

In short, the steps you must take are:

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

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

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

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

---

5.1.1 Overview of ENS (video)

Nick Johnson on the Ethereum Name System

5.2 Feeds

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

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

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

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

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

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

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

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

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

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

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

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

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

5.2.1 Feed Manifests

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

5.2.2 Feeds API

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

HTTP API

Posting to a Feed

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

1. Get Feed template

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

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

Where:

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

Note:

- If `topic` is omitted, it is assumed to be zero, 0x000...

- if `name=<name>` (optional) is provided, a subtopic is composed with that name
- A common use is to omit topic and just use `name`, allowing for human-readable topics

---

You will receive a JSON like the below:

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

2. Post the update

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

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

Where:

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

Reading a Feed

To retrieve a Feed's last update:

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

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

Note:

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

To get a previous update:

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

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

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

Creating a Feed Manifest

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

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

This will create a manifest referencing the provided Feed.

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

Go API

Query object

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

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

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

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

Request object

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

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

Posting to a Feed

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

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

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

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

Reading a Feed

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

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

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

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

Creating a Feed Manifest

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

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

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

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

Example Go code

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

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

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

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

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

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

Command-Line

Posting to a Feed

To update a Feed with the cli:

```
swarm feed update [command options] <0x Hex data>

creates a new update on the specified topic
```

(continues on next page)

(continued from previous page)

```
           The topic can be specified directly with the --topic flag as an hex␣
↪string
           If no topic is specified, the default topic (zero) will be used
           The --name flag can be used to specify subtopics with a specific name.
           If you have a manifest, you can specify it with --manifest instead of --
↪topic / --name
           to refer to the feed
  OPTIONS:
 --manifest value  Refers to the feed through a manifest
 --name value      User-defined name for the new feed, limited to 32 characters.␣
↪If combined with topic, the feed will be a       subtopic with this name
 --topic value     User-defined topic this feed is tracking, hex encoded. Limited␣
↪to 64 hexadecimal characters
```

Reading Feed status

```
swarm feed info [command options] [arguments...]

obtains information about an existing Swarm feed
          The topic can be specified directly with the --topic flag as an hex␣
↪string
          If no topic is specified, the default topic (zero) will be used
          The --name flag can be used to specify subtopics with a specific name.
          The --user flag allows to refer to a user other than yourself. If not␣
↪specified,
          it will then default to your local account (--bzzaccount)
          If you have a manifest, you can specify it with --manifest instead of --
↪topic / --name / ---user
          to refer to the feed

OPTIONS:
--manifest value  Refers to the feed through a manifest
--name value      User-defined name for the new feed, limited to 32 characters. If␣
↪combined with topic, it will refer to a subtopic with this name
--topic value     User-defined topic this feed is tracking, hex encoded. Limited␣
↪to 64 hexadecimal characters
--user value      Indicates the user who updates the feed
```

Creating a Feed Manifest

The Swarm CLI allows to create Feed Manifests directly from the console:

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

```
swarm feed create [command options]

creates and publishes a new feed manifest pointing to a specified user's updates␣
↪about a particular topic.
          The feed topic can be built in the following ways:
          * use --topic to set the topic to an arbitrary binary hex string.
          * use --name to set the topic to a human-readable name.
              For example --name could be set to "profile-picture", meaning this␣
↪feed allows to get this user's current profile picture.
          * use both --topic and --name to create named subtopics.
            For example, --topic could be set to an Ethereum contract address and -
↪-name could be set to "comments", meaning
          this feed tracks a discussion about that contract.
```

(continues on next page)

(continued from previous page)

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

OPTIONS:
--name value   User-defined name for the new feed, limited to 32 characters. If␣
↪combined with topic, it will refer to a subtopic with this name
--topic value  User-defined topic this feed is tracking, hex encoded. Limited to␣
↪64 hexadecimal characters
--user value   Indicates the user who updates the feed
```

5.2.3 Computing Feed Signatures

1. computing the digest:

The digest is computed concatenating the following:

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

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

JavaScript example

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

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

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




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

(continues on next page)

(continued from previous page)

```
    protocolVersion = request.protocolVersion

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

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

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

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

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

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

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

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

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

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

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

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

// obtain digest
```

(continues on next page)

(continued from previous page)

```
digest = feedUpdateDigest(request, data)

console.log(digest)
```

5.3 Manifests

In general manifests declare a list of strings associated with swarm hashes. A manifest matches to exactly one hash, and it consists of a list of entries declaring the content which can be retrieved through that hash. Let us begin with an introductory example.

This is demonstrated by the following example. Let's create a directory containing the two orange papers and an html index file listing the two pdf documents.

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

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

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

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

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

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

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

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

> {
  "entries": [
    {
      "hash": "4b3a73e43ae5481960a5296a08aaae9cf466c9d5427e1eaa3b15f600373a048d",
      "contentType": "text/html; charset=utf-8"
    },
    {
      "hash": "4b3a73e43ae5481960a5296a08aaae9cf466c9d5427e1eaa3b15f600373a048d",
      "contentType": "text/html; charset=utf-8",
      "path": "index.html"
    },
    {
      "hash": "69b0a42a93825ac0407a8b0f47ccdd7655c569e80e92f3e9c63c28645df3e039",
      "contentType": "application/pdf",
      "path": "smash.pdf"
    },
    {
      "hash": "6a18222637cafb4ce692fa11df886a03e6d5e63432c53cbf7846970aa3e6fdf5",
      "contentType": "application/pdf",
      "path": "sw^3.pdf"
    }
  ]
}
```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

{"entries":[{"hash":
↪"25e7859eeb7366849f3a57bb100ff9b3582caa2021f0f55fb8fce9533b6aa810","path":
↪"avicon.ico","mode":493,"size":32038,"mod_time":"2018-06-12T15:36:29Z"},
            {"hash":
↪"97cfd23f9e36ca07b02e92dc70de379a49be654c7ed20b3b6b793516c62a1a03","path":"onts/
↪glyphicons-halflings-regular.","contentType":"application/bzz-manifest+json",
↪"mod_time":"0001-01-01T00:00:00Z"}
 ]}
```

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

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

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

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

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

5.4 Encryption

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

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

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

When your data is retrieved it will only get decrypted on your local Swarm node. During the whole retrieval process the chunks traverse the network in their encrypted form, and none of the participating

peers are able to decrypt them. They are only decrypted and assembled on the Swarm node you use for the download.

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

Note: Swarm currently supports both encrypted and unencrypted `swarm up` commands through usage of the `--encrypt` flag. This might change in the future as we will refine and make Swarm a safer network.

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

Example usage:

```
swarm up foo.txt
> 4b964a75ab19db960c274058695ca4ae21b8e19f03ddf1be482ba3ad3c5b9f9b
# note the short reference of the unencrypted upload
swarm up --encrypt foo.txt
>␣
↪c2ebba57da7d97bc4725a542ff3f0bd37163fd564e0298dd87f320368ae4faddd1f25a870a7bb7e5d526a7623338e4e
# note the longer reference of the encrypted upload
swarm up --encrypt foo.txt
>
e76efd76ef1161e4903acc43b5dc634c02fbba7e5f242c32726e78d4e71ffa9cf5a6ca8a19cbada15f38cac79557a9300
# note the different reference on the second upload (because of the random␣
↪encryption key)
```

5.5 Access Control

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

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

5.5.1 Password protection

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

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

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

5.5.2 Selective access using EC keys

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

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

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

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

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

5.5.3 Usage

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

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

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

5.5.4 CLI usage

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

Protecting content with a password:

Note: The `--password` flag when using the `pass` strategy refers to the password that protects the access-controlled content. This file should contain the password in plaintext. The command expects you to input the uploaded swarm content hash you'd like to limit access to (`REF`)

```
$ echo 'mysupersecretpassword' > /path/to/password/file
$ swarm access new pass --password /path/to/password/file <REF>
4b964a75ab19db960c274058695ca4ae21b8e19f03ddf1be482ba3ad3c5b9f9b
```

The returned hash `4b964a75ab19db960c274058695ca4ae21b8e19f03ddf1be482ba3ad3c5b9f9b` is the hash of the access controlled manifest. When requesting this hash through the HTTP gateway you should receive an `HTTP Unauthorized 401` error:

```
$ curl http://localhost:8500/bzz:/
↪4b964a75ab19db960c274058695ca4ae21b8e19f03ddf1be482ba3ad3c5b9f9b
```

The same request should make an authentication dialog pop-up in the browser. You could then input the password needed and the content should correctly appear.

Requesting the same hash with HTTP basic authentication (password only) would return the content too:

```
$ curl http://:mysupersecretpassword@localhost:8500/bzz:/
↪4b964a75ab19db960c274058695ca4ae21b8e19f03ddf1be482ba3ad3c5b9f9b
```

Protecting content with Elliptic curve keys (single grantee):

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

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

```
$ swarm --bzzaccount 2f1cd699b0bf461dcfbf0098ad8f5587b038f0f1 access new pk --
↪grant-key 02e6f8d5e28faaa899744972bb847b6eb805a160494690c9ee7197ae9f619181db
↪<REF>
4b964a75ab19db960c274058695ca4ae21b8e19f03ddf1be482ba3ad3c5b9f9b
```

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

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

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

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

---

Note: the `act` strategy expects a grantee public-key list and/or a list of permitted passwords to be communicated to the CLI. This is done using the `--grant-keys` flag and/or the `--password` flag. Grantee public keys are expected to be in an *secp256 compressed* form - 66 characters long string (e.g. `02e6f8d5e28faaa899744972bb847b6eb805a160494690c9ee7197ae9f619181db`). Each

grantee should appear in a separate line. Passwords are also expected to be line-separated. Comments and other characters are not allowed.

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

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

The access controlled content could be accessed in one of the following ways:

1. Request the hash through one of the nodes that were granted access and/or posses the granted private key (and that the requesting node has been started with the appropriate `bzzaccount` that is associated with the relevant key) - either the local node that was used to upload the content or one of the nodes which were granted access through their public keys
2. Request the hash with HTTP authentication using one of the granted passwords

5.5.5 HTTP usage

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

When accessing password protected content:

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

The credentials for accessing content protected by a password can be provided in the initial request in the form of: `http://:<password>@localhost:8500/bzz:/<hash or ens name>`

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

When accessing EC key protected content:

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

5.6 FUSE

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

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

5.6.1 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
```

5.6.2 CLI Usage

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

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

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

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

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

For example:

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

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

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

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

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

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

mounting the previous version). While you change the current mount, this happens under the hood and your mount remains up-to-date.

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

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

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

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

```
swarm fs list
```

Example Output:

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

5.7 BZZ URL schemes

Swarm offers 6 distinct URL schemes:

5.7.1 bzz

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

Example:

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

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

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

Swarm is awesome%
```

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

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

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

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

5.7.2 bzz-raw

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

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

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

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

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

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

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

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

5.7.3 bzz-list

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

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

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

5.7.4 bzz-hash

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

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

Response content type is *text/plain*.

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

5.7.5 bzz-immutable

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

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

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

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

(continues on next page)

(continued from previous page)

```
    "Details": ""
}
```

5.7.6 bzz-resource

`bzz-resource` allows you to receive hash pointers to content that the ENS entry resolved to at different versions

bzz-resource://<id> - get latest update bzz-resource://<id>/<n> - get latest update on period n bzz-resource://<id>/<n>/<m> - get update version m of period n <id> = ens name

Chapter 6

PSS

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

Note: `pss` is still an experimental feature and under active development and is available as of POC3 of Swarm. Expect things to change.

6.1 Basics

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

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

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

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

6.1.1 Privacy features

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

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

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

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

6.2 Usage

See the API Reference for details.

6.2.1 Registering a recipient

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

1. `Encryption key` - can be a ECDSA public key for asymmetric encryption or a 32 byte symmetric key.
2. `Topic` - an arbitrary 4 byte word
3. `Address`- destination (fully or partially specified Swarm overlay address) to use for deterministic routing.

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

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

6.2.2 Sending a message

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

1. `Encryption key id` - id of the stored recipient's encryption key.
2. `Topic` - an arbitrary 4 byte word
3. `Message payload` - the message data as an arbitrary byte sequence.

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

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

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

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

6.2.3 Sending a raw message

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

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

6.2.4 Receiving messages

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

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

6.3 Advanced features

Note: This functionalities are optional features in pss. They are compiled in by default, but can be omitted by providing the appropriate build tags.

6.3.1 Handshakes

`pss` provides a convenience implementation of Diffie-Hellman handshakes using ephemeral symmetric keys. Peers keep separate sets of keys for a limited amount of incoming and outgoing communications, and create and exchange new keys when the keys expire.

6.3.2 Protocols

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

Chapter 7

API reference

7.1 HTTP

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

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

7.2 JavaScript

Swarm currently supports a Javascript API through a few packages:

7.2.1 erebos

erebos is available through NPM by issuing the following command:

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

Note: Full documentation is available on the documentation website.

7.2.2 swarm-js

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

```
npm install swarm-js
```

Note: Full documentation is available on the GitHub page.

7.2.3 swarmgw

swarmgw is available through NPM by issuing the following command:

```
npm install swarmgw
```

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

```
npm install -g swarmgw
```

Note: Full documentation is available on the GitHub page.

7.3 RPC

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

7.3.1 FUSE

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

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

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

7.3.2 PSS

`pss` methods are by default exposed via IPC. If websockets are activated on the node, they will also be available there.

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

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

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

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

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

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

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

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

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

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

Note: The following methods are used to control the optional pss handshake module. This is an advanced feature, and not required for sending and receiving messages using pss.

---

pss.addHandshake(topic) Activate handshake functionality on the specified topic. The method has no return value.

pss.removeHandshake(topic) Remove handshake functionality on the specified topic. The method has no return value.

pss.handshake(publickey, topic, bool block, bool flush) Instantiate handshake with peer, refreshing symmetric encryption keys. If parameter 3 is false the handshake will happen asynchronously. If parameter 4 is true, it will force expiry of all existing keys. The method returns a list of symmetric key ids created by the handshake. If the handshake is asynchronous, however, returned array will be empty.

pss.getHandshakeKeys(publickey, topic, bool incoming, bool outgoing) Returns the set of valid symmetric encryption keys for a specified peer and topic. If the incoming and outgoing parameters are set, the keys valid for the respective communcations directions are included.

pss.getHandshakeKeyCapacity(symkeyid) Returns the number of messages (uint16) a symmetric handshake key is valid for.

pss.getHandshakePublicKey(symkeyid) Returns the public key associated with the specified symmetric handshake key.

pss.releaseHandshakeKey(publickey, topic, symkeyid, bool instant) Invalidate the specified symmetric handshake key. Normally, the key will be kept for a grace period to allow decryption of messages not yet received at the time of release. If the instant parameter is set, this grace period is omitted, and the key removed instantaneously. This method has no return value.

Chapter 8

Configuration

8.1 Command line options for swarm

The `swarm` executable supports the following configuration options:

- Configuration file
- Environment variables
- Command line

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

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

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

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

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

8.2 Config File

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

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

To run Swarm with a config file, use:

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

8.3 General configuration parameters

Config file	Command-line flag	Environment variable	Default value	Description
n/a	–config value	n/a	n/a	Path to config file in TOML format
n/a	–bzzapi value	n/a	http://127.0.0.1:8500	Swarm HTTP endpoint
BootNodes	–bootnodes value	SWARM_BOOTNODES		Boot nodes
BzzAccount	–bzzaccount value	SWARM_ACCOUNT		Swarm account key
BzzKey	n/a	n/a	n/a	Swarm node base address ($hash(PublicKey)hash(PublicKey)$). This is used to decide storage based on radius and routing by kademlia.
Contract	–chequebook value	SWARM_CHEQUEBOOK_ADDR		Swap chequebook contract address
Cors	–corsdomain value	SWARM_CORS		Domain on which to send Access-Control-Allow-Origin header (multiple domains can be supplied separated by a ‘,’)
n/a	–debug value	n/a	n/a	Prepends log messages with call-site location (file and line number)
n/a	–defaultpath value	n/a		path to file served for empty url path (none)
n/a	–delivery-skip-check value	SWARM_DELIVERY_SKIP_CHECK	false	Skip chunk delivery check (default false)
EnsApi	–ens-api value	SWARM_ENS_API	<$GETH_DATADIR/geth.ipc	Ethereum Name Service API address
EnsRoot	–ens-addr value	SWARM_ENS_ADDR	ens_test_address	Ethereum Name Service contract address
ListenAddr	–httpaddr value	SWARM_LISTEN_ADDR	127.0.0.1	Swarm listen address
n/a	–manifest value	n/a	true	Automatic manifest upload (default true)
n/a	–mime value	n/a	n/a	Force mime type on upload
NetworkId	–bzznetworkid value	SWARM_NETWORK_ID	322	Network ID
Path	–datadir value	GETH_SWARM_DATADIR	<$GETH_DATADIR/swarm	Path to swarm configuration directory
Port	–bzzport value	SWARM_PORT	8500	Port to run the http proxy server
PublicKey	n/a	n/a	n/a	Public key of swarm base account
n/a	–recursive	n/a	false	Upload directories recursively (default false)
n/a	–stdin	n/a	n/a	Reads data to be uploaded from stdin
n/a	–store.path value	SWARM_STORE_PATH	<$GETH_DATADIR/swarm/bzz-<$BZZ_KEY>/chunks	Path to swarm chunk DB

Continued on next page

Table 1 – continued from previous page

Config file	Command-line flag	Environment variable	Default value	Description
n/a	–storage.size value	SWARM_STORE_CAPACITY	5000000	Number of chunks (5M is roughly 20-25GB) (default 5000000)]
n/a	–storage.cache.size value	SWARM_STORE_CACHE_CAPACITY	5000	Number of recent chunks cached in memory (default 5000)
n/a	–sync-update-delay value	SWARM_ENV_SYNC_UPDATE_DELAY	15s	Duration for sync subscriptions update after no new peers are added (default 15s)
SwapApi	–swap-api	SWARM_SWAP_API		URL of the Ethereum API provider to use to settle SWAP payments
SwapEnabled	–swap	SWARM_SWAP_ENABLE	false	Enable SWAP
SyncDisabled	–nosync	SWARM_ENV_SYNC_DISABLE	false	Disable swarm node synchronization
n/a	–verbosity value	n/a	3	Logging verbosity: 0=silent, 1=error, 2=warn, 3=info, 4=debug, 5=detail
n/a	–ws	n/a	false	Enable the WS-RPC server
n/a	–wsaddr value	n/a	localhost	WS-RPC server listening interface
n/a	–wsport value	n/a	8546	WS-RPC server listening port
n/a	–wsapi value	n/a	n/a	API's offered over the WS-RPC interface
n/a	–wsorigins value	n/a	n/a	Origins from which to accept websockets requests
n/a	n/a	SWARM_AUTO_DEFAULTPATH	false	Toggle automatic manifest default path on recursive uploads (looks for index.html)

Chapter 9

Architecture

This chapter is aimed at developers who want to understand the underlying concepts and design of Swarm. In the first part we describe the Swarm network of nodes. In the second part we explain how messaging and data storage can use the Swarm network as a distributed preimage archive.

Contents

9.1 Preface

Swarm defines 3 crucial notions:

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

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

manifest A manifest is a data structure describing file collections; they specify paths and corresponding content hashes allowing for URL based content retrieval. The bzz protocol suite assumes that the content referenced in the domain is a manifest and renders the content entry whose path matches the one in the request path. Manifests can also be mapped to a filesystem directory tree, which allows for uploading and downloading directories. Finally, manifests can also be considered indexes, so it can be used to implement a simple key-value store, or alternatively, a database index. This offers the functionality of *virtual hosting*, storing entire directories, web3 websites or primitive data structures; analogous to web2.0, with centralized hosting taken out of the equation.

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

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

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

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

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

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

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

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

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

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

Nodes cache content that they pass on at retrieval, resulting in an auto scaling elastic cloud: popular (oft-accessed) content is replicated throughout the network decreasing its retrieval latency. Caching also results in a *maximum resource utilisation* in as much as nodes will fill their dedicated storage space with data passing through them. If capacity is reached, least accessed chunks are purged by a garbage collection process. As a consequence, unpopular content will end up getting deleted. Storage insurance (to be implemented in POC4 2019) will offer users a secure guarantee to protect important content from being purged.

9.2 Overlay network

9.2.1 Logarithmic distance

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

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

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

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

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

9.2.2 Kademlia topology

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

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

If each point of a connected subgraph has kademlia connectivity, then we say the subgraph has *kademlia topology*. In a graph with kademlia topology, (1) a path between any two points exists, (2) it can be

found using only local decisions on each hop and (3) is guaranteed to terminate in no more steps than the depth of the destination plus one.

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

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

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

9.2.3 Bootstrapping and discovery

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

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

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

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

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

9.3 Distributed preimage archive

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

A DPA is opinionated about which nodes store what content and this implies a few more restrictions: (1) load balancing of content among nodes is required and is accomplished by splitting content into equal sized chunks (*chunking*); (2) there has to be a process whereby chunks get to where they are supposed to be stored (*syncing*); and (3) since nodes do not have a say in what they store, measures of *plausible deniability* should be employed.

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

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

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

9.3.1 Redundancy

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

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

In POC 4 further measures of redundancy with erasure codes will be implemented. (https://en.wikipedia.org/wiki/Erasure_code), see the orange papers for our specific application)

9.3.2 Caching and purging Storage

Node synchronisation is the protocol that makes sure content ends up where it is queried. Since the Swarm has an address-key based retrieval protocol, content will be twice as likely be requested from a

node that is one bit (one proximity bin) closer to the content's address. What a node stores is determined by the access count of chunks: if we reach the capacity limit for storage the oldest unaccessed chunks are removed. On the one hand, this is backed by an incentive system rewarding serving chunks. This directly translates to a motivation, that a content needs to be served with frequency X in order to make storing it profitable. On the one hand , frequency of access directly translates to storage count. On the other hand, it provides a way to combine proximity and popularity to dictate what is stored.

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

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

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

9.3.3 Synchronisation

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

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

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

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

9.4 Data layer

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

- *message*: p2p RLPx network layer. Messages are relevant for the devp2p wire protocols The bzz protocol suite.
- *chunk*: fixed size data unit of storage in the distributed preimage archive
- *file*: the smallest unit that is associated with a mime-type and not guaranteed to have integrity unless it is complete. This is the smallest unit semantic to the user, basically a file on a filesystem.

- *collection*: a mapping of paths to files is represented by the *swarm manifest*. This layer has a mapping to file system directory tree. Given trivial routing conventions, a url can be mapped to files in a standardised way, allowing manifests to mimic site maps/routing tables. As a result, Swarm is able to act as a webserver, a virtual cloud hosting service.

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

9.4.1 Files

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

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

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

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

9.4.2 Manifests

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

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

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

These HTTP proxy API is described in detail in the *API reference* section.

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

9.5 Components

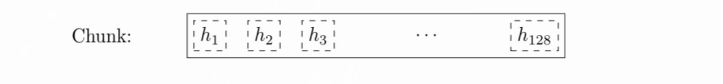
In what follows we describe the components in more detail.

9.5.1 Swarm Hash

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

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



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

Hashes of data chunks are defined as the hashes of the concatenation of the 64-bit length (in LSB-first order) of the content and the content itself. Because of the inclusion of the length, it is resistant to [length extension attacks](http://en.wikipedia.org/wiki/Length_extension_attack), even if the underlying chunk hash function is not. Intermediate chunks are composed of the hashes of the concatenation of the 64-bit length (in LSB-first order) of the content subsumed under this chunk followed by the references to its children (reference is either a chunk hash or chunk hash plus decryption key for encrypted content).

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

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

9.5.2 Chunker

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

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

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

9.5.3 Web3 services

On top of a storage solution outlined so far, Swarm offers various services of a web3 stack needed to build decentralised realtime interactive web applications.

POC3 * pss node-to-node messaging: (basis for higher-level communication platforms like forum, reddit) * mru (mutable resource updates) implement fast, cheap and restrictions

POC4 * Swarm database services * swap, swear and swindle games: payment channel network and standardised service-level agreements for decentralied new

Chapter 10

Resources

10.1 Homepage

the *Swarm homepage* is accessible via Swarm at *theswarm.eth*. The page can be accessed through the public gateway on https://swarm.ethereum.org or https://swarm-gateways.net/bzz:/theswarm.eth/

10.2 Blogposts

- Announcement of POC3
- POC2 public alpha announcement

10.3 Swarm Orange Summit

- Swarm summit 2018 promo video
- 2018 May 7-11 Ljubljana
- 2017 June 4-10 Berlin

10.4 Orange papers

- Viktor Trón, Aron Fischer, Dániel Nagy A and Zsolt Felföldi, Nick Johnson: swap, swear and swindle: incentive system for Swarm. May 2016 - https://swarm-gateways.net/bzz:/theswarm.eth/ethersphere/orange-papers/1/sw^3.pdf
- Viktor Trón, Aron Fischer, Nick Johnson: smash-proof: auditable storage for Swarm secured by masked audit secret hash. May 2016 - https://swarm-gateways.net/bzz:/theswarm.eth/ethersphere/orange-papers/2/smash.pdf
- Viktor Trón, Aron Fischer, Ralph Pilcher, Fabio Barone: swap swear and swindle games: scalable infrastructure for decentralised service economies. Work in progress. June 2018. - https://www.sharelatex.com/read/yszmsdqyqbvc, pfd on swarm.
- Viktor Trón, Aron Fischer, Daniel A. Nagy. Swarm: a decentralised peer-to-peer network for messaging and storage. Work in progress. June, 2018. - https://www.sharelatex.com/read/gxhwssqzgfpr; pdf on swarm.
- P.O.T. data structures and databases on swarm. In preparation.
- Mutable Resource Updates. An off-chain scheme for versioning content in Swarm. In preparation.

- Privacy on swarm. Encryption, access control, private browsing in Swarm. Tentative.
- Analysis of attack resilience of swarm storage. Tentative.

10.5 Podcasts

https://oktahedron.diskordia.org/?podcast=oh003-Swarm

10.6 Videos

Aron Fischer, Louis Holbrook, Daniel A. Nagy: Swarm Development Update - Devcon3 Cancun, November 2017

Viktor Trón and Aron Fischer - Swap, Swear and Swindle Games - Devcon3 Cancun, November 2017

---

sw3 London

Louis Holbrook: Resource Updates - EthCC, Paris, March 2018

Daniel A Nagy: Encryption in Swarm - EthCC, Paris, March 2018

Viktor Tron Base layer infrastructure services for web3 - EthCC, Paris, March 2018

Louis Holbrook (Ethersphere, Jaak) PSS - Node to node Communication Over Swarm - Devcon3 Cancun, November 2017

Daniel A Nagy - Scalable Responsive Đapps with Swarm and ENS - Devcon3 Cancun, November 2017

Aron Fischer - Data retrieval in Swarm - Swarm Orange Summit, Berlin, June 2017

Zahoor Mohamed (EF, Swarm team): Swarm Fuse Demo - Ethereum Meetup, Berlin, June 2017

Daniel Nagy: Network topology for distributed storage - Swarm Orange Summit, Berlin, June 2017

Fabian Vogelsteller - Swarm Integration in Mist - Swarm Orange Summit, Berlin, June 2017

Daniel Nagy (EF, Swarm team): Plausible Deniability (2 parts) - Swarm Orange Summit, Berlin, June 2017

Elad Verbin: Data structures and security on Swarm (2 parts) - Swarm orange summit, Berlin, June 2017

Louis Holbrook (Ethersphere, Jaak): PSS - internode messaging protocol - Swarm Orange Summit, Berlin, June 2017

Viktor Tron - Distributed Database Services - Swarm Orange Summit 2017

Viktor Tron - network testing framework and visualisation - Ethereum Meetup, Berlin, June 2017

Doug Petkanics (Livepeer): Realtime video streaming on Swarm - Swarm Orange Summit, Berlin, June 2017

Nick Johnson on the Ethereum Name System

Viktor Trón, Aron Fischer: Swap, Swear and Swindle. Swarm Incentivisation.

Viktor Trón: Towards Web3 Infrastructure.

Dániel A. Nagy: Developing Scalable Decentralized Applications for Swarm and Ethereum

Aron Fischer, Dániel A. Nagy, Viktor Trón: Swarm - Ethereum.

Viktor Trón, Nick Johnson: Swarm, web3, and the Ethereum Name Service.

Nagy Dániel, Trón Viktor: Ethereum és Swarm: okos szerződések és elosztott világháló.

Dániel Nagy: Swarm: Distributed storage for Ethereum, the Turing-complete blockchain.

Viktor Trón, Dániel A. Nagy: Swarm. Ethereum Devcon1, London, November 2015.

Dániel A. Nagy: Keeping the public record safe and accessible. Ethereum Devcon0, Berlin, December 2014.

Chapter 11

Indices and tables

- genindex
- modindex
- search

Index