THE BOOK OF SWARM

STORAGE AND COMMUNICATION INFRASTRUCTURE FOR SELF-SOVEREIGN DIGITAL SOCIETY

BACK-END STACK FOR THE DECENTRALISED WEB

Viktor Trón

v1.0 pre-release 7 - worked on November 17, 2020

the swarm is headed toward us

Satoshi Nakamoto

CONTENTS

LIST OF FIGURES

PROLEGOMENA

INTENDED AUDIENCE

The primary aim of this book is to capture the wealth of output of the first phase of the Swarm project, and to serve as a compendium for teams and individuals participating in bringing Swarm to life in the forthcoming stages.

The book is intended for the technically inclined reader who is interested in using Swarm in their development stack and wishes to better understand the motivation and design decisions behind the technology. Researchers, academics and decentralisation experts are invited to check our reasoning and audit the consistency of Swarm's overall design. Core developers and developers from the wider ecosystem who build components, tooling or client implementations, should benefit from the concrete specifications we present, as well as from the explanation of the thoughts behind them.

STRUCTURE OF THE BOOK

The book has three major parts. The Prelude (i), explains the motivation by describing the historical context, setting the stage for a fair data economy. We then presents the Swarm vision.

The second, Design and Architecture (ii), contains a detailed exposition of the design and architecture of Swarm. This part covers all areas relevant to Swarm's core functionality.

The third part, Specifications (iii), provides the formal specification of components. This is meant to serve as the reference handbook for Swarm client developers.

The index, glossary of terms and acronyms, and an appendix (iv) containing formalised arguments (A), complete the compendium.

HOW TO USE THE BOOK

The first two parts – the Prelude and Design and Architecture – can be read as one continuous narrative. Those wishing to jump right into the technology can start with the Design and Architecture part, skipping the Prelude.

A Swarm client developer can start from any particular component spec in the Specifications part and work their way back to Design and Architecture, via the in-text references whenever wider context is needed or one is interested in the justifications for the choices apparent in the specs.

ACKNOWLEDGMENTS

EDITING

Many helped me write this book, but a few that took very active roles in making it happen are due credit. Daniel Nickless is the man behind all the diagrams, we enjoyed many hours of my giving birth to impromptu visualisations and was happy to redo them several times. Dan is a virtuoso of Illustrator and has also become a latex expert to typeset some nice trees.

I am hugely indebted to Črt Ahlin who, beside managing the book project, has also contributed some top quality text. He also undertook many of the unrewarding tedious jobs of compiling the indexes and glossaries. Both Črt and Dan (a native speaker of English) did an amazing job at proof-reading and correcting mistakes and typos.

Special thanks is due to Edina Lovas whose support and enthusiasm has always helped push me along.

AUTHORS

Swarm is co-fathered by my revered friend and colleague, the awesome Daniel A. Nagy. Daniel invented the fundamental design of Swarm and should get the credit for quite a few major architectural decisions, innovative ideas and formal insights presented in this book.

Aron Fisher's contribution to Swarm would be hard to overstate. Most of what is Swarm now started or got worked out in sessions with Aron. He not only contributed ingredients overall, but also coauthored the first few orange papers and not least was always at the forefront, presenting and explaining our tech in conferences and meetups.

I thank Daniel and Aron for bearing with me, suffering my sloppy, half-baked ideas, bringing clarity and always understanding the maths. Without claiming full endorsement or any responsibility for what is written here, I would consider Daniel and Aron as co-authors of this book.

I owe deep gratitude to my partner in crime Gregor Žavcer who is basically running the project currently. Gregor's honest fascination and unrelenting dedication to the project is what kept me going through many a low moment in the past. Our shared vision of decentralised future of data economy ranging from technological innovation to ethical direction served as the foundation for our collaboration. A great

part of this book got shaped during our night-long brainstorming sessions. Gregor even contributed content in the book.

I would like to thank Rinke Hendriksen, who contributed significant insight and innovation, mainly in the area of incentivisation. His economic theory background continues to prove invaluable in understanding our incentive design. Deep discussions with him led to new solutions, many improvements and insight. He currently manages development as product owner.

People who not only have a major part in coming up with the ideas but inadvertently contributed actual text in the form of excepts from earlier work are Daniel A. Nagy, Aron Fisher. Also Fabio Barone on incentives, Javier Peletier on feeds and Louis Holbrook on feeds and pss.

feedback

The book benefited immensely from feedback. Those who went through the pain of reading early versions and commented on the work in progress giving their criticism or pointing out unclarities deserve to be mentioned: Henning Dietrich, Brendan Graetz, Marcelo Ortelli, Santiago Regusci, Vojtěch Šimetka of IOV labs, Attila Lendvai, Abel Bodo, Elad Nachmias, Janoš Guljaš, Petar Radović, Louis Holbrook, Zahoor Mohamed.

conception and influences

The book of Swarm is itself an expression of the grand idea of Swarm: the [pursuit of] the vision of decentralised storage and messaging on top of the blockchain. The concept and first formulation of Swarm as one of the pillars of a holy trinity to realise the Decentralised Web appeared before the launch of Ethereum in early 2015. It was by the Ethereum founders Vitalik Buterin, Gavin Wood and Jeffrey Wilcke that Swarm was trolled onto the slippery slope of bee jokes and geek humor. The protocol tags *bzz* and *shh* were both coined by Vitalik.

People who were close to the cradle of Swarm are Alex Leverington, Felix Lange early discussions with whom catalysed fundamental decisions that led to the design of Swarm as it now presents.

The foundations of Swarm were laid over the course of 2015. Daniel worked with Zsolt Felföldi, of light client fame, whose code is still being seen here and there in the Go Ethereum-based Swarm implementation. His ideas clearly have a hallmark on what Swarm set out to be.

We are hugely indebted to Elad Verbin, who for years has been acting as a scientific as well as strategic advisor to Swarm. Elad put considerable effort into the project, his insight and depth is in all areas of Swarm are unparalleled, his insight regarding the isomorphism between pointer-based functional data structures and content addressed

distributed data had a major impact on how we handle higher level functionality. Our work on a swarm interpreter inspired the Swarm script.

Special thanks due to Daniel Varga, Attila Lendvai, Attila Gazso for long nights of brainstorming, I learnt an awful lot from you. Thanks to Alexey Akhunov, Jordi Baylina for major technical insight and inspiration. My very special friend Anand Jaisingh deserves my gratitude for his unshakeable trust in me and the project and unique inspiration and synergy that was catalised by his presence in the halo of Swarm.

Early in the project, we spent quite some time with Alex van der Sande, Fabian Vogelsteller discussing Swarm and its potentials. Many ideas that came to life as a result, including manifests and the HTTP API owe them credit. People in or around the Ethereum Foundation who shaped the idea of Swarm include Taylor Gerring, Christian Reitwiessner, Stephan Tual and Alex Beregszaszi, Piper Merriam and Nick Savers.

team

First and foremost, thanks to Jeffrey Wilcke, one of the three founders, and team lead of the Amsterdam go-ethereum team. He was supporting the Ethersphere subteam Dani, Fefe and me, protecting the project in times of austerity. I am forever grateful to all current and past members of the Swarm team: ethernal gratitude to Nick Johnson who, during the brief period he was on the swarm team, created ENS. Thanks to those special ones longest on the team: Louis Holbrook, Zahoor Mohamed and Fabio Barone their creativity and faith helped us through rough times. Thanks to Anton Evangelatov, Bálint Gábor, Janoš Guljaš and Elad Nachmias for their massive contribution to the codebase. Thanks to Ferenc Szabo, Vlad Gluhovsky, Rafael Matias and many that cannot be named but contributed to the code. Ralph Pichler deserves a special mention, he has been a keen follower and supporter of our project for many years and gradually became an honorary member, he implemented the initial version of the entire smart contract suite for swap, swear and swindle, and been driving the development of blockchain interaction and key management in the recent year.

Major kudos to Janoš Guljaš who bravely took over the role of engineering lead and created a new killer team in Belgrade with the excellent Petar Radović, Svetomir Smiljković and Ivan Vandot.

I am grateful to Vojtěch Šimetka and the Swarm contingent at IOV labs who basically saved the project, Marcelo Ortelli, Santiago Regusci are major contributors to the current codebase alongside the Belgrade team.

I would also like to thank Tim Bansemer, who is one of a kind, with unimaginable effectiveness and drive, his contributions will always be felt in and around the team, the code, the documentation, and the processes.

ECOSYSTEM

Swarm has always attracted a wide community of enthusiasts as well as an ecosystem of companies and projects whose support and encouragement kept us alive during some dark days. Jarrad Hope, Jacek Sieka, Oscar Thoren of Status, Marcin Rabenda of Consensys, Tadej Fius, Zenel Batagelj from Datafund, Javier and Alfonso Peletier of Epic Labs, Eric Tang and Doug Petkanics of LivePeer, Sourabh Niyogi of Wolk, Vaughn McKenzie, Fred Tibbles and Oren Sokolovsky of JAAK, Carl Youngblood, Paul Le Cam, Shane Howley, Doug Leonard from Mainframe, Anand Jaisingh, Dimitry Kholkhov, Igor Sharudin of BeeFree, Attila Gazso from the Felfele Foundation.

I would like to thank all participants of Swarm Summits, hack weeks and other gatherings. Many of the ideas in this book developed as a result of conversations on these events. I want to thank Michelle Thuy, Kevin Mohanan.

Finally, I want to express gratitude to my mentors and teachers, András Kornai, László Kálmán, Péter Halácsy and Péter Rebrus who shaped my thinking and skills and will always be my intellectual heroes.

Part I

PRELUDE

1

THE EVOLUTION

This chapter gives background information about the motivation for and evolution of Swarm and its vision today. 1.1 lays out a historical analysis of the World Wide Web, focusing on how it became the place it is today. 1.2 introduces the concept, and explains the importance of data sovereignty, collective information and a fair data economy. It discusses the infrastructure a self-sovereign society will need in order to be capable of collectively hosting, moving and processing data. Finally, 1.3 recaps the values behind the vision, spells out the requirements of the technology and lays down the design principles that guide us in manifesting Swarm.

1.1 HISTORICAL CONTEXT

While the Internet in general – and the World Wide Web (WWW) in particular – dramatically reduced the costs of disseminating information, putting a publisher's power at every user's fingertips, these costs are still not zero and their allocation heavily influences who gets to publish what content and who will consume it.

In order to appreciate the problems we are trying to solve, a little journey into the history of the evolution of the World Wide Web is useful.

1.1.1 *Web 1.0*

In the times of Web 1.0, in order to have your content accessible to the whole world, you would typically fire up a web server or use some free or cheap web hosting space to upload your content that could then be navigated through a set of HTML pages. If your content was unpopular, you still had to either maintain the server or pay the hosting to keep it accessible, but true disaster struck when, for one reason or another, it became popular (e.g. you got "slashdotted"). At this point, your traffic bill skyrocketed just before either your server crashed under the load or your hosting provider throttled your bandwidth to the point of making your content essentially unavailable for the majority of your audience. If you wanted to stay popular you had to invest in high availability clusters connected with fat pipes; your costs grew together with your popularity, without any obvious way to cover them. There were very few practical ways to allow (let alone require) your audience to share the ensuing financial burden directly.

The common wisdom at the time was that it would be the internet service provider (ISP) that would come to the rescue since in the early days of the Web revolution, bargaining about peering arrangements between the ISP's involved arguments about where providers and consumers were located, and which ISP was making money from which other's network. Indeed, when there was a sharp imbalance between originators of TCP connection requests (aka SYN packets), it was customary for the originator ISP to pay the recipient ISP, which made the latter somewhat incentivised to help support those hosting popular content. In practice, however, this incentive structure usually resulted in putting a free *pron* or *warez* server in the server room to tilt back the scales of SYN packet counters. Blogs catering to a niche audience had no way of competing and were generally left out in the cold. Note, however, that back then, creator-publishers still typically owned their content.

1.1.2 *Web 2.0*

The transition to Web 2.0 changed much of that. The migration from a personal home page running on one's own server using Tim Berners Lee's elegantly simplistic and accessible hypertext markup language toward server-side scripting using cgi-gateways, perl and the inexorable php had caused a divergence from the beautiful idea that anyone could write and run their own website using simple tools. This set the web on a path towards a prohibitively difficult and increasingly complex stack of scripting languages and databases. Suddenly the world wide web wasn't a beginner friendly place any more, and at the same time new technologies began to make it possible to create web applications which could provide simple user interfaces to enable unskilled publishers to simply POST their data to the server and divorce themselves of the responsibilities of actually delivering the bits to their end users. In this way, the Web 2.0 was born.

Capturing the initial maker spirit of the web, sites like MySpace and Geocities now ruled the waves. These sites offered users a piece of the internet to call their own complete with as many scrolling text marquees, flashing pink glitter Comic Sans banners and all the ingenious XSS attacks a script kiddie could dream of. It was a web within the web, an accessible and open environment for users to start publishing their own content increasingly without need to learn HTML, and without rules. Platforms abounded and suddenly there was a place for everyone, a phpBB forum for any niche interest imaginable. The web became full of life and the dotcom boom showered Silicon Valley in riches.

Of course, this youthful naivete, the fabulous rainbow coloured playground wouldn't last. The notoriously unreliable MySpace fell victim to its open policy of allowing scripting. Users' pages became unreliable and the platform became unusable. When Facebook arrived with a clean-looking interface that worked, MySpace's time was up and people migrated in droves. The popular internet acquired a more self-important undertone, and we filed into the stark white corporate office of Facebook. But there

was trouble in store. While offering this service for 'free,' Mr. Zuckerberg and others had an agenda. In return for hosting our data, we (the dumb f*cks [Carlson, 2010]) would have to trust him with it. Obviously, we did. For the time being, there was ostensibly no business model, beyond luring in more venture finance, amassing huge user-bases and we'll deal with the problem later. But from the start, extensive and unreadable T&C's gave all the rights to the content to the platforms. While in the Web 1.0 it was easy to keep a backup of your website and migrate to a new host, or simply host it from home yourself, now those with controversial views had a new verb to deal with: 'deplatformed'.

At the infrastructure level, this centralisation began to manifest itself in unthinkably huge data-centers. Jeff Bezos evolved his book-selling business to become the richest man on Earth by facilitating those unable to deal with the technical and financial hurdles of implementing increasingly complex and expensive infrastructure. At any rate, this new constellation was capable of dealing with those irregular traffic spikes that had crippled widely successful content in the past. When others followed, soon, enough huge amounts of the web began to be hosted by a handful huge companies. Corporate acquisitions and endless streams of VC money effected more and more concentration of power. A forgotten alliance of the open source programmers who created the royalty free Apache web server, and Google, who provided paradigm-shifting ways to organise and access the exponential proliferation of data helped dealing a crippling blow to Microsoft's attempt to force the web into a hellish, proprietary existence, forever imprisoned in Internet Explorer 6. But of course, Google eventually accepted 'parental oversight,' shelved its promise to 'do no evil' and succumbed to its very own form of megalomania and began to eat the competition. Steadily, email became Gmail, online ads became AdSense and Google crept into every facet of daily life in the web.

On the surface, everything was rosy. Technological utopia hyper-connected the world in a way no-one could have imagined. No longer was the web just for academics and the super 1337, it made the sum of human knowledge available to anyone, and now that smartphones became ubiquitous, it could be accessed anywhere. Wikipedia, gave everyone superhuman knowledge, Google allowed us to find and access it in a moment and Facebook gave us the ability to communicate with everyone we had ever known, for free. However, underneath all this, there was one problem buried just below the glittering facade. Google knew what they were doing. So were Amazon, Facebook and Microsoft. So did some punks, since 1984.

The time came to cut a cheque to the investors, once the behemoth platforms had all the users. The time to work out a business model had come. To provide value back to the shareholders, the content providing platforms found advertising revenue as panacea. And little else. Google probably really tried but could not think of any alternative. Now the web started to get complicated, and distracting. Advertising appeared everywhere and the pink flashing glitter banners were back, this time pulling

your attention from the content you came for to deliver you to the next user acquisition opportunity.

And as if this weren't enough, there were more horrors to come. The Web lost the last shred of its innocence when the proliferation of data became unwieldy and algorithms were provided to 'help' to better provide us access to the content that we want. Now the platforms had all our data, they were able to analyse it to work out what we wanted to see, seemingly knowing us even better than we ever knew ourselves. Everyone would be fed their most favourite snack, along with the products they would most likely buy. There was a catch to these secret algorithms and all encompassing data-sets: they were for sale to the highest bidder. Deep-pocketed political organisations were able to target swing voters with unprecedented accuracy and efficacy. Cyberspace became a very real thing all of sudden, just as consensus as a normality became a thing of the past. News did not only become fake, but personally targeted manipulation, as often as not to nudge you to act against your best interest, without even realising it.

The desire to save on hosting costs had turned everyone into a target to become a readily controllable puppet. Some deal.

At the same time, more terrifying revelations lay in wait. It turned out the egalitarian ideals that had driven the initial construction of a trustful internet were the most naive of all. In reality, the DoD had brought it to you, and now wanted it back. Edward Snowden walked out of the NSA with a virtual stack of documents no-one could have imagined. Instead of course, if you had taken the Bourne Conspiracy for being a documentary. It turned out that the protocols were broken, and all the warrant canaries long dead – the world's governments had been running a surveillance dragnet on the entire world population – incessantly storing, processing, cataloguing, indexing and providing access to the sum total of a persons online activity. It was all available at the touch of an XKeyStore button, an all seeing, unblinking Optical Nerve determined to 'collect it all' and 'know it all' no matter who or what the context. Big Brother turned out to look like Sauron. The gross erosion of privacy – preceded by many other, similar efforts by variously power-drunk or megalomaniac institutions and individuals across the world to track and block the packets of suppressed people, political adversaries or journalists, targeted by repressive regimes – had provided impetus for the Tor project. This unusual collaboration between the US Military, MIT and the EFF had responded by providing not only a way to obfuscate the origin of a request but also to serve up content in a protected, anonymous way. Wildly successful and a household name in some niches, it has not found much use outside them, due to a relatively high latency that results from its inherent inefficiencies.

By the time of the revelations of Snowden, the web had become ubiquitous and completely integral to almost every facet of human life, but the vast majority of it was run by corporations. While reliability problems had become a thing of the past, there was a price to pay. Context-sensitive, targeted advertising models now extended their Faustian bargain to content producers, with a grin that knew there was no alternative.

"We will give you scalable hosting that will cope with any traffic your audience throws at it", they sing, "but in return you must give us control over your content: we are going to track each member of your audience and collect (and own, *whistle*) as much of their personal data as we are able to. We will, of course, decide who can and who cannot see it, as is our right, no less. And we will proactively censor it, and naturally share your data with authorities whenever prudent to protect our business.". As a consequence, millions of small content producers created immense value for a hand-full of mega corporations, getting peanuts in return. Typically, free hosting and advertisement. What a deal!

Putting aside, for a moment, the resulting FUD of the Web 2.0 data and news apocalypse that we witness today, there are also a couple of technical problems with the architecture. The corporate approach has engendered a centralist maxim, so that all requests now must be routed through some backbone somewhere, to a monolith data-center, then passed around, processed, and finally returned back. Even if to simply send a message to someone in the next room. This is client-server architecture, which also – no afterthought – has at best flimsy security and was so often breached that it became the new normal, leaving the oil-slicks of unencrypted personal data and even plaintext passwords in its wake, spread all over the web. The last nail in the coffin is the sprawl of incoherent standards and interfaces this has facilitated. Today, spaghetti code implementations of growing complexity subdivide the web into multifarious micro-services. Even those with the deep pockets find it increasingly difficult to deal with the development bills, and it is common now that fledgling start-ups drown in a feature-factory sea of quickly fatal, spiralling technical debt. A modern web application's stack in all cases is a cobbled together Heath-Robinson machine comprising so many moving parts that it is almost impossible even for a supra-nation-state corporation to maintain and develop these implementations without numerous bugs and regular security flaws. Well, except Google and Amazon to be honest. At any rate. It is time for a reboot. In the end, it's the data describing our lives. They already try but yet they have no power to lock us into this mess.

1.1.3 *Peer-to-peer networks*

As the centrist Web 2.0 took over the world, the peer-to-peer (P2P) revolution was also gathering pace, quietly evolving in parallel. Actually, P2P traffic had very soon taken over the majority of packets flowing through the pipes, quickly overtaking the above mentioned SYN-bait servers. If anything, it proved beyond doubt that end-users, by working together to use their hitherto massively underutilised *upstream bandwidth*, could provide the same kind of availability and throughput for their content as previously only achievable with the help of big corporations and their data centers attached as they are to the fattest pipes of the Internet's backbone. What's more, it could be realized at a fraction of the cost. Importantly, users retained a lot more control and freedom over their data. Eventually, this mode of data distribution proved to be

remarkably resilient even in the face of powerful and well-funded entities exerting desperate means to shut it down.

However, even the most evolved mode of P2P file sharing, tracker-less BitTorrent [Pouwelse et al., 2005], was only that: file-level sharing. Which was not at all suitable for providing the kind of interactive, responsive experience that people were coming to expect from web applications on Web 2.0. In addition to this, while becoming extremely popular, BitTorrent was not conceived of with economics or game theory in mind, i.e. very much in the era before the world took note of the revolution it's namesake would precipitate: to say, before anyone understood blockchains and the power of cryptocurrency and incentivisation.

1.1.4 *The economics of BitTorrent and its limits*

The genius of BitTorrent lies in its clever resource optimisation [Cohen, 2003]: if many clients want to download the same content from you, give them different parts of it and in a second phase, let them swap the parts between each another in a tit-for-tat fashion, until everyone has all the parts. This way, the upstream bandwidth use of a user hosting content (the seeder in BitTorrent parlance) is roughly always the same, no matter how many clients want to download the content simultaneously. This solves the most problematic, ingrained issue of the ancient, centralised, master-and-slave design of Hypertext Transfer Protocol (HTTP), the protocol underpinning the World Wide Web.

Cheating (i.e. feeding your peers with garbage) is discouraged by the use of hierarchical, piece-wise hashing, whereby a package offered for download is identified by a single short hash and any part of it can be cryptographically proven to be a specific part of the package without knowledge about any of the other parts, and at the cost of only a very small computational overhead.

But this beautifully simple approach has five consequential shortcomings, [Locher et al., 2006, Piatek et al., 2007], all somewhat related.

- *lack of economic incentives* – There are no built-in incentives to seed downloaded content. In particular, one cannot exchange one's upstream bandwidth provided by seeding one's content, for downstream bandwidth required for downloading other content. Effectively, the upstream bandwidth provided by seeding content to users is not rewarded. Because as much upstream as possible can improve the experience with some online games, it can be a rational if selfish choice to switch seeding off. Add some laziness and it stays off forever.

- *initial latency* – Typically, downloads start slowly and with some delay. Clients that are further ahead in downloading have significantly more to offer to newcomers than they can offer in return. I.e. the newcomers have nothing to download (yet) for those further ahead. The result of this is that BitTorrent downloads start as a trickle before turning into a full-blown torrent of bits. This

peculiarity has severely limited the use of BitTorrent for interactive applications that require both fast responses and high bandwidth. Even though it would otherwise constitute a brilliant solution for many games.

- *lack of fine-granular content addressing* – Small chunks of data can only be shared as parts of the larger file that they are part of. They can be pinpointed for targeted that leaves the rest of a file out to optimise access. But peers for the download can only be found by querying the distributed hash table (DHT) for a desired *file*. It is not possible to look for peers at the chunk-level, because the advertising of the available content happens exclusively at the level of files. This leads to inefficiencies as the same chunks of data can often appear verbatim in multiple files. So, while theoretically, all peers who have the chunk could provide it, there is no way to find those peers, because only its enveloping file has a name (or rather, an announced hash) and can be sought for.

- *no incentive to keep sharing* – Nodes are not rewarded for their sharing efforts (storage and bandwidth) once they have achieved their objective, i.e. retrieving all desired files from their peers.

- *no privacy or ambiguity* – Nodes advertise exactly the content they are seeding. It is easy for attackers to discover the IP address of peers hosting content they would like to see removed, and then as a simple step for adversaries to DDOS them, or for corporations and nation states to petition the ISP for the physical location of the connection. This has led to a grey market of VPN providers helping users circumvent this. Although these services offer assurances of privacy, it is usually impossible to verify them as their systems are usually closed source.

To say, while spectacularly popular and very useful, BitTorrent is at the same time primitive, a genius first step. It is how far we can get simply by sharing our upstream bandwidth, hard-drive space, and a tiny amounts of computing power – without proper accounting and indexing. However – surprise! – if we add just a few more emergent technologies to the mix, most importantly of course, the blockchain, we get something that truly deserves the Web 3.0 moniker: a decentralised, censorship-resistant device for sharing, and also for collectively creating content, all while retaining full control over it. What's more, the cost of this is almost entirely covered by using and sharing the resources supplied by the breathtakingly powerful, underutilized super-computer (by yesteryear's standards :-)) that you already own.

1.1.5 *Towards Web 3.0*

```
The Times 03/
Jan/2009 Chancel
lor on brink of
second bailout f
or banks
```

On (or after) 6:15 Saturday the 3rd of January 2009, the world changed forever. A mysterious Cypherpunk created the first block of a chain that would come to encircle the entire world, and the genie was out of the bottle. This first step would put in motion a set of reactions that would result in an unprecedentedly humongous amount of money flowing from the traditional reservoirs of fiat and physical goods into a totally new vehicle to store and transmit value: cryptocurrency. 'Satoshi Nakamato' had managed to do something no-one else had been able to, he had, de facto, yet at small scale, disintermediated the banks, decentralised trustless value transfer, and since that moment, we are effectively back at gold standard: everyone can now own central bank money. Money that no-one can multiply or inflate out of your pocket. What's more: everyone can now print money themselves that comes with its own central bank and electronic tranmission system. It is still not well understood how much this will change our economies.

This first step was a huge one and a monumental turning point. Now we had authentication and value transfer baked into the system at its very core. But as much as it was conceptually brilliant, it had some minor and not so minor problems with utility. It allowed to transmit digital value, one could even 'colour' the coins or transmit short messages like the one above that marks the fateful date of the first block. But that's it. And, regarding scale ... every transaction must be stored on every node. Sharding was not built-in. Worse, the protection if the digital money made it necessary that every node processed exactly the same as every other node, all the time. This was the opposite of a parallelised computing cluster and millions of times slower.

When Vitalik conceived of Ethereum, he accepted some of these limitations but the utility of the system took a massive leap. He added the facility for Turing-complete computation via the Ethereum Virtual Machine (EVM) which enabled a cornucopia of applications that would run in this trustless setting. The concept was at once a dazzling paradigm shift, and a consistent evolution of Bitcoin, which itself was based on a tiny virtual machine, with every single transactions really being – unbeknownst to many – a mini program. But Ethereum went all the way and that again changed everything. The possibilities were numerous and alluring and Web 3.0 was born.

However, there was still a problem to overcome when transcending fully from the Web 2.0 world: storing data on the blockchain was prohibitively expensive for anything but a tiny amount. Both Bitcoin and Ethereum had taken the layout of BitTorrent and run with it, complementing the architecture with the capability to transact but leaving any consideration about storage of non-systemic data for later. Bitcoin had in fact added a second, much less secured circuit below the distribution of blocks: candidate transactions are shipped around without fanfare, as secondary citizens, literally without protocol. Ethereum went further, separated out the headers from the blocks, creating a third tier that ferried the actual block data around ad-hoc, as needed. Because both classes of data are essential to the operation of the system, these could be called critical design flaws. Bitcoin's maker probably didn't envision a reality where mining had become the exclusive domain of a highly specialized elite. Any transactor

will have been thought to be able to basically mine their own transactions. Ethereum faced the even harder challenge of data availability and presumably because it was always obvious that the problem could be addressed separately later, just ignored it for the moment.

In other news, the straightforward approach for data dissemination of BitTorrent had successfully been implemented for web content distribution by ZeroNet [ZeroNet community, 2019]. However, because of the aforementioned issues with BitTorrent, ZeroNet turned out unable to support the responsiveness that users of web services have come to expect.

In order to try to enable responsive, distributed web applications (or dapps), the InterPlanetary File System (IPFS) [IPFS, 2014] introduced their own major improvements over BitTorrent. A stand-out feature being the highly web-compatible, URL-based retrieval scheme. In addition, the directory of the available data, the indexing, (like BitTorrent organized as a DHT) was vastly improved, making it possible to also search for a small part of any file, called a chunk.

There are numerous other efforts to fill the gap and provide a worthy Web 3.0 surrogate for the constellation of servers and services that have come to be expected by a Web 2.0 developer, to offer a path to emancipation from the existing dependency on the centralized architecture that enables the data reapers. These are not insignificant roles to supplant, even the most simple web app today subsumes an incredibly large array of concepts and paradigms which have to be remapped into the trustless setting of Web 3.0. In many ways, this problem is proving to be perhaps even more nuanced than implementing trustless computation in the blockchain. Swarm responds to this with an array of carefully designed data structures, which enable the application developer to recreate concepts we have grown used to in Web 2.0, in the new setting of Web 3.0. Swarm successfully reimagines the current offerings of the web, re-implemented on solid, cryptoeconomic foundations.

Imagine a sliding scale, starting on the left with: large file size, low frequency of retrieval and a more monolithic API; to the right: small data packets, high frequency of retrieval, and a nuanced API. On this spectrum, file storage and retrieval systems like a posix filesystem, S3, Storj and BitTorrent live on the left hand side. Key–value stores like LevelDB and databases like MongoDB or Postgres live on the right. To build a useful app, different modalities, littered all over the scale are needed, and furthermore there must be the ability to combine data where necessary, and to ensure only authorised parties have access to protected data. In a centrist model, it is easy to handle these problems initially, getting more difficult with growth, but every range of the scale has a solution from one piece of specialized software or another. However, in the decentralised model, all bets are off. Authorisation must be handled with cryptography and the combination of data is limited by this. As a result, in the nascent, evolving Web 3.0 stack of today, many solutions deal piecemeal with only part of this spectrum of requirements. In this book, you will learn how Swarm spans the entire spectrum, as well as providing high level tools for the new guard of Web 3.0

developers. The hope is that from an infrastructure perspective, working on Web 3.0 will feel like the halcyon days of Web 1.0, while delivering unprecedented levels of agency, availability, security and privacy.

To respond to the need for privacy to be baked in at the root level in file-sharing – as it is so effectively in Ethereum – Swarm enforces anonymity at an equally fundamental and absolute level. Lessons from Web 2.0 have taught us that trust should be given carefully and only to those that are deserving of it and will treat it with respect. Data is toxic [Schneier, 2019], and we must treat it carefully in order to be responsible to ourselves and those for whom we take responsibility. We will explain later, how Swarm provides complete and fundamental user privacy.

And of course, to fully transition to a Web 3.0-decentralised world, we also deal with the dimensions of incentives and trust, which are traditionally 'solved' by handing over responsibility to the (often untrustworthy) centralised gatekeeper. As we have noted, this is one problem that BitTorrent also struggled to solve, and that it responded to with a plethora of seed ratios and private (i.e., centralised) trackers.

The problem of lacking incentive to reliably host and store content is apparent in various projects such as ZeroNet or MaidSafe. Incentivisation for distributed document storage is still a relatively new research field, especially in the context of blockchain technology. The Tor network has seen suggestions [Jansen et al., 2014, Ghosh et al., 2014] but these schemes are mainly academic, they are not built into the heart of the underlying system. Bitcoin has been repurposed to drive other systems like Permacoin [Miller et al., 2014], some have created their own blockchain, such as Sia [Vorick and Champine, 2014] or Filecoin [Filecoin, 2014] for IPFS. BitTorrent is currently testing the waters of blockchain incentivisation with their own token [Tron Foundation, 2019, BitTorrent Foundation, 2019]. However, even with all of these approaches combined, there would still be many hurdles to overcome to provide the specific requirements for a Web 3.0 dapp developer.

We will see later how Swarm provides a full suite of incentivisation measures, as well as other checks and balances to ensure that nodes are working to benefit the whole of the ... swarm. This includes the option to rent out large amounts of disk space to those willing to pay for it – irrespective of the popularity of their content – while ensuring that there is also a way to deploy your interactive dynamic content to be stored in the cloud, a feature we call upload and disappear.

The objective of any incentive system for peer-to-peer content distribution is to encourage cooperative behavior and discourage freeriding: the uncompensated depletion of limited resources. The incentive strategy outlined here aspires to satisfy the following constraints:

- It is in the node's own interest, regardless of whether other nodes follow it.
- It must be expensive to expend the resources of other nodes.
- It does not impose unreasonable overhead.

- It plays nice with "naive" nodes.
- It rewards those that play nice, including those following this strategy.

In the context of Swarm, storage and bandwidth are the two most important limited resources and this is reflected in our incentives scheme. The incentives for bandwidth use are designed to achieve speedy and reliable data provision, while the storage incentives are designed to ensure long term data preservation. In this way, we ensure that all requirements of web application development are provided for – and that incentives are aligned so that each individual node's actions benefit not only itself, but the whole of the network.

1.2 FAIR DATA ECONOMY

In the era of Web 3.0, the Internet is no longer just a niche where geeks play, but has become a fundamental conduit of value creation and a huge share of overall economic activity. Yet the data economy in it's current state is far from fair, the distribution of the spoils is under the control of those who control the data - mostly companies keeping the data to themselves in isolated data silos. To achieve the goal of a fair data economy many social, legal and technological issues will have to be tackled. We will now describe some of the issues as they currently present and how they will be addressed by Swarm.

1.2.1 *The current state of the data economy*

Digital mirror worlds already exist, virtual expanses that contain shadows of physical things and consist of unimaginably large amounts of data [Economist, 2020a]. Yet more and more data will continue to be synced to these parallel worlds, requiring new infrastructure and markets, and creating new business opportunities. Only relatively crude measures exist for measuring the size of the data economy as a whole, but for the USA, one figure puts the financial value of data (with related software and intellectual property) at $1.4trn-2trn in 2019 [Economist, 2020a]. The EU Commission projects the figures for the data economy in the EU27 for 2025 at €829bln (up from €301bln in 2018) [European Commission, 2020a].

Despite this huge amount of value, the asymmetric distribution of the wealth generated by the existing data economy has been put forward as a major humanitarian issue [Economist, 2020c]. As efficiency and productivity continue to rise, as a results of better data, the profits that result from this will need to be distributed. Today, the spoils are distributed unequally: the larger the companies' data set, the more it can learn from the data, attract more users and hence even more data. Currently, this is most apparent with the dominating large tech companies such as FAANG, but it is predicted that this will also be increasingly important in non-tech sectors, even nation states. Hence, companies are racing to become dominant in a particular sector,

and countries hosting these platforms will gain an advantage. As Africa and Latin America host so few of these, they risk becoming exporters of raw data and then paying other countries to import the intelligence provided, as has been warned by the United Nations Conference on Trade and Development [Economist, 2020c]. Another problem is that if a large company monopolises a particular data market, it could also become the sole purchaser of data - maintaining a complete control of setting prices and affording the possibility that the "wages" for providing data could be manipulated to keep them artificially low. In many ways, we are already seeing evidence of this.

Flows of data are becoming increasingly blocked and filtered by governments, using the familiar reasoning based on the protection of citizens, sovereignty and national economy [Economist, 2020b]. Leaks by several security experts have shown that for governments to properly give consideration to national security, data should be kept close to home and not left to reside in other countries. GDPR is one such instance of a "digital border" that has been erected – data may leave the EU only if appropriate safeguards are in place. Other countries, such as India, Russia and China, have implemented their own geographic limitations on data. The EU Commission has pledged to closely monitor the policies of these countries and address any limits or restrictions to data flows in trade negotiations and through the actions in the World Trade Organization [European Commission, 2020b].

Despite this growing interest in the ebb and flow of data, the big tech corporations maintain a firm grip on much of it, and the devil is in the details. Swarm's privacy-first model requires that no personal data be divulged to any third parties, everything is end-to-end encrypted out of the box, ending the ability of service providers to aggregate and leverage giant datasets. The outcome of this is that instead of being concentrated at the service provider, control of the data remains decentralised and with the individual to which it pertains. And as a result, so does the power. Expect bad press.

1.2.2 *The current state and issues of data sovereignty*

As a consequence of the Faustian bargain described above, the current model of the World Wide Web is flawed in many ways. As a largely unforeseen consequence of economies of scale in infrastructure provision, as well as network effects in social media, platforms became massive data silos where vast amounts of user data passes through, and is held on, servers that belong to single organisations. This 'side-effect' of the centralized data model has allowed large private corporations the opportunity to collect, aggregate and analyse user data, positioning their data siphons right at the central bottleneck: the cloud servers where everything meets. This is exactly what David Chaum predicted in 1984, kicking off the Cypherpunk movement that Swarm is inspired by.

The continued trend of replacing human-mediated interactions with computer-mediated interactions, combined with the rise of social media and the smartphone,

has resulted in more and more information about our personal and social lives becoming readily accessible to the companies provisioning the data flow. These have unraveled lucrative data markets where user demographics are linked with underlying behavior, to understand you better than you understand yourself. A treasure trove for marketeers.

Data companies have meanwhile evolved their business models towards capitalising on the sale of the data, rather than the service they initially provided. Their primary source of revenue is now selling the results of user profiling to advertisers, marketeers and others who would seek to 'nudge' members of the public. The circle is closed by showing such advertisement to users on the same platforms, measuring their reaction, thus creating a feedback loop. A whole new industry has grown out of this torrent of information, and as a result, sophisticated systems emerged that predict, guide and influence ways for users to entice them to allocate their attention and their money, openly and knowingly exploiting human weaknesses in responding to stimuli, often resorting to highly developed and calculated psychological manipulation. The reality is, undisputedly, that of mass manipulation in the name of commerce, where not even the most aware can truly exercise their freedom of choice and preserve their intrinsic autonomy of preference regarding consumption of content or purchasing habits.

The fact that business revenue is coming from the demand for micro-targeted users to present adverts to is also reflected in quality of service. The content users' needs – who used to be and should continue to be the 'end' user – became secondary to the needs of the "real" customers: the advertiser, often leading to ever poorer user experience and quality of service. This is especially painful in the case of social platforms when inertia caused by a network effect essentially constitutes user lock-in. It is imperative to correct these misaligned incentives. In other words, provide the same services to users, but without such unfortunate incentives as they are resulting from the centralised data model.

The lack of control over one's data has serious consequences on the economic potential of the users. Some refer to this situation, somewhat hysterically, as data slavery. But they are technically correct: our digital twins are captive to corporations, put to good use for them, without us having any agency, quite to the contrary, to manipulate us out of it and make us less well informed and free.

The current system then, of keeping data in disconnected datasets has various drawbacks:

- *unequal opportunity* - Centralised entities increase inequality as their systems siphon away a disproportionate amount of profit from the actual creators of the value.
- *lack of fault tolerance* - They are a single point of failure in terms of technical infrastructure, notably security.
- *corruption* - The concentration of decision making power constitutes an easier target for social engineering, political pressure and institutionalised corruption.

- *single attack target* - The concentration of large amounts of data under the same security system attrackts attacks as it increases the potential reward for hackers.

- *lack of service continuity guarantees* - Service continuity is in the hands of the organisation, and is only weakly incentivised by reputation. This introduces the risk of inadvertent termination of the service due to bankruptcy, or regulatory or legal action.

- *censorship* - Centralised control of data access allows for, and in most cases eventually leads to, decreased freedom of expression.

- *surveillance* - Data flowing through centrally owned infrastructure offers perfect access to traffic analysis and other methods of monitoring.

- *manipulation* - Monopolisation of the display layer allows the data harvesters to have the power to manipulate opinions by choosing which data is presented, in what order and when, calling into question the sovereignty of individual decision making.

1.2.3 *Towards self-sovereign data*

We believe that decentralisation is a major game-changer, which by itself solves a lot of the problems listed above.

We argue that blockchain technology is the final missing piece in the puzzle to realise the cypherpunk ideal of a truly self-sovereign Internet. As Eric Hughes argued in the *Cypherpunk Manifesto* [Hughes, 1993] already in 1993, "We must come together and create systems which allow anonymous transactions." One of the goals of this book is to demonstrate how decentralised consensus and peer-to-peer network technologies can be combined to form a rock-solid base-layer infrastructure. This foundation is not only resilient, fault tolerant and scalable; but also egalitarian and economically sustainable, with a well designed system of incentives. Due to a low barrier of entry for participants, the adaptivity of these incentives ensures that prices automatically converge to the marginal cost. On top of this, add Swarm's strong value proposition in the domain of privacy and security.

Swarm is a Web 3.0 stack that is decentralised, incentivised, and secured. In particular, the platform offers participants solutions for data storage, transfer, access, and authentication. These data services are more and more essential for economic interactions. By providing universal access to all for these services, with strong privacy guarantees and without borders or external restrictions, Swarm fosters the spirit of global voluntarism and represents the *infrastructure for a self-sovereign digital society*.

1.2.4 *Artificial intelligence and self-sovereign data*

Artificial Intelligence (AI) is promising to bring about major changes to our society. On the one hand, it is envisioned to allow for a myriad of business opportunities, while on the other hand it is expected to displace many professions and jobs, where not merely augmenting them [Lee, 2018].

The three "ingredients" needed for the prevalent type of AI today, machine learning (ML), are: computing power, models and data. Today, computing power is readily available and specialized hardware is being developed to further facilitate processing. An extensive headhunt for AI talent has been taking place for more than a decade and companies have managed to monopolise workers in possession of the specialised talents needed to work on the task to provide the models and analysis. However, the dirty secret of today's AI, and deep learning, is that the algorithms, the 'intelligent math' is already commoditised. It is Open Source and not what Google or Palantir make their money with. The true 'magic trick' to unleash their superior powers is to get access to the largest possible sets of data.

Which happen to be the organizations that have been systematically gathering it in data silos, often by providing the user with an application with some utility such as search or social media, and then stockpiling the data for later use very different from that imagined by the 'users' without their express consent and not even knowledge. This monopoly on data has allowed multinational companies to make unprecedented profits, with only feeble motions to share the financial proceeds with the people whose data they have sold. Potentially much worse though, the data they hoard is prevented from fulfilling its potentially transformative value not only for individuals but for society as a whole.

Perhaps it is no coincidence, therefore, that the major data and AI "superpowers" are emerging in the form of the governments of the USA and China and the companies that are based there. In full view of the citizens of the world, an AI arms-race is unfolding with almost all other countries being left behind as "data colonies" [Harari, 2020]. There are warnings that as it currently stands, China and the United States will inevitably accumulate an insurmountable advantage as AI superpowers [Lee, 2018].

It doesn't have to be so. In fact, it likely won't because the status quo is a bad deal for billions of people. Decentralised technologies and cryptography are the way to allow for privacy of data, and at the same time enable a fair data economy to gradually emerge that will present all of the advantages of the current centralised data economy, but without the pernicious drawbacks. This is the change that many consumer and tech organizations across the globe are working for, to support the push back against the big data behemoths, with more and more users beginning to realise that they have been swindled into giving away their data. Swarm will provide the infrastructure to facilitate this liberation.

Self-sovereign storage might well be the only way that individuals can take back control of their data and privacy, as the first step towards reclaiming their autonomy,

stepping out of the filter bubble and reconnect instead with their own culture. Swarm represents at its core solutions for many of the problems with today's Internet and the distribution and storage of data. It is built for privacy from the ground up, with intricate encryption of data and completely secure and leak-proof communication. Furthermore, it enables sharing of selected data with 3rd parties, on the terms of the individual. Payments and incentives are integral parts of Swarm, making financial compensation in return for granular sharing of data a core concern.

Because as Hughes wrote, "privacy in an open society requires anonymous transaction systems. ... An anonymous transaction system is not a secret transaction system. An anonymous system empowers individuals to reveal their identity when desired and only when desired; this is the essence of privacy."

Using Swarm will allow to leverage a fuller set of data and to create better services, while still having the option to contribute it to the global good with self-verifiable anonymisation. The best of all worlds.

This new, wider availability of data, e.g. for young academic students and startups with disruptive ideas working in the AI and big-data sectors, would greatly facilitate the evolution of the whole field that has so much to contribute to science, healthcare, eradication of poverty, environmental protection, disaster prevention, to name a few; but which is currently at an impasse, despite its eye-catching success for robber barons and rogue states. With the facilities that Swarm provides, a new set of options will open up for companies and service providers, different but no less powerful. With widespread decentralisation of data, we can collectively own the extremely large and valuable data sets that are needed to build state-of-the-art AI models. The portability of this data, already a trend that is being hinted at in traditional tech, will enable competition and – as before – personalised services for individuals. But the playing field will be levelled for all, driving innovation worthy of the year 2020.

1.2.5 *Collective information*

Collective information began to accumulate from the first emergence of the Internet, yet the concept has just recently become recognized and discussed under a variety of headings such as *open source*, *fair data* or *information commons*.

A collective, as defined by Wikipedia (itself an example of "collective information") is:

> "A group of entities that share or are motivated by at least one common issue or interest, or work together to achieve a common objective."

The internet allows collectives to be formed on a previously unthinkable scale, despite differences in geographic location, political convictions, social status, wealth, even general freedom, and other demographics. Data produced by these collectives, through joint interaction on public forums, reviews, votes, repositories, articles and polls is a form of collective information – as is the metadata that emerges from the traces of

these interactions. Since most of these interactions, today, are facilitated by for-profit entities running centralized servers, the collective information ends up being stored in data silos owned by a commercial entity, the majority concentrated in the hands of a few big technology companies. And while the actual work results are often in the open, as the offering of these providers, the metadata, which can often be the more valuable, powerful and dangerous representation of the interaction of contributors, is usually held and monetized secretly.

These "platform economies" have already become essential and are only becoming ever more important in a digitising society. We are, however, seeing that the information the commercial players acquire over their users is increasingly being used against the very users' best interests. To put it mildly, this calls into question whether these corporations are capable of bearing the ethical responsibility that comes with the power of keeping our collective information.

While many state actors are trying to obtain unfettered access to the collective mass of personal data of individuals, with some countries demanding magic key-like backdoor access, there are exceptions. Since AI has the potential for misuse and ethically questionable use, a number of countries have started 'ethics' initiatives, regulations and certifications for AI use, for example the German Data Ethics Commission or Denmark's Data Ethics Seal.

Yet, even if corporations could be made to act more trustworthy, as would be appropriate in the light of their great responsibility, the mere existence of data silos stifles innovation. The basic shape of the client-server architecture itself led to this problem, as it has made centralised data storage (on the 'servers' in their 'farms') the default (see 1.1.1 and 1.1.2). Effective peer-to-peer networks such as Swarm (1.1.3) now make it possible to alter the very topology of this architecture, thus enabling the collective ownership of collective information.

1.3 THE VISION

Swarm is infrastructure for a self-sovereign society.

1.3.1 *Values*

Self-sovereignty implies freedom. If we break it down, this implies the following metavalues:

- *inclusivity* - Public and permissionless participation.
- *integrity* - Privacy, provable provenance.
- *incentivisation* - Alignment of interest of node and network.
- *impartiality* - Content and value neutrality.

These metavalues can be thought of as systemic qualities which contribute to empowering individuals and collectives to gain self-sovereignty.

Inclusivity entails we aspire to include the underprivileged in the data economy; and to lower the barrier of entry to define complex data flows and to build decentralised applications. Swarm is a network with open participation: for providing services and permissionless access to publishing, sharing, and investing your data.

Users are free to express their intention as 'action' and have full authority to decide if they want to remain anonymous or share their interactions and preferences. The integrity of online persona is required.

Economic incentives serve to make sure participants' behaviour align with the desired emergent behaviour of the network (see 3).

Impartiality ensures content neutrality and prevents gate-keeping. It also reaffirms that the other three values are not only necessary but sufficient: it rules out values that would treat any particular group as privileged or express preference for particular content or data from any particular source.

1.3.2 *Design principles*

The Information Society and its data economy bring about an age where online transactions and big data are essential to everyday life and thus the supporting technology is critical infrastructure. It is imperative therefore, that this base layer infrastructure be *future proof*, i.e., provided with strong guarantees for continuity.

Continuity is achieved by the following generic requirements expressed as *systemic properties*:

- *stable* - The specifications and software implementations be stable and resilient to changes in participation, or politics (political pressure, censorship).
- *scalable* - The solution be able to accommodate many orders of magnitude more users and data than starting out with, without leading to prohibitive reductions in performance or reliability during mass adoption.
- *secure* - The solution is resistant to deliberate attacks, and immune to social pressure and politics, as well as tolerating faults in its technological dependencies (e.g. blockchain, programming languages).
- *self-sustaining* - The solution runs by itself as an autonomous system, not depending on human or organisational coordination of collective action or any legal entity's business, nor exclusive know-how or hardware or network infrastructure.

1.3.3 *Objectives*

When we talk about the 'flow of data,' a core aspect of this is how information has provable integrity across modalities, see table 1. This corresponds to the original

Ethereum vision of the world computer, constituting the trust-less (i.e. fully trustable) fabric of the coming datacene: a global infrastructure that supports data storage, transfer and processing.

dimension	model	project area
time	memory	storage
space	messaging	communication
symbolic	manipulation	processing

Table 1: Swarm's scope and data integrity aspects across 3 dimensions.

With the Ethereum blockchain as the CPU of the world computer, Swarm is best thought of as its "hard disk". Of course, this model belies the complex nature of Swarm, which is capable of much more than simple storage, as we will discuss.

The Swarm project sets out to bring this vision to completion and build the world computer's storage and communication.

1.3.4 *Impact areas*

In what follows, we try to identify feature areas of the product that best express or facilitate the values discussed above.

Inclusivity in terms of permissionless participation is best guaranteed by a decentralised peer-to-peer network. Allowing nodes to provide service and get paid for doing so will offer a zero-cash entry to the ecosystem: new users without currency can serve other nodes until they accumulate enough currency to use services themselves. A decentralised network providing distributed storage without gatekeepers is also inclusive and impartial in that it allows content creators who risk being deplatformed by repressive authorities to publish without their right to free speech being censored.

The system of economic incentives built into the protocols works best if it tracks the actions that incur costs in the context of peer-to-peer interactions: bandwidth sharing as evidenced in message relaying is one such action where immediate accounting is possible as a node receives a message that is valuable to them. On the other hand, promissory services such the commitment to preserve data over time must be rewarded only upon verification. In order to avoid the tragedy of commons problem, such promissory commitments should be guarded against by enforcing individual accountability through the threat of punitive measures, i.e. by allowing staked insurers.

Integrity is served by easy provability of authenticity, while maintaining anonymity. Provable inclusion and uniqueness are fundamental to allowing trustless data transformations.

1.3.5 *The future*

The future is unknown and many challenges lie ahead for humanity. What is certain in today's digital society is that to be sovereign and in control of our destinies, nations and individuals alike must retain access and control over their data and communication.

Swarm's vision and objectives stem from the decentralized tech community and its values, since it was originally designed to be the file storage component in the trinity which would form the world computer: Ethereum, Whisper, and Swarm.

It offers the responsiveness required by dapps running on the devices of users and well incentivised storage using any kind of storage infrastructure - from smartphone to high-availability clusters. Continuity will be guaranteed with well designed incentives for bandwidth and storage.

Content creators will get fair compensation for the content they will offer and content consumers will be paying for it. By removing the middlemen providers that currently benefit from the network effects, the benefits of these network effects will be spread throughout the network.

But it will be much more than that. Every individual, every device is leaving a trail of data. That data is picked up and stored in silos, its potential used up only in part and to the benefit of large players.

Swarm will be the go-to place for digital mirror worlds. Individuals, societies and nations will have a cloud storage solution, that will not depend on any one large provider.

Individuals will be able to fully control their own data. They will no longer have the need to be part of the data slavery, giving their data in exchange for services. Not only that, they will be able to organize into data collectives or data co-operatives - sharing certain kinds of data as commons for reaching common goals.

Nations will establish self-sovereign Swarm clouds as data spaces to cater to the emerging artificial intelligence industry - in industry, health, mobility and other sectors. The cloud will be between peers, though maybe inside an exclusive region, and third parties will not be able to interfere in the flow of data and communication - to monitor, censor, or manipulate it. Yet, any party with proper permissions will be able to access it thus hopefully levelling the playing field for AI and services based on it.

Swarm can, paradoxically, be the "central" place to store the data and enable robustness of accessibility, control, fair distribution of value and leveraging the data for benefiting individuals and society.

Swarm will become ubiquitous in the future society, transparently and securely serving data of individuals and devices to data consumers in the Fair data economy.

Part II

DESIGN AND ARCHITECTURE

The Swarm project is set out to build permissionless storage and communication infrastructure for tomorrow's self-sovereign digital society. From a developer's perspective, Swarm is best seen as public infrastructure that powers real-time interactive web applications familiar from the Web 2.0 era. It provides a low-level API to primitives that serve as building blocks of complex applications, as well as the basis for the tools and libraries for a Swarm-based Web 3.0 development stack. The API and the tools are designed to allow access to the Swarm network from any traditional web browser, so that Swarm can immediately provide a private and decentralised alternative to today's World Wide Web (WWW).

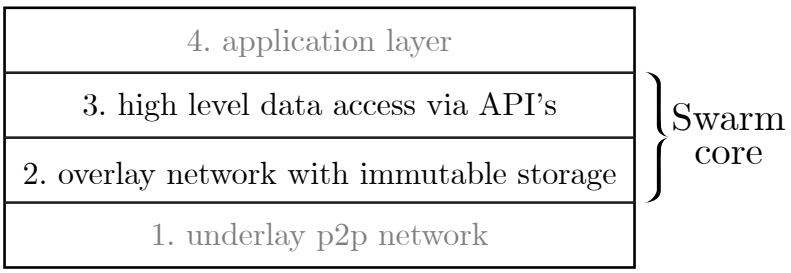


Figure 1: Swarm's layered design

This part details the design and architecture of the system. In accordance with the principles laid out in 1.3.2, we put an emphasis on modular design, and conceive of Swarm as having clearly separable layers, each dependent on the previous one (see figure 1):

- (1) *a peer-to-peer network protocol to serve as underlay transport,*
- (2) *an overlay network with protocols powering a distributed immutable storage of chunks (fixed size data blocks),*
- (3) *a component providing high-level data access and defining APIs for base-layer features, and*
- (4) *an application layer defining standards, and outlining best practices for more elaborate use-cases.*

We regard (2) and (3) as the core of Swarm. Since the network layer relies on it, we will formulate also the requirements for (1), but we consider the detailed treatment of both (1) and (4) outside the scope of this book.

Central to the design, the architecture of Swarm overlay network (layer 2 in figure 1) is discussed in chapter 2 and complemented by chapter 3, describing the system of economic incentives which makes Swarm self-sustaining. In chapter 4, we introduce the algorithms and conventions that allow Swarm to map data concepts on to the chunk layer to enable the high-level functionalities for storage and communication,

notably data structures such as filesystems and databases, access control, indexed feeds and direct messaging which comprise layer 3 of Swarm. In chapter 5, we present ways to prevent garbage collected chunks from disappearing from the network, including: erasure codes, pinning and insurance, and also provide ways to monitor, recover and re-upload them using missing chunk notifications and insurance challenges. Finally, in chapter 6 we will look at functionality from a user experience perspective.

2

NETWORK

This chapter offers a narrative on how the Swarm overlay network is built on top of a peer-to-peer network protocol to form a topology that allows for routing of messages between nodes (2.1). In 2.2, we describe how such a network can serve as a scalable distributed storage solution for data chunks (2.2.1) and present the logic underpinning the protocols for retrieval/download and syncing/upload (2.3).

2.1 TOPOLOGY AND ROUTING

This section sets the scene (2.1.1) for the overlay network (layer 2) of Swarm by making explicit the assumptions about the underlay network (layer 1). 2.1.2 introduces the overlay address space and explains how nodes are assigned an address. In 2.1.3, we present the Kademlia overlay topology (connectivity pattern) and explain how it solves routing between nodes. In 2.1.4 we show how nodes running the Swarm client can discover each other, bootstrap and then maintain the overlay topology.

2.1.1 *Requirements for underlay network*

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

1. *addressing* – Nodes are identified by their underlay address.
2. *dialling* – Nodes can initiate a direct connection to a peer by dialing them on their underlay address.
3. *listening* – Nodes can listen to other peers dialing them and can accept incoming connections. Nodes that do not accept incoming connections are called light nodes.
4. *live connection* – A node connection establishes a channel of communication which is kept alive until explicit disconnection, so that the existence of a connection means the remote peer is online and accepting messages.

5. *channel security* – The channel provides identity verification and implements encrypted and authenticated transport resisting man in the middle attacks.

6. *protocol multiplexing* – The underlay network service can accommodate several protocols running on the same connection. Peers communicate the protocols with the name and versions that they implement and the underlay service identifies compatible protocols and starts up peer connections on each matched protocol.

7. *delivery guarantees* – Protocol messages have guaranteed delivery, i.e. delivery failures due to network problems result in direct error response. Order of delivery of messages within each protocol is guaranteed. Ideally the underlay protocol provides prioritisation. If protocol multiplexing is over the same transport channel, this most likely implies framing, so that long messages do not block higher priority messages.

8. *serialisation* – The protocol message construction supports arbitrary data structure serialisation conventions.

The libp2p library can provide all the needed functionality and is the one given in the specification as underlay connectivity driver, see 8.1.[1]

2.1.2 *Overlay addressing*

While clients use the underlay address to establish connections to peers, each node running Swarm is additionally identified with an overlay address. It is this address that determines which peers a node will connect to and directs the way that messages are forwarded. The overlay address is assumed to be stable as it defines a nodes' identity across sessions and ultimately affects which content is most worth storing in the nodes' local storage.

The node's overlay address is derived from an Ethereum account by hashing the corresponding elliptic curve public key with the bzz network ID, using the 256-bit Keccak algorithm (see 7.2). The inclusion of the BZZ network ID stems from the fact that there can be many Swarm networks (e.g. test net, main net, or private Swarms). Including the BZZ network ID makes it impossible to use the same address across networks. Assuming any sample of base accounts independently selected, the resulting overlay addresses are expected to have a uniform distribution in the address space of 256-bit integers. This is important to deriving the address from a public key as

1 Swarm's current golang implementation uses Ethereum's devp2p/rlpx which satisfies the above criteria and uses TCP/IP with custom cryptography added for security. The underlay network address devp2p uses is represented using the enode URL scheme. Devp2p dispatches protocol messages based on their message ID. It uses RLP serialisation which is extended with higher level data type representation conventions. In order to provide support for Ethereum 1.x blockchain and state on Swarm, we may provide a thin devp2p node that proxies queries to a libp2p-based Swarm client or just uses its API. Otherwise we expect the devp2p networking support to be discontinued.

it allows the nodes to issue commitments associated with an overlay location using cryptographic signatures which are verifiable by 3rd parties.

Using the long-lived communication channels of the underlay network, Swarm nodes form a network with *quasi-permanent* peer connections. The resulting connectivity graph can then realise a particular topology defined over the address space. The overlay topology chosen is called Kademlia: It enables communication between any two arbitrary nodes in the Swarm network by providing a strategy to relay messages using only underlay peer connections. The protocol that describes how nodes share information with each other about themselves and other peers, called 'Hive' is described in 8.3. How nodes use this protocol to bootstrap the overlay topology is discussed in 2.1.4. The theoretical basis for Kademlia topology is formalised rigorously in 111.

Crucially, the overlay address space is all 256-bit integers. Central to Swarm is the concept of proximity order (PO), which quantifies the relatedness of two addresses on a discrete scale.[2] Given two addresses, x and y, $PO(x, y)$ counts the matching bits of their binary representation starting from the most significant bit up to the first one that differs. The highest proximity order is therefore 256, designating the maximum relatedness, i.e. where $x = y$.

2.1.3 *Kademlia routing*

Kademlia topology can be used to route messages between nodes in a network using overlay addressing. It has excellent scalability as it allows for a universal routing such that both (1) the number of hops and (2) the number of peer connections are always logarithmic to the size of the network.

In what follows, we show the two common flavours of routing: *iterative/zooming* and *recursive/forwarding*. Swarm's design crucially relies on choosing the latter, the forwarding flavour. However, this is unusual, and, as the iterative flavour predominates within much of the peer-to-peer literature and most other implementations are using iterative routing (see [Maymounkov and Mazieres, 2002, Baumgart and Mies, 2007, Lua et al., 2005]), we consider it useful to walk the reader through both approaches so their idiosyncrasies may be revealed.

Iterative and forwarding Kademlia

Let R be an arbitrary binary relation over nodes in a network. Nodes that are in relation R with a particular node x are called peers of x. Peers of x can be indexed by their proximity order (PO) relative to x (see A.1). The equivalence classes of peers are called proximity order bins, or just bins for short. Once arranged in bins, these groups of peers form the Kademlia table of the node x (see figure 2).

2 Proximity order is the discrete logarithmic scale of proximity, which, in turn is the inverse of normalised XOR distance. See A.1 for a formal definition.

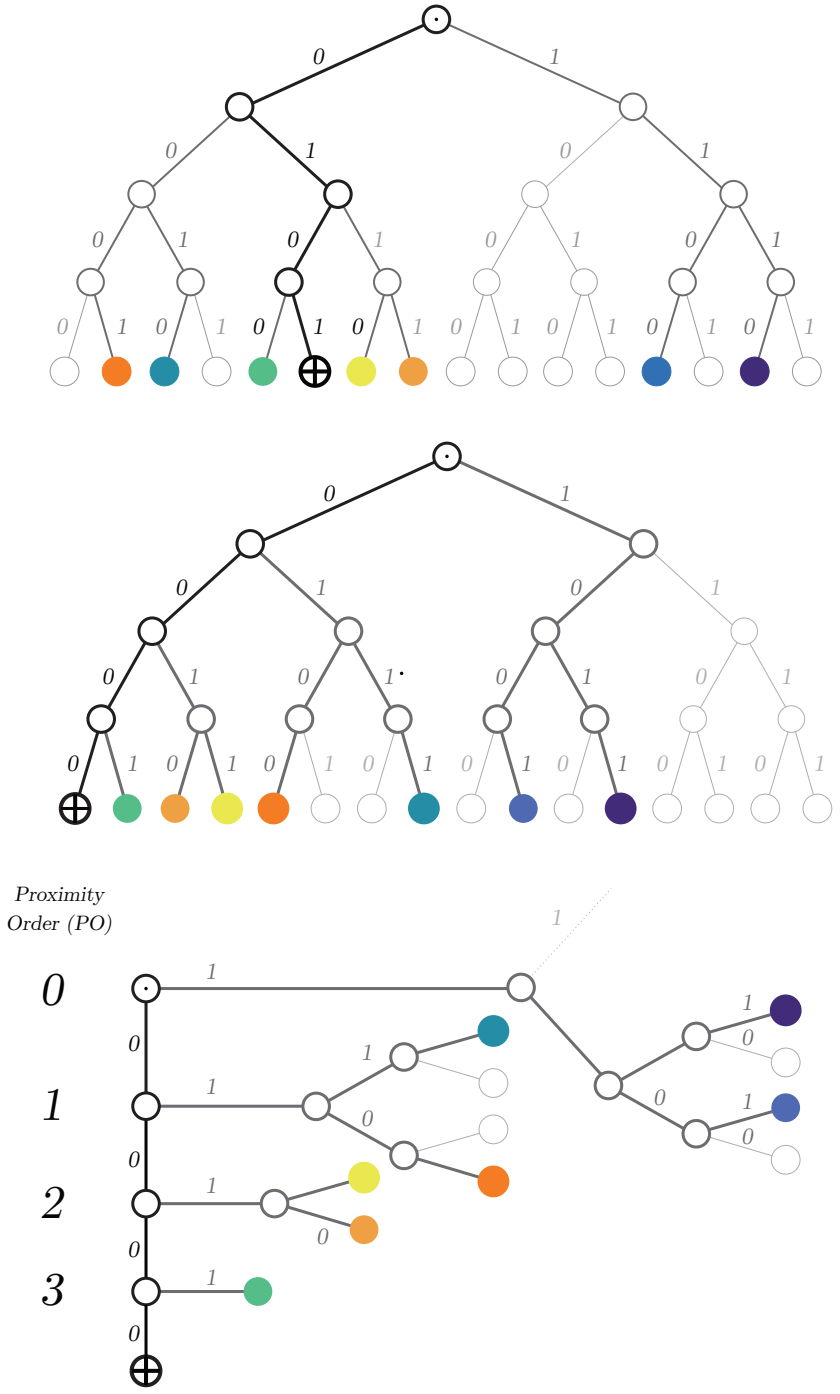


Figure 2: From overlay address space to Kademlia table. **Top**: the overlay address space is represented with a binary tree, colored leaves are actual nodes. The path of the pivot node (+) is shown with thicker lines. **Centre**: peers of the pivot nodes are shown keyed by the bits of their xor distance measured from the pivot. Here zeros represent a matching bit with the pivot, ones show a differing bit. The leaf nodes are ordered by their xor distance from the pivot (leftmost node). **Bottom**: the Kademlia table of the pivot: the subtrees branching off from the pivot path on the left are displayed as the rows of the table representing proximity order bins in increasing order.

Node x has a saturated Kademlia table if there is a $0 \leq d_x \leq maxPO$ called the neighbourhood depth such that (1) the node has at least one peer in each bin up to and excluding proximity order bin d_x and (2) all nodes at least as near as d_x (called the nearest neighbours) are peers of x. If each node in a network has a saturated Kademlia table, then we say that the network has Kademlia topology.

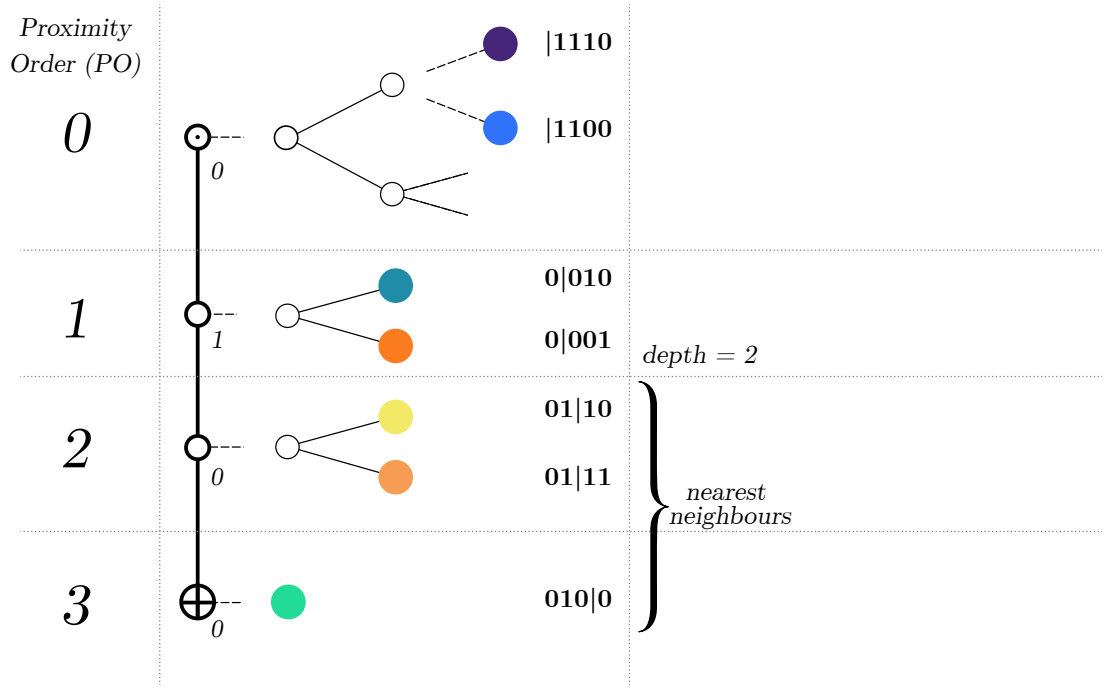


Figure 3: Nearest neighbours in a 4 bit network with $d = 2$

Let R be the "is known to" relation: y "is known to" x if x has both overlay and underlay addressing information for y. In the iterative Kademlia routing the requestor node iteratively extends the graph of peers that are known to it. Using their underlay address, the requestor node will contact the peers that they know are nearest the destination address for peers that are further away (commonly using UDP), on each successive iteration the peers become at least one order closer to the destination (see figure 4). Because of the Kademlia criteria, the requestor will end up eventually discovering the destination node's underlay address and can then establish direct communication with it. This iterative strategy[3] critically depends on the nodes' ability to find peers that are currently online. In order to find such a peer, a node needs to collect several candidates for each bin. The best predictor of availability is the recency of the peer's last response, so peers in a bin should be prioritised according to this ordering.

Swarm uses an alternative flavour of Kademlia routing which is first described in [Heep, 2010] and then expanded on and worked out in [Tron et al., 2019b]. Here,

3 The iterative protocol is equivalent to the original Kademlia routing that is described in [Maymounkov and Mazieres, 2002].

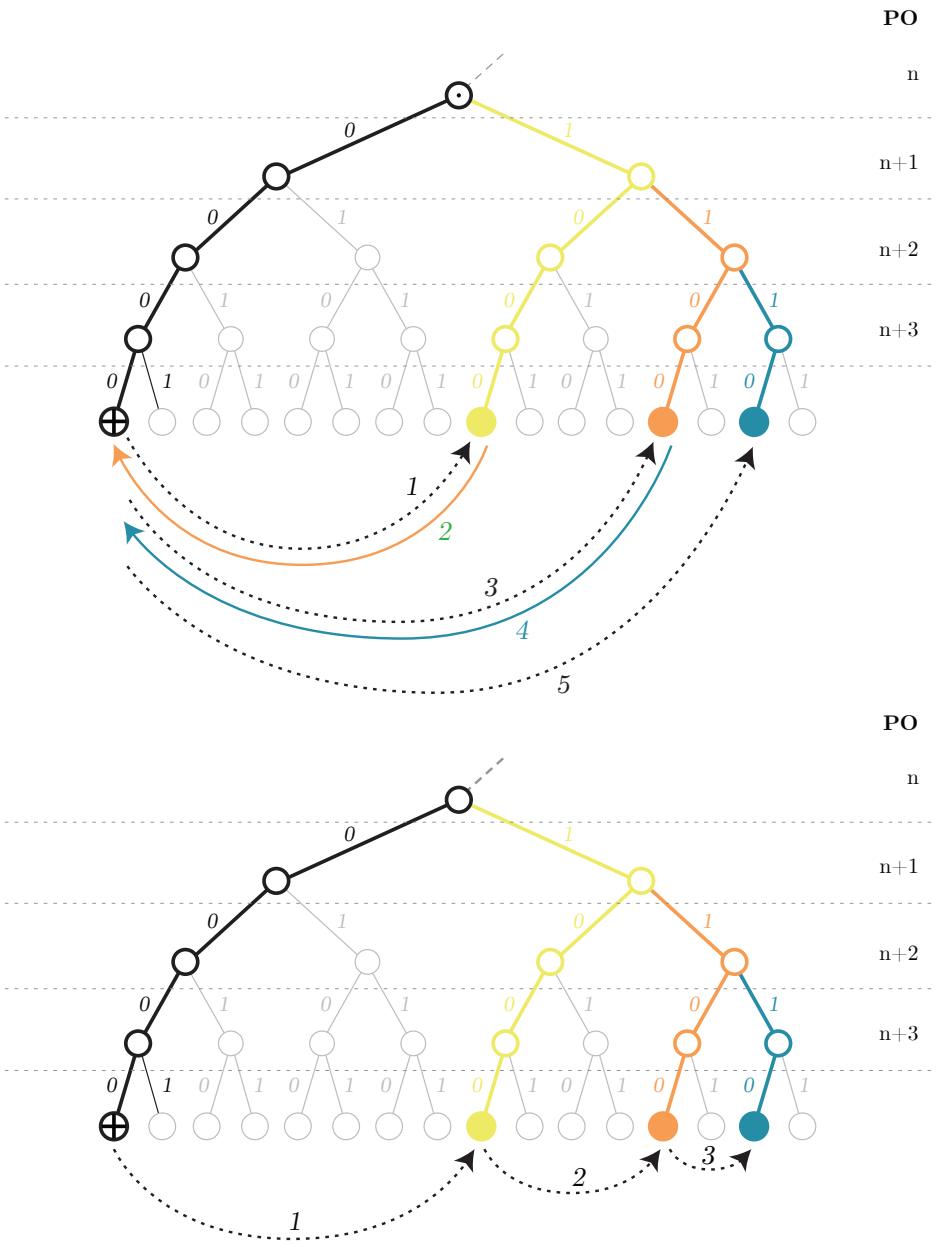


Figure 4: Iterative and Forwarding Kademlia routing: A requestor node shown with a cross in the circle at address ...0000... wants to route to a destination address ...1111... to which the closest peer online is the blue circle at ...1110... These initial ellipses represent the prefix shared by requestor and destination addresses which is n bits long. **Top:** In the iterative flavour, the requestor contacts the peers (step 1, dotted black arrows) that they know are nearest the destination address. Peers that are online (yellow) respond with information about nodes that are even closer (green arrow, step 2) so the requestor can now repeat the query using these closer peers (green, step 3). On each successive iteration the peers (yellow, green and blue) are at least one PO closer to the destination until eventually the requestor is in direct contact with the node that is nearest to the destination address. **Bottom:** In the forwarding flavour, the requestor forwards a message to the connected peer they know that is nearest to the destination (yellow). The recipient peer does the same. Applying this strategy recursively relays the message via a chain of peers (yellow, green, blue) each at least one PO closer to the destination.

a recursive method is employed, whereby the successive steps of the iteration are "outsourced" to a downstream peer. Each node recursively passes a message to a direct peer at least one proximity order closer to the destination. Routing here then means simply relaying messages via a chain of peers which are ever closer to the destination, as shown in figure 4.

In this way, Swarm's underlay transport offers quasi-stable peer connections over TCP with communication channels that are kept alive. These open connections can then be used as R to define another notion of a peer. The two criteria of healthy Kademlia connectivity in the swarm translate as: For each node x, there exists a neighbourhood depth d_x such that (1) node x has an open connection with at least one node for each proximity order bin up to but excluding d_x and (2) is connected to all the online nodes that are at least as near as d_x. If each node in the network has a saturated Kademlia table of peers, then the network is said to have Kademlia topology. Since connected peers are guaranteed to be online, the recursive step consists solely of forwarding the message to a connected peer strictly closer to the destination. We can call this alternative a forwarding Kademlia.

In a forwarding Kademlia network, a message is said to be *routable* if there exists a path from sender to destination through which the message can be relayed. In a mature subnetwork with Kademlia topology every message is routable.

If all peer connections are stably online, a thin Kademlia table, i.e. a single peer for each bin up to d, is sufficient to guarantee routing between nodes. In reality, however, networks are subject to churn, i.e. nodes may be expected to go offline regularly. In order to ensure routability in the face of churn, the network needs to maintain Kademlia topology. This means that each individual node needs to have a saturated Kademlia table at all times. By keeping several connected peers in each proximity order bin, a node can ensure that node dropouts do not damage the saturation of their Kademlia table. Given a model of node dropouts, we can calculate the minimum number of peers needed per bin to guarantee that nodes are saturated with a probability that is arbitrarily close to 1. The more peers a node keeps in a particular proximity order bin, the more likely that the message destination address and the peer will have a longer matching prefix. As a consequence of forwarding the message to that peer, the proximity order increases more quickly, and the message ends up closer to the destination than it would with less peers in each bin (see also figure 5).

With Kademlia saturation guaranteed, a node will always be able to forward a message and ensure routability. If nodes comply with the forwarding principles (and that is ensured by aligned incentives, see 9.2) the only case when relaying could possibly break down is when a node drops out of the network after having received a message but before it has had chance to forward it.[4]

4 Healthy nodes could commit to being able to forward within a (very short) constant time we can call the forwarding lag. In the case that a downstream peer disconnects before this forwarding lag passes, then the upstream peer can re-forward the message to an alternative peer, thereby keeping the message passing unbroken. See 2.3.1 for more detail.

An important advantage of forwarding Kademlia is that this method of routing requires a lot less bandwidth than the iterative algorithm. In the iterative version, known peers are not guaranteed to be online, so finding one that is adds an additional level of unpredictability.

Sender anonymity

Sender anonymity is a crucial feature of Swarm. It is important that since requests are relayed from peer-to-peer, those peers further down in the request cascade can never know who the originator of the request was.

The above rigid formulation of Kademlia routing would suggest that if a node receives a message from a peer and that message and peer have a proximity order of 0, then the recipient would be able to conclude that the peer it received the message from must be the sender. If we allow light node Swarm clients who, since they are running in a low-resource environment, do not keep a full Kademlia saturation but instead have just a local neighbourhood, even a message from a peer in bin 0 remains of ambiguous origin.

Bin density and multi-order hops

As a consequence of logarithmic distance and uniform node distribution, farther peers of a particular node are exponentially more numerous. As long as the number of connected peers is less than double the number in the bin one closer, then shallower bins will always allow more choice for nodes. In particular, nodes have a chance to increase the number of connections per bin in such a way that peer addresses maximise density (in proximity order bin b, the subsequent bits of peers addresses form a balanced binary tree). Such an arrangement is optimal in the sense that for a bin depth of d, nodes are able to relay all messages so that in one hop, the proximity order of the destination address will increase by d (see figure 5).

Factoring in underlay proximity

It is expected that as Swarm clients continue to evolve and develop, nodes may factor in throughput when they select peers for connection. All things being equal, nodes in geographic physical proximity tend to have higher throughput, and therefore will be preferred in the long run. This is how forwarding Kademlia is implicitly aware of underlay topology [Heep, 2010]. See 9.1 for a more detailed discussion of connectivity strategy.

2.1.4 *Bootstrapping and maintaining Kademlia topology*

This section discusses how a stable Kademlia topology can emerge. In particular, the exact bootstrapping protocol each node must follow to reach a saturated Kademlia

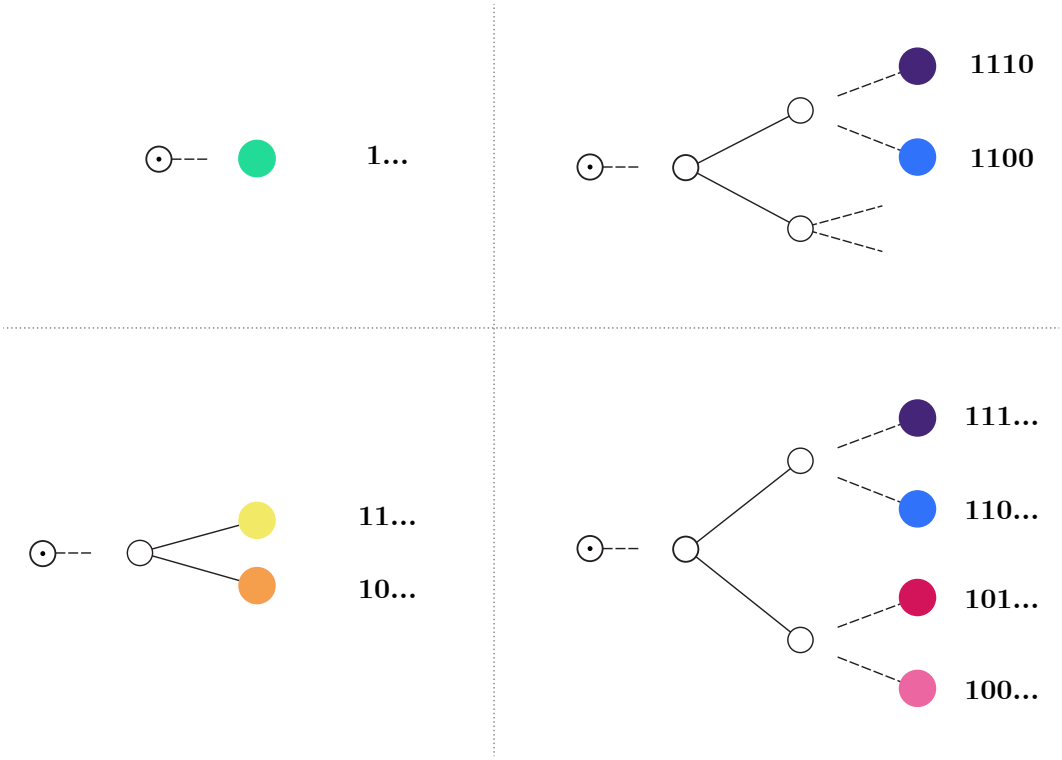


Figure 5: Bin density: types of saturation for PO bin 0 for a node with overlay address starting with bit 0. **Top left**: A "thin" bin with a single peer is not resilient to churn and only increases PO by 1 in one hop. **Top right:** At least two peers are needed to maintain Kademlia topology in case of churn; two peers when not balanced cannot guarantee multi-order hops. **Bottom left:** Two peers balanced guarantees an increase of 2 PO-s in one hop. **Bottom right:** Four peers when balanced can guarantee an increase of 3 PO-s in one hop.

connectivity and maintain it. Nodes joining a decentralised network are supposed to be initially naive, potentially initiating connection via only a single known peer with no prior knowledge. For this reason, the bootstrapping process needs to include a discovery component with the help of which nodes exchange information about each other. This protocol is called the hive protocol and is formally specified in 8.3.

Bootnodes

Swarm has no distinct node type or operation mode for bootnodes. This means that naive nodes should be able to connect to any node on the network and bootstrap their desired connectivity. In order not to overburden any single node, specifying a particular node as an initial connection should be avoided and the role of bootnode with respect to new naive nodes should ideally be balanced between participant nodes. This is achieved either with an invite system or a centralised bootnode service running a public gateway which responds to an API call with the bzz address of a randomly chosen node among online peers.

Once connected to a node in the network, the hive protocol will kick in and the naive node will learn about the bzz addresses of other nodes and can start bootstrapping its connectivity.

Building up connections

Initially, each node begins with zero as their saturation depth. Nodes keep advertising their saturation depth to their connected peers as it changes. When a node A receives an attempt to establish a new connection from a node B, she notifies each of her other peers of the node B connecting to her only in the case that each peer's proximity order relative to the connecting node A is not lower than that peer's advertised saturation depth. The notification is always sent to a peer that shares a proximity order bin with the new connection. Formally, when x connects with a peer y, x notifies each of her peers if $PO(x,p) = PO(x,y)$ or $d_p \leq PO(y,p)$. In particular, notification includes the full overlay address and underlay address information (see 7.2).[5]

Mature connectivity

After a sufficient number of nodes are connected, a bin becomes saturated, and the node's neighbourhood depth can begin to increase. Nodes keep their peers up to date by advertising their current depth if it changes. As their depth increases, nodes will get notified of fewer and fewer peers. Once the node finds all their nearest neighbours and has saturated all the bins, no new peers are to be expected. For this reason, a node

5 Light nodes that do not wish to relay messages and do not aspire to build up a healthy Kademlia, are not included, see section 2.3.4.

can conclude a saturated Kademlia state if it receives no new peers for some time.[6] Instead of having a hard deadline and a binary state of saturation, we can quantify the certainty of saturation by the age of the last new peer received. Assuming stable connections, eventually each node online will get to know its nearest neighbours and connect to them while keeping each bin up to d non-empty. Therefore each node will converge on the saturated state. If no new nodes join, health (Kademlia topology) is maintained even if peer connections change. A node is not supposed to go back to a lower saturation state for instance. This is achieved by requiring several peers in each proximity order bin.

2.2 SWARM STORAGE

In this section, we first show how a network with quasi-permanent connections in a Kademlia topology can support a load balancing, distributed storage of fixed-sized datablobs in 2.2.1. In 2.2.1, we detail the generic requirements on chunks and introduce actual chunk types. Finally, in 2.2.5, we turn to redundancy by neighbourhood replication as a fundamental measure for churn-resistance.

2.2.1 *Distributed immutable store for chunks*

In this section we discuss how networks using Kademlia overlay routing are a suitable basis on which to implement serverless storage solution using distributed hash tables (DHTs). Then we introduce the DISC[7] model, Swarm's narrower interpretation of a DHT for storage. This model imposes some requirements on chunks and necessitates 'upload' protocols.

As is customary in Swarm, we provide a few resolutions of this acronym, which summarise the most important features:

- *decentralised infrastructure for storage and communication*,
- *distributed immutable store for chunks*,
- *data integrity by signature or content address*,
- *driven by incentives with smart contracts*.

6 Although 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: the depth of network is $\log_2(n+1)+d$ where n is the number of remote peers in the nearest neighbourhood and d is the depth of that neighbourhood. It then follows that the total number of nodes in the network can be estimated simply by taking this the exponent of 2 .

7 DISC is distributed immutable store for chunks. In earlier work, we have referred to this component as the 'distributed preimage archive' (DPA), however, this phrase became misleading since we now also allow chunks that are not the preimage of their address.

From DHT to DISC

Swarm's DISC shares many similarities with the more familiar distributed hash tables. The most important difference is that Swarm does not keep a list of *where* files are to be found, instead it actually *stores pieces of the file itself* directly with the closest node(s). In what follows, we review DHTs, as well as dive into the similarities and differences with DISC in more detail.

Distributed hash tables use an overlay network to implement a key–value container distributed over the nodes (see figure 6). The basic idea is that the keyspace is mapped on to the overlay address space, and the value for a key in the container is to be found with nodes whose addresses are in the proximity of the key. In the simplest case, let us say that this is the single closest node to the key that stores the value. In a network with Kademlia connectivity, any node can route to a node whose address is closest to the key, therefore a *lookup* (i.e. looking up the value belonging to a key) is reduced simply to routing a request.

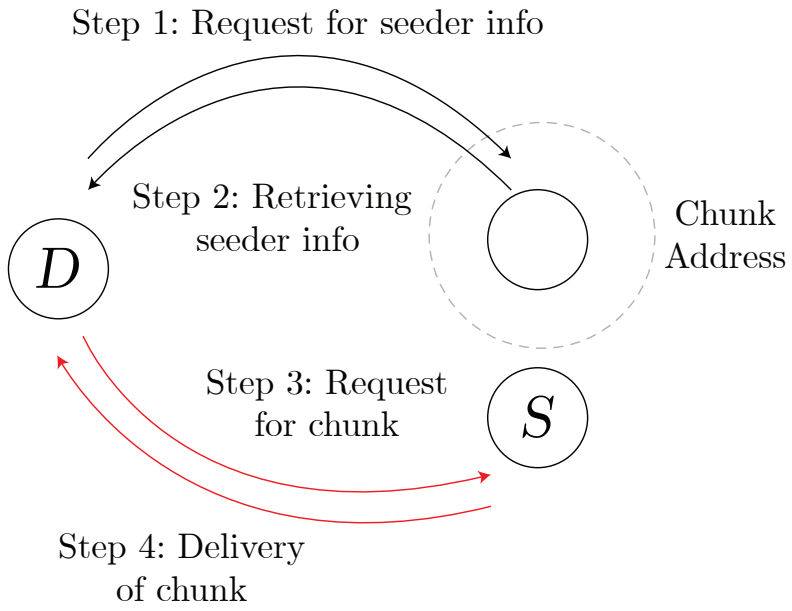


Figure 6: Distributed hash tables (DHTs) used for storage: node *D* (downloader) uses Kademlia routing in step 1 to query nodes in the neighbourhood of the chunk address to retrieve seeder info in step 2. The seeder info is used to contact node *S* (seeder) directly to request the chunk and deliver it in steps 3 and 4.

DHTs used for distributed storage typically associate content identifiers (as keys/addresses) with a changing list of seeders (as values) that can serve that content [IPFS, 2014, Crosby and Wallach, 2007]. However, the same structure can be used directly: in Swarm, it is not information about the location of content that is stored at the swarm node closest to the address, but the content itself (see figure 7).

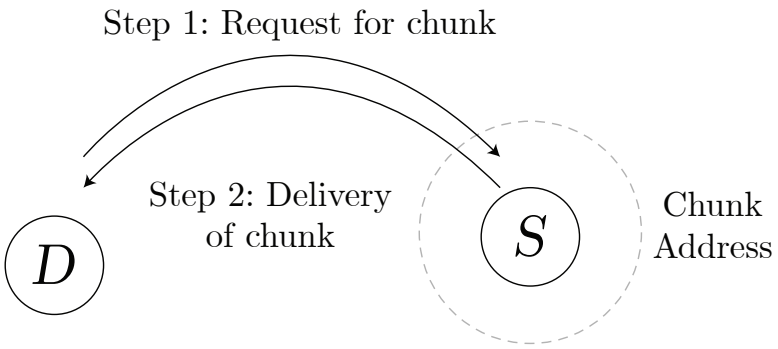


Figure 7: Swarm DISC: Distributed Immutable Store for Chunks. In step 1, downloader node *D* uses forwarding Kademlia routing to request the chunk from a storer node *S* in the neighbourhood of the chunk address. In step 2 the chunk is delivered along the same route using forwarding step just backwards.

Constraints

The DISC storage model is opinionated about which nodes store what content and this implies the following restrictions:

1. *fixed-size chunks* – Load balancing of content is required among nodes and is realised by splitting content into equal sized units called chunks (see 2.2.1).
2. *syncing* – There must be a process whereby chunks get to where they are supposed to be stored no matter which node uploads them (see 2.3.2).
3. *plausible deniability* – Since nodes do not have a say in what they store, measures should be employed which serve as the basis of legal protection for node operators to plausibly deny knowing (or even being able to know) anything about the chunks' contents (see 2.2.4).
4. *garbage collection* – Since nodes commit to store anything close to them, there needs to be a strategy to select which chunks are kept and which are discarded in the presence of storage space constraints (see 9.7).

Chunks

Chunks are the basic storage units used in Swarm's network layer. They are an association of an address with content. Since retrieval in Swarm (2.3.1) assumes that chunks are stored with nodes close to their address, the addresses of chunks should also be uniformly distributed in the address space and have their content limited and roughly uniform in size to achieve fair and equal load balancing.

When chunks are retrieved, the downloader must be able to verify the correctness of the content given the address. Such integrity translates to guaranteeing uniqueness

of content associated with an address. In order to protect against frivolous network traffic, third party forwarding nodes should be able to verify the integrity of chunks using only local information available to the node.

The deterministic and collision free nature of addressing implies that chunks are unique as a key–value association: If there exists a chunk with an address, then no other valid chunk can have the same address; this assumption is crucial as it makes the chunk store immutable, i.e. there is no replace/update operation on chunks. Immutability is beneficial in the context of relaying chunks as nodes can negotiate information about the possession of chunks simply by checking their addresses. This plays an important role in the stream protocol (see 2.3.3) and justifies the DISC resolution as *distributed immutable store for chunks*.

To sum up, chunk addressing needs to fulfill the following requirements:

1. *deterministic* – To enable local validation.
2. *collision free* – To provide integrity guarantee.
3. *uniformly distributed* – To deliver load balancing.

In the current version of Swarm, we support two types of chunk: content addressed chunks and single owner chunks.

2.2.2 *Content addressed chunks*

A content addressed chunk is not assumed to be a meaningful storage unit, i.e. they can be just blobs of arbitrary data resulting from splitting a larger data blob, a file. The methods by which files are disassembled into chunks when uploading and then reassembled from chunks when downloading are detailed in 4.1. The data size of a content addressed Swarm chunk is limited to 4 kilobytes. One of the desirable consequences of using this small chunk size is that concurrent retrieval is available even for relatively small files, reducing the latency of downloads.

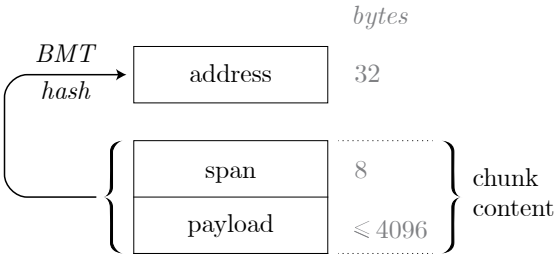


Figure 8: Content addressed chunk. An at most 4KB payload with a 64-bit little endian encoded span prepended to it constitutes the chunk content. The BMT hash of the payload concatenated with the span then yields the content address.

Binary Merkle tree hash

The canonical content addressed chunk in Swarm is called a binary Merkle tree chunk (BMT chunk). The address of BMT chunks is calculated using the binary Merkle tree hash algorithm (BMT hash) described in 7.3.3. The base hash used in BMT is Keccak256, properties of which such as uniformity, irreversibility and collision resistance all carry over to the BMT hash algorithm. As a result of uniformity, a random set of chunked content will generate addresses evenly spread in the address space, i.e. imposing storage requirements balanced among nodes.

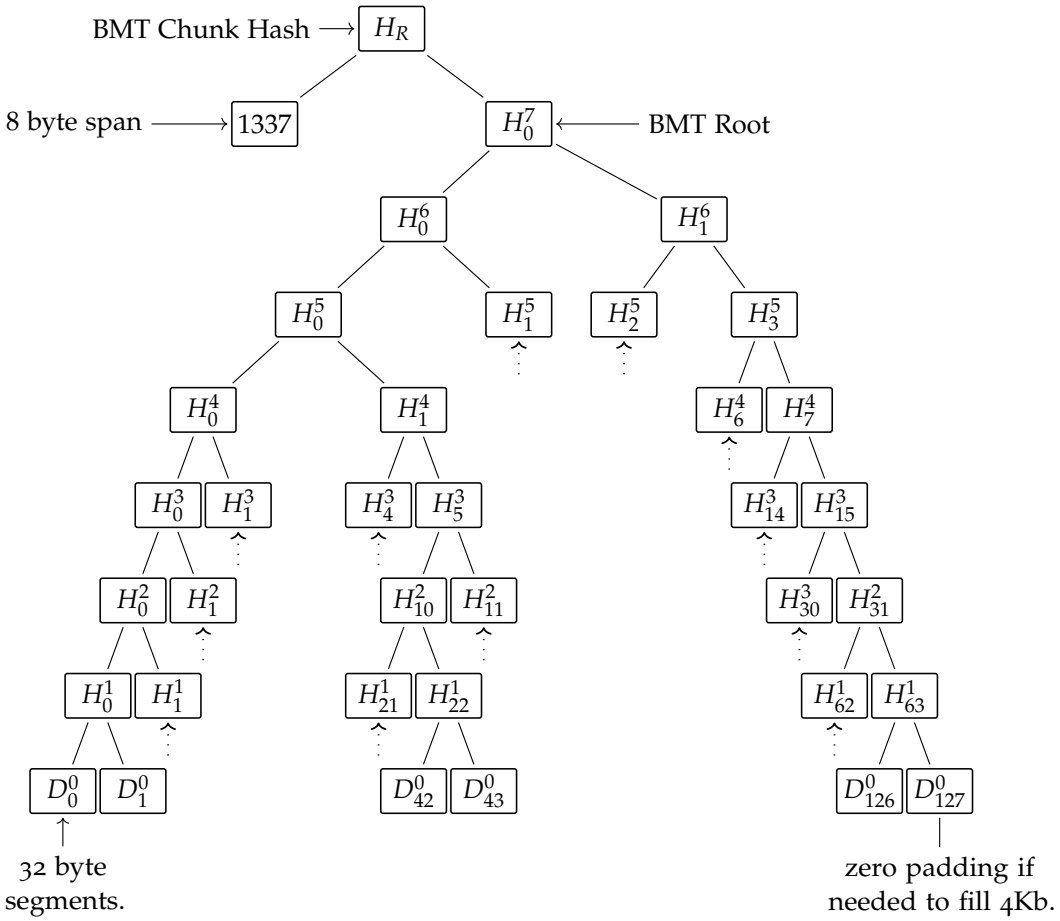


Figure 9: BMT (Binary Merkle Tree) chunk hash in Swarm: the 1337 bytes of chunk data is segmented into 32 byte segments. Zero padding is used to fill up the rest up to 4 kilobytes. Pairs of segments are hashed together using Keccak256 to build up the binary tree. On level 8, the binary Merkle root is prepended with the 8 byte span and hashed to yield the BMT chunk hash.'

The BMT chunk address is the hash of the 8 byte span and the root hash of a binary Merkle tree (BMT) built on the 32-byte segments of the underlying data (see figure 9). If the chunk content is less than 4k, the hash is calculated as if the chunk was padded with all zeros up to 4096 bytes.

This structure allows for compact inclusion proofs with a 32-byte resolution. An inclusion proof is a proof that one string is a substring of another string, for instance, that a string is included in a chunk. Inclusion proofs are defined on a data segment of a particular index, see figure 10. Such Merkle proofs are also used as proof of custody when storer nodes provide evidence that they possess a chunk (see 3.3.2). Together with the Swarm file hash (see 4.1.1 and 7.4.1), it allows for logarithmic inclusion proofs for files, i.e. proof that a string is found to be part of a file.

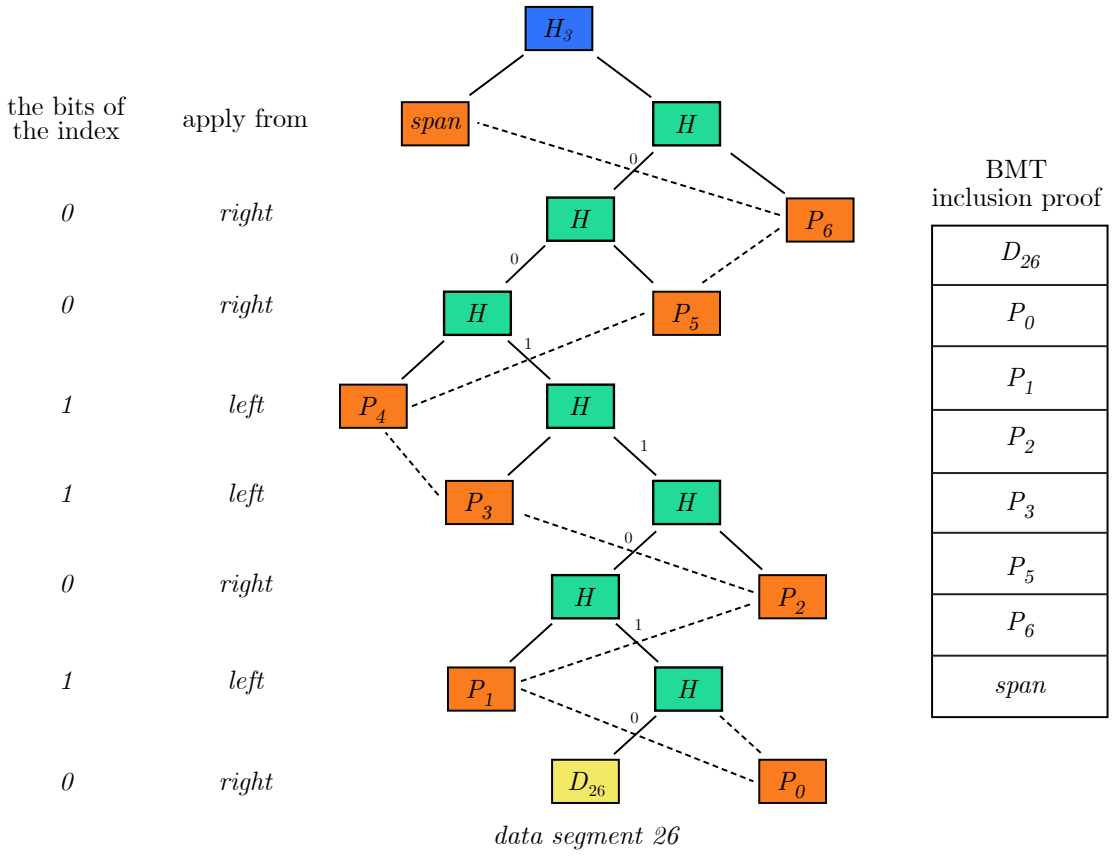


Figure 10: Compact segment inclusion proofs for chunks. Assume we need proof for segment 26 of a chunk (yellow). The orange hashes of the BMT are the sister nodes on the path from the data segment up to the root and constitute what needs to be part of a proof. When these are provided together with the root hash and the segment index, the proof can be verified. The side on which proof item i needs to be applied depends on the i-th bit (starting from least significant) of the binary representation of the index. Finally the span is prepended and the resulting hash should match the chunk root hash.

2.2.3 *Single-owner chunks*

With single owner chunks, a user can assign arbitrary data to an address and attest chunk integrity with their digital signature. The address is calculated as the hash of an identifier and an owner. The chunk content is presented in a structure composed of the identifier, the payload and a signature attesting to the association of identifier and payload (see figure 11).

- *content*:
  - *identifier* – 32 bytes arbitrary identifier,
  - *signature* – 65 bytes $\langle r, s, v \rangle$ representation of an EC signature (32+32+1 bytes),
  - *span* – 8 byte little endian binary of uint64 chunk span,
  - *payload* – max 4096 bytes of regular chunk data.
- *address* – Keccak256 hash of identifier + owner account.

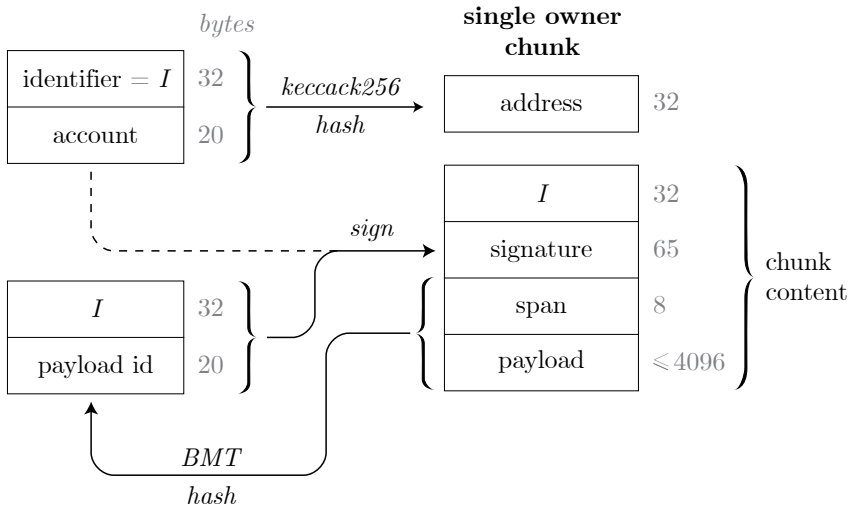


Figure 11: Single-owner chunk. The chunk content is composed of headers followed by an at most 4KB payload. The last header field is the 8 byte span prepended just like in content addressed chunks. The first two header fields provide single owner attestation of integrity: an identifier and a signature signing off on the identifier and the BMT hash of span and payload. The address is the hash of the id and the signer account.

Validity of a single owner chunk is checked with the following process:

1. Deserialise the chunk content into fields for identifier, signature and payload.

2. Construct the expected plain text composed of the identifier and the BMT hash of the payload.

3. Recover the owner's address from the signature using the plain text.

4. Check the hash of the identifier and the owner (expected address) against the chunk address.

Single-owner chunks offer a virtual partitioning of part of the address space into subspaces associated with the single owner. Checking their validity is actually an authentication verifying that the owner has write access to the address with the correct identifier.

As suggested by the span and the length of the payload, a single owner chunk can encapsulate a regular content addressed chunk. Anyone can simply reassign a regular chunk to an address in their subspace designated by the identifier (see also 4.4.4).

It should be noted that the notion of integrity is somewhat weaker for single owner chunks than in the case of content addressed chunks: After all, it is, in principle, possible to assign and sign any payload to an identifier. Nonetheless, given the fact that the chunk can only be created by a single owner (of the private key that the signature requires), it is reasonable to expect uniqueness guarantees because we hope the node will want to comply with application protocols to get the desired result. However, if the owner of the private key signs two different payloads with the same identifier and uploads both chunks to Swarm, the behaviour of the network is unpredictable. Measures can be taken to mitigate this in layer (3) and are discussed later in detail in 4.3.3.

With two types of chunk, integrity is linked to collision free hash digests, derived from either a single owner and an arbitrary identifier attested by a signature or directly from the content. This justifies the resolution of the DISC acronym as *data integrity through signing or content address*.

2.2.4 *Chunk encryption*

Chunks should be encrypted by default. Beyond client needs for confidentiality, encryption has two further important roles. (1) Obfuscation of chunk content by encryption provides a degree of plausible deniability; using it across the board makes this defense stronger. (2) The ability to choose arbitrary encryption keys together with the property of uniform distribution offer predictable ways of mining chunks, i.e. generating an encrypted variant of the same content so that the resulting chunk address satisfies certain constraints, e.g. is closer to or farther away from a particular address. This is an important property used in (1) price arbitrage (see 3.1.2) and (2) efficient use of postage stamps (see 3.3.1).

Chunk encryption (see figure 12) is formally specified in 7.3.4. Chunks shorter than 4 kilobytes are padded with random bytes (generated from the chunk encryption

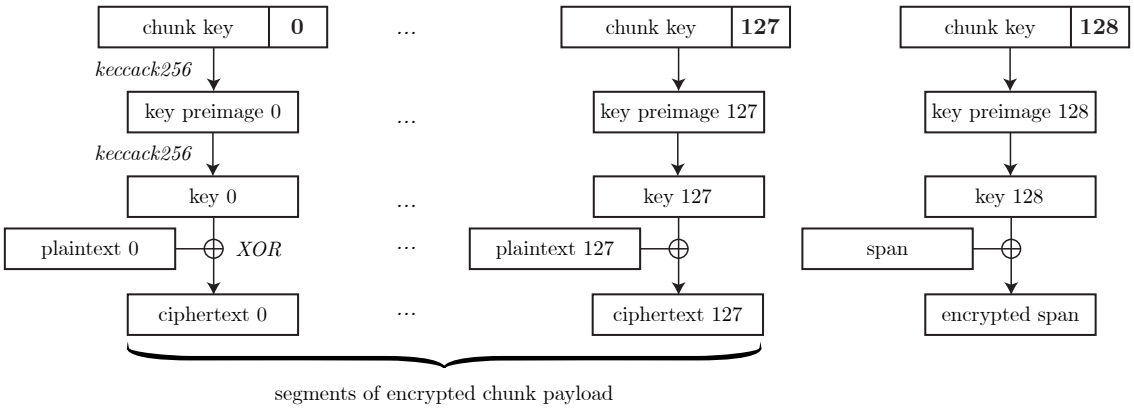


Figure 12: Chunk encryption in Swarm. Symmetric encryption with a modified counter-mode block cipher. The plaintext input is the content padded with random bytes to 4 kilobytes. The span bytes are also encrypted as if they were continuations of the payload.

seed). The full chunk plaintext is encrypted and decrypted using a XOR-based block cipher seeded with the corresponding symmetric key. In order not to increase the attack surface by introducing additional cryptographic primitives, the block cipher of choice is using Keccak256 in counter mode, i.e. hashing together the key with a counter for each consecutive segment of 32 bytes. In order to allow selective disclosure of individual segments being part of an encrypted file, yet leak no information about the rest of the file, we add an additional step of hashing to derive the encryption key for a segment in the chunk. This scheme is easy and cheap to implement in the (EVM), lending itself to use in smart contracts constraining the plaintext of encrypted Swarm content.

The prepended metadata encoding the chunk span is also encrypted as if it was a continuation of the chunk, i.e. with counter 128. Encrypted chunk content is hashed using the BMT hash digest just as unencrypted ones are. The fact that a chunk is encrypted may be guessed from the span value, but apart from this, in the network layer, encrypted chunks behave in exactly the same way as unencrypted ones.

Single owner chunks can also be encrypted, which simply means that they wrap an encrypted regular chunk. Therefore, their payload and span reflect the chunk content encryption described above, the hash signed with the identifier is the BMT hash of the encrypted span and payload, i.e. the same as that of the wrapped chunk.

2.2.5 *Redundancy by replication*

It is important to have a resilient means of requesting data. To achieve this Swarm implements the approach of defence in depth. In the case that a request fails due to a

problem with forwarding, one can retry with another peer or, to guard against these occurrences, a node can start concurrent retrieve requests right away. However, such fallback options are not available if all the single last node that stores the chunk drop out from the network. Therefore, redundancy is of major importance to ensure data availability. If the closest node is the only storer of the requested data and it drops out of the network, then there is no way to retrieve the content. This basic scenario is handled by ensuring each set of nearest neighbours hold replicas of each chunk that is closest to any one of them, duplicating the storage of chunks and therefore providing data redundancy.

Size of nearest neighbourhoods

If the Kademlia connectivity is defined over storer nodes, then in a network with Kademlia topology there exists a depth d such that (1) each proximity order bin less than d contains at least k storer peers, and (2) all storer nodes with proximity order d or higher are actually connected peers. In order to ensure data redundancy, we can add to this definition a criterion that (3) the nearest neighbourhood defined by d must contain at least r peers.

Let us define neighbourhood size $NHS_x(d)$ as the cardinality of the neighbourhood defined by depth d of node x. Then, a node has Kademlia connectivity with redundancy factor r if there exists a depth d such that (1) each proximity order bin less than d contains at least k storer peers (k is the bin density parameter see 2.1.3), and (2) all storer nodes with proximity order d or higher are actually connected peers, and (3) $NHS_x(d) \geq r$.

We can then take the highest depth d' such that (1) and (2) are satisfied. Such a d is guaranteed to exist and the Hive protocol is always able to bootstrap it. As we decrease d', the amount of different neighbourhood grow proportionally, so for any redundancy parameter not greater than the network size $r \leq N = NHS_x(0)$, there will be a highest $0 < d_r \leq d'$ such that $NHS_x(d_r) \geq r$. Therefore, redundant Kademlia connectivity is always achievable.

For a particular redundancy, the area of the fully connected neighbourhood defines an area of responsibility. The proximity order boundary of the area of responsibility defines a radius of responsibility for the node. A storer node is said to be *responsible* for (storing) a chunk if the chunk address falls within the node's radius of responsibility.

It is already instructive at this point to show neighbourhoods and how they are structured, see figure 13.

Redundant retrievability

A chunk is said to have redundant retrievability with degree r if it is retrievable and would remain so even after any r nodes responsible for it leave the network. The naive approach presented so far requiring the single closest node to keep the content can be interpreted as degree zero retrievability. If nodes in their area of responsibility fully

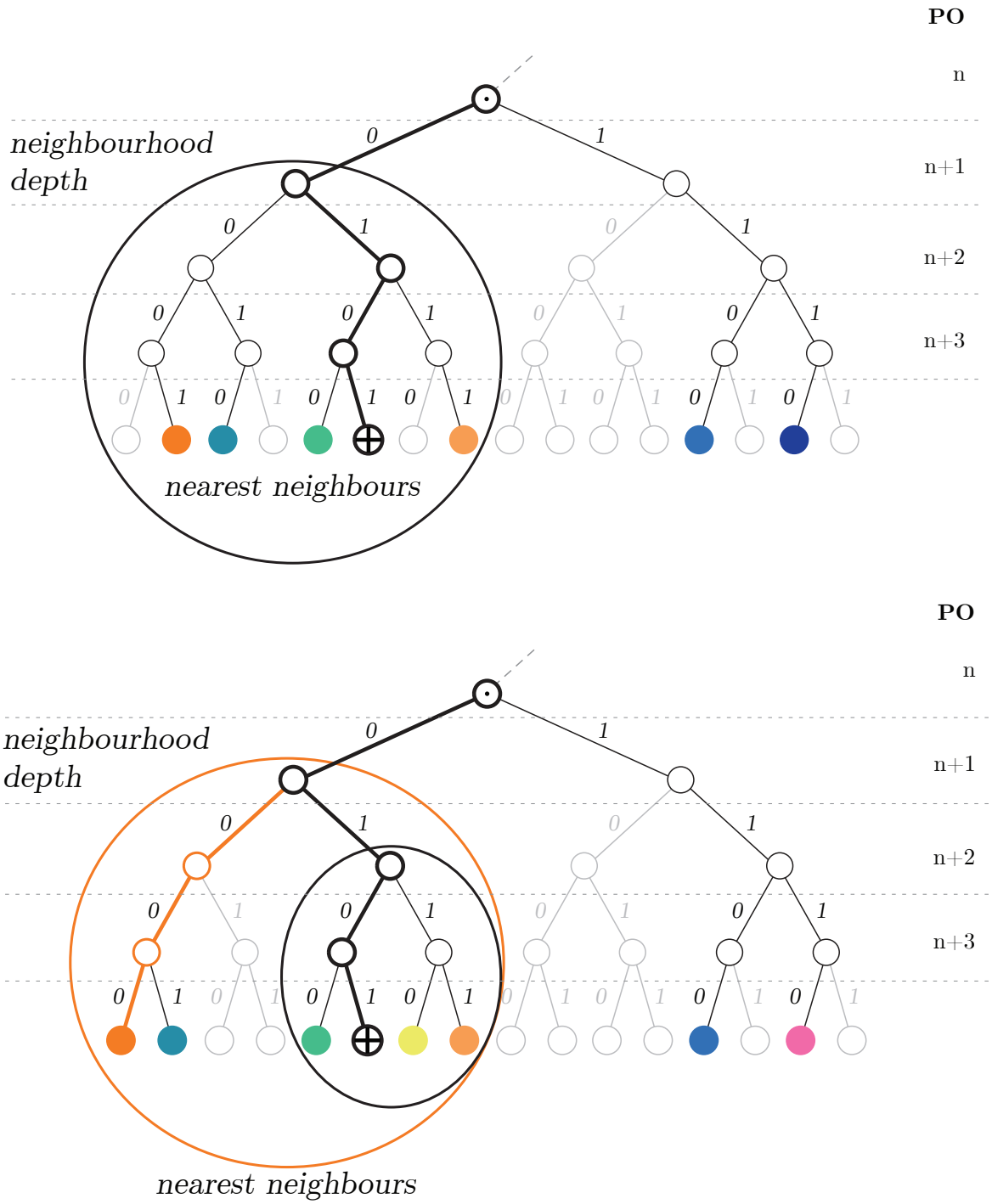


Figure 13: Nearest neighbours. **Top**: Each PO defines a neighbourhood, the neighbourhood depth of the node (black circle) is defined as the highest PO such that the neighbourhood has at least R=4 peers (redundancy parameter) and all shallower bins are non-empty. **Bottom**: An asymmetric neighbourhood. Nearest neighbours of the orange node include the black node but not the other way round.

replicate their content (see 2.3.3), then every chunk in the Swarm DISC is redundantly retrievable with degree r. Let us take the node x that is closest to a chunk c. Since it has Kademlia connectivity with redundancy r, there are $r+1$ nodes responsible for the chunk in a neighbourhood fully connected and replicating content. After r responsible nodes drop out, there is just one node remaining which still has the chunk. However, if Kademlia connectivity is maintained as the r nodes leave, this node will continue to be accessible by any other node in the network and therefore the chunk is still retrievable. Now, for the network to ensure all chunks will remain redundantly retrievable with degree r, the nodes comprising the new neighbourhood formed due to the reorganising of the network must respond by re-syncing their content to satisfy the protocol's replication criteria. This is called the guarantee of eventual consistency.

Resource constraints

Let us assume then that (1) the forwarding strategy that relays requests along stable nodes and (2) the storage strategy that each node in the nearest neighbourhood (of r storer nodes) stores all chunks the address of which fall within their radius of responsibility. As long as these assumptions hold, each chunk is retrievable even if r 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. Realistically, however, all nodes have resource limitations. With time, the overall amount of distinct chunks ever uploaded to Swarm will increase indefinitely. Unless the total storage capacity steadily increases, we should expect that the nodes in Swarm are able to store only a subset of chunks. Some nodes will reach the limit of their storage capacity and therefore face the decision whether to stop accepting new chunks via syncing or to make space by deleting some of their existing chunks.

The process that purges chunks from their local storage is called garbage collection. The process that dictates which chunks are chosen for garbage collection is called the garbage collection strategy (see 9.7). For a profit-maximizing node, it holds that it is always best to garbage-collect the chunks that are predicted to be the least profitable in the future and, in order to maximize profit, it is desired for a node to get this prediction right (see 3.3.1). So, in order to factor in these capacity constraints, we will introduce the notion of chunk value and modify our definitions using the minimum value constraint:

In Swarm's DISC, at all times there is a chunk value v such that every chunk with a value greater than v is both retrievable and eventually (after syncing) redundantly retrievable with degree r.

This value ideally corresponds to the relative importance of preserving the chunk that uploaders need to indicate. In order for storer nodes to respect it, the value should also align with the profitability of chunk and is therefore expressed in the pricing of uploads (see 3.3.3).

In this section, we demonstrate how chunks actually move around in the network: How they are pushed to the storer nodes in the neighbourhood they belong to when they are uploaded, as well as how they are pulled from the storer nodes when they are downloaded.

2.3.1 *Retrieval*

In a distributed chunk store, we say that a chunk is an accessible chunk if a message is routable between the requester and the node that is closest to the chunk. Sending a retrieve request message to the chunk address will reach this node. Because of eventual consistency, the node closest to the chunk address will store the chunk. Therefore, in a DISC distributed chunk store with healthy Kademlia topology all chunks are always accessible for every node.

Chunk delivery

For retrieval, accessibility needs to be complemented with a process to have the content delivered back to the requesting node, preferably using only the chunk address. There are at least three alternative ways to achieve this (see figure 14):

1. direct delivery – The chunk delivery is sent via a direct underlay connection.
2. routed delivery – The chunk delivery is sent as message using routing.
3. backwarding – The chunk delivery response simply follows the route along which the request was forwarded, just backwards all the way to the originator.

Firstly, using the obvious direct delivery, the chunk is delivered in one step via a lower level network protocol. This requires an ad-hoc connection with the associated improvement in latency traded off for worsened security of privacy.[8] Secondly, using routed delivery, a chunk is delivered back to its requestor using ad-hoc routing as determined from the storer's perspective at the time of sending it. Whether direct or routed, allowing deliveries routed independently of the request route presupposes that the requestor's address is (at least partially) known by the storer and routing nodes and as a consequence, these methods leak information identifying the requestor. However, with forwarding–backwarding Kademlia this is not necessary: The storer node responds back to their requesting peer with the delivery while intermediate forwarding nodes remember which of their peers requested what chunk. When the chunk is delivered, they pass it on back to their immediate requestor, and so on until it eventually arrives at the node that originally requested it. In other words, the chunk

8 Beeline delivery has some merit, i.e. bandwidth saving and better latency, so we do not completely rule out the possibility of implementing it.

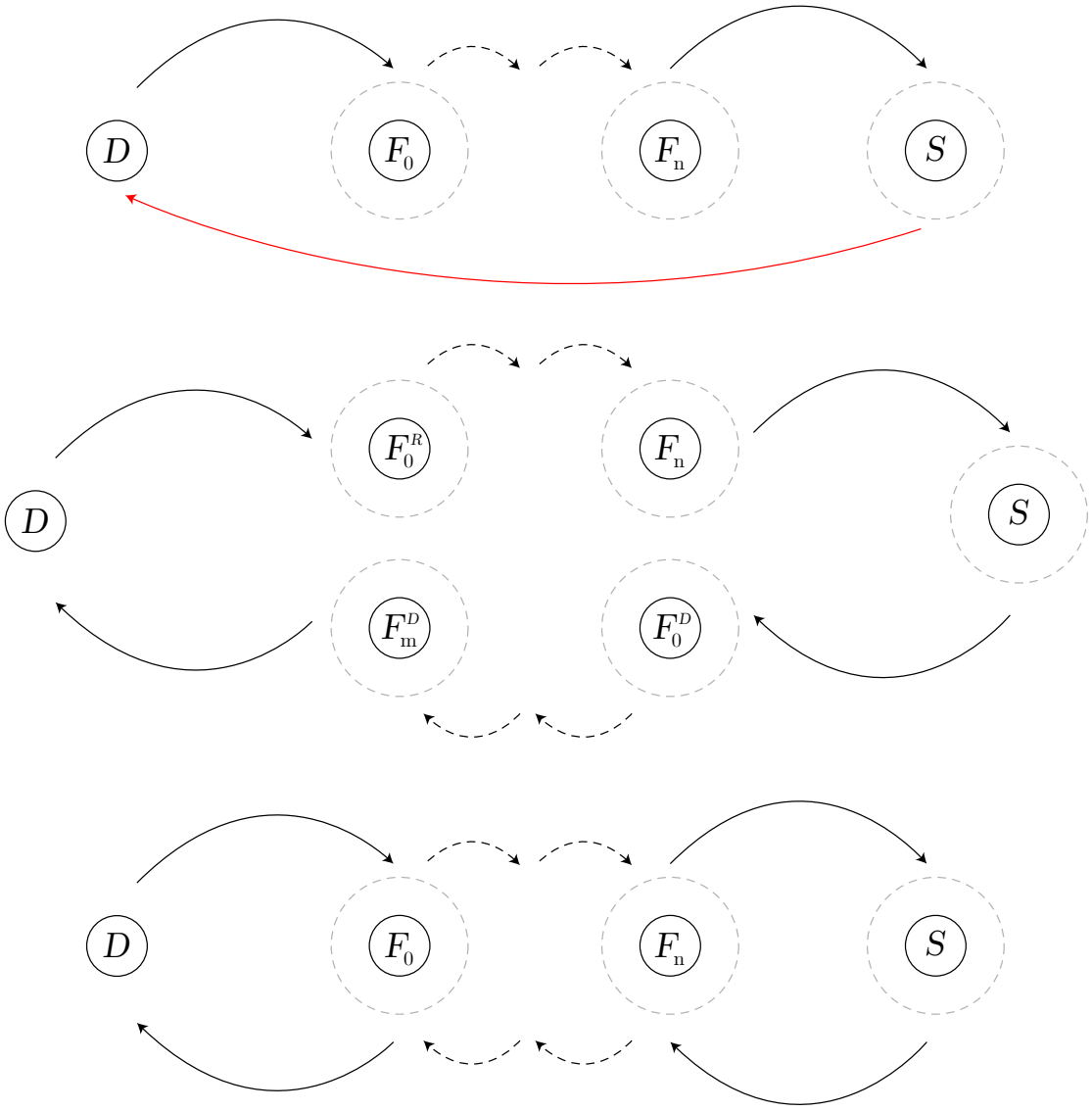


Figure 14: Alternative ways to deliver chunks. **Top:** *direct delivery*: via direct underlay connection. **Centre:** *routed delivery*: chunk is sent using Kademlia routing. **Bottom:** backwarding re-uses the exact peers on the path of the request route to relay the delivery response.

delivery response simply follows the request route back to the originator (see figure 15). Since it is the reverse of the forwarding, we can playfully call this backwarding. Swarm uses this option, which makes it possible to disclose no requestor identification in any form, and thus Swarm implements completely anonymous retrieval.

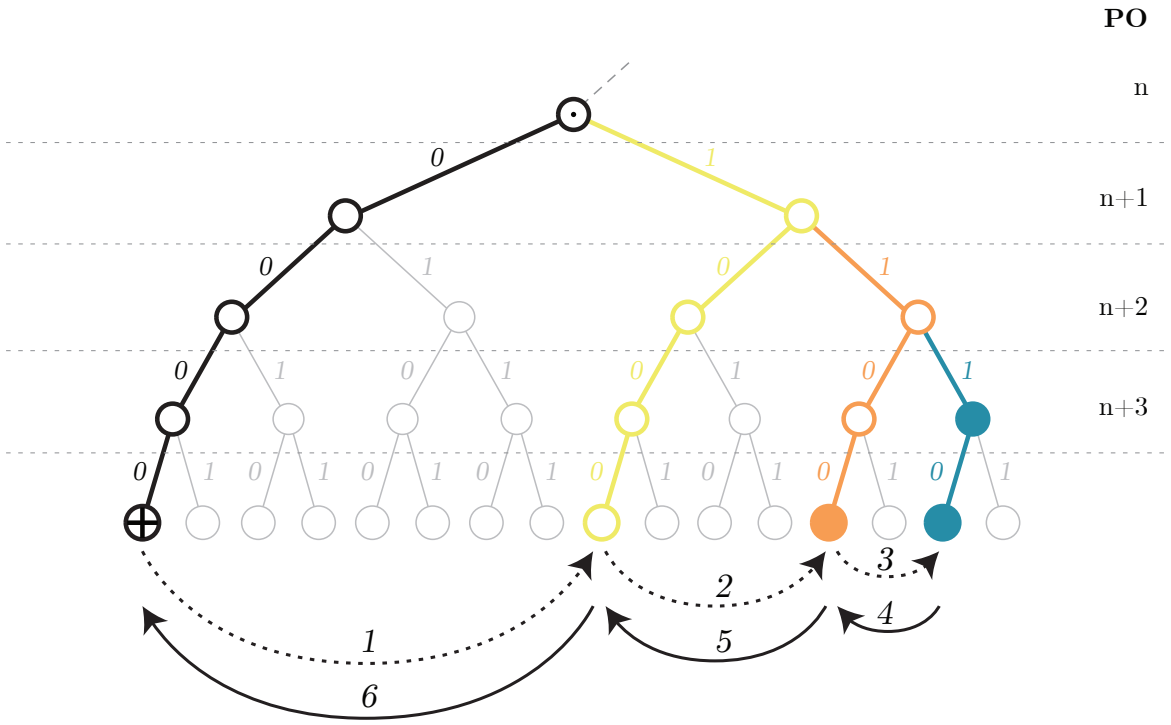


Figure 15: Backwarding: pattern for anonymous request–response round-trips in forwarding Kademlia. Here a node with overlay address ...0000... sending a request to target ....1111... to which the closest online node is ...1110... The leading ellipsis represents the prefix shared by the requestor and target and has a length of n bits, the trailing ellipsis represents part of the address that is not relevant for routing as at that depth nodes are already unique. The request uses the usual Kademlia forwarding, but the relaying nodes on the way remember the peer a request came from so that when the response arrives, they can *backward* it (i.e. pass it back) along the same route.

Requestor anonymity by default in the retrieval protocol is a crucial feature that Swarm insists upon to ensure user privacy and censorship-resistant access.

The generic solution of implementing retrieval by backwarding as depicted in figure 16 has further benefits relating to spam protection, scaling and incentivisation, which are now discussed in the remainder of this section.

Protection against unsolicited chunks

In order to remember requests, the forwarding node needs to create a resource for which it bears some cost (it takes up space in memory). The requests that are not

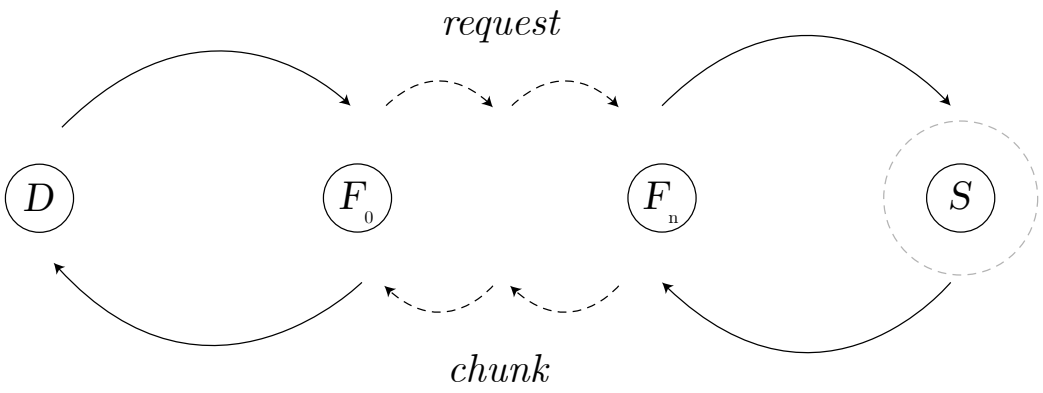


Figure 16: Retrieval. Node D (Downloader) sends a retrieve request to the chunk's address. Retrieval uses forwarding Kademlia, so the request is relayed via forwarding nodes F_0, ..., F_n all the way to node S, the storer node closest to the chunk address. The chunk is then delivered by being passed back along the same route to the downloader.

followed by a corresponding delivery should eventually be garbage collected, so there needs to be a defined time period during which they are active. Downstream peers also need to be informed about the timeout of this request. This makes sense since the originator of the request will want to attach a time to live duration to the request to indicate how long it will wait for a response.

Sending unsolicited chunks is an offence as it can lead to denial of service (DoS). By remembering a request, nodes are able to recognise unsolicited chunk deliveries and penalise the peers sending them. Chunks that are delivered after the request expires will be treated as unsolicited. Since there may be some discrepancy assessing the expiry time between nodes, there needs to be some tolerance for unsolicited chunk deliveries, but if they go above a particular (but still small) percentage of requests forwarded, the offending peer is disconnected and blacklisted. Such local sanctions are the easiest and simplest way to incentivise adherence to the protocol (see 3.2.7).

Rerequesting

There is the potential for a large proportion of Swarm nodes to not be always stably online. Such a high churn situation would be problematic if we used the naive strategy of forwarding requests to any one closer node: If a node on the path were to go offline before delivery is completed, then the request-response round trip is broken, effectively rendering the chunk requested not retrievable. Commitment to pay for a chunk is considered void if the connection to the requested peer is dropped, so there is no harm in re-requesting the chunk from another node (see 9.2).

Timeout vs not found

Note that in Swarm there is no explicit negative response for chunks not being found. In principle, the node that is closest to the retrieved address can tell that there is no chunk at this address and could issue a "not found" response, however this is not desirable for the following reason. While the closest node to a chunk can verify that a chunk is indeed not at the place in the network where it is supposed to be, all nodes further away from the chunk cannot credibly conclude this as they cannot verify it first-hand and all positive evidence about the chunk's retrievability obtained later is retrospectively plausibly deniable.

All in all, as long as delivery has the potential to create earnings for the storer, the best strategy is to keep a pending request open until it times out and be prepared in case the chunk should appear. There are several ways the chunk could arrive after the request: (1) syncing from existing peers (2) appearance of a new node or (3) if a request precedes upload, e.g. the requestor has already "subscribed" to a single owner address (see 6.3) to decrease latency of retrieval. This is conceptually different from the usual server-client based architectures where it makes sense to expect a resource to be either on the host server or not.

Opportunistic caching

Using the backwarding for chunk delivery responses to retrieve requests also enables opportunistic caching, where a forwarding node receives a chunk and the chunk is then saved in case it will be requested again. This mechanism is crucial in ensuring that Swarm scales the storage and distribution of popular content automatically (see 3.1.2).

Incentives

So far, we have shown that by using the retrieval protocol and maintaining Kademlia connectivity, nodes in the network are capable of retrieving chunks. However, since forwarding is expending a scarce resource (bandwidth), without providing the ability to account for this bandwidth use, network reliability will be contingent on the proportion of freeriding and altruism. To address this, in section 3, we will outline a system of economic incentives that align with the desired behaviour of nodes in the network. When these profit maximising strategies are employed by node operators, they give rise to emergent behaviour that is beneficial for users of the network as a whole.

2.3.2 *Push syncing*

In the previous sections, we presented how a network of nodes maintaining a Kademlia overlay topology can be used as a distributed chunk store and how Forwarding

Kademlia routing can be used to define a protocol for retrieving chunks. When discussing retrieval, we assumed that chunks are located with the node whose address is closest to theirs. This section describes the protocol responsible for realising this assumption: ensuring delivery of the chunk to its prescribed storer after it has been uploaded to any arbitrary node.

This network protocol, called push syncing, is analogous to chunk retrieval: First, a chunk is relayed to the node closest to the chunk address via the same route as a retrieval request would be, and then in response a statement of custody receipt is passed back along the same path (see figure 17). The statement of custody sent back by the storer to the uploader indicates that the chunk has reached the neighbourhood from which it is then universally retrievable. By tracking these responses for each constituent chunk of an upload, uploaders can make sure that their upload is fully retrievable by any node in the network before sharing or publishing the address of their upload. Keeping this count of chunks push-synced and receipts received serves as the back-end for a *progress bar* that can be displayed to the uploader to give positive feedback of the successful propagation of their data across the network (see 6.1 and 10.2.5).

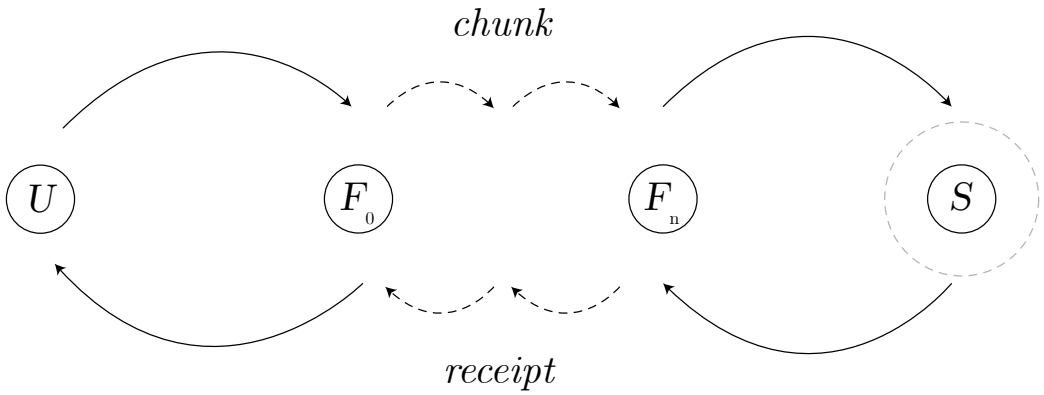


Figure 17: Push syncing. Node U (Uploader) push-syncs a chunk to the chunk's address. Push-sync uses forwarding, so the chunk is relayed via forwarding nodes F_0, ..., F_n all the way to node S, the storer node closest to the chunk address (the arrows represent transfer of the chunk via direct peer-to-peer connection). A statement of custody receipt signed by S is then passed back along the same route as an acknowledgment to the uploader.

Statements of custody are signed by the nodes that claim to be the closest to the address. Similarly to downloaders in the retrieval protocol, the identity of uploaders can also remain hidden, hence forwarding Kademlia can implement anonymous uploads.

Another similarity is that in order to allow backwarding for responses, nodes should remember which peer sent a particular chunk. This record should persist for a short

period while the statement of custody responses are expected. When this period ends, the record is removed. A statement of custody not matching a record is considered unsolicited and is allowed only up to a small percentage of all push-sync traffic with a peer. Going above this tolerance threshold is sanctioned with disconnection and blacklisting (see 3.2.7).

In this section we described how the logistics of chunk uploads can be organised with a network protocol using Forwarding Kademlia routing with response backwarding. However, this solution is not complete until it is secured with aligned incentives: The strategy to follow this protocol should be incentivised and DoS abuse should be disincentivised. These are discussed later in detail in 3.3.1 and 3.1.3).

2.3.3 *Pull syncing*

Pull syncing is the protocol that is responsible for the following two properties:

- *eventual consistency* – Syncing neighbourhoods as and when the topology changes due to churn or new nodes joining.
- *maximum resource utilisation* – Nodes can pull chunks from their peers to fill up their surplus storage.[9]

Pull syncing is node centric as opposed to chunk centric, i.e. it makes sure that a node's storage is filled if needed, as well as syncing chunks within a neighbourhood. When two nodes are connected they will start syncing both ways so that on each peer connection there is bidirectional chunk traffic. The two directions of syncing are managed by distinct and independent *streams* (see 8.6). In the context of a stream, the consumer of the stream is called downstream peer or client, while the provider is called the upstream peer or server.

When two nodes connect and engage in chunk synchronisation, the upstream peer offers all the chunks it stores locally in a data stream per proximity order bin. To receive chunks closer to the downstream peer than to the upstream peer, a downstream peer can subscribe to the chunk stream of the proximity order bin that the upstream peer belongs to in their Kademlia table. If the peer connection is within the nearest neighbour depth d, the client subscribes to all streams with proximity order bin d or greater. As a result, peers eventually replicate all chunks belonging to their area of responsibility.

A pull syncing server's behaviour is referred to as being that of a stream provider in the stream protocol (see 8.6). Nodes keep track of when they stored a chunk locally by indexing them with an ever increasing storage count, called the bin ID. For each

9 Maximum storage utilisation may not be optimal in terms of the profitability of nodes. Put differently, storer nodes have an optimal storage capacity, based on how often content is requested from them. This means that in practice, profit-optimised maximum utilisation of storage capacity requires operators to run multiple node instances.

proximity order bin, upstream peers offer to stream chunks in descending order of storage timestamp. As a result of syncing streams on each peer connection, a chunk can be synced to a downstream peer from multiple upstream peers. In order to save bandwidth by not sending data chunks to peers that already have them, the stream protocol implements a round-trip: Before sending chunks, the upstream peer offers a batch of chunks identified by their address, to which downstream responds with stating which chunks in the offered batch they actually need (see figure 18). Note that downstream peer decides whether they have the chunk based on the chunk address. Thus, this method critically relies on the chunk integrity assumption discussed in 2.2.1.

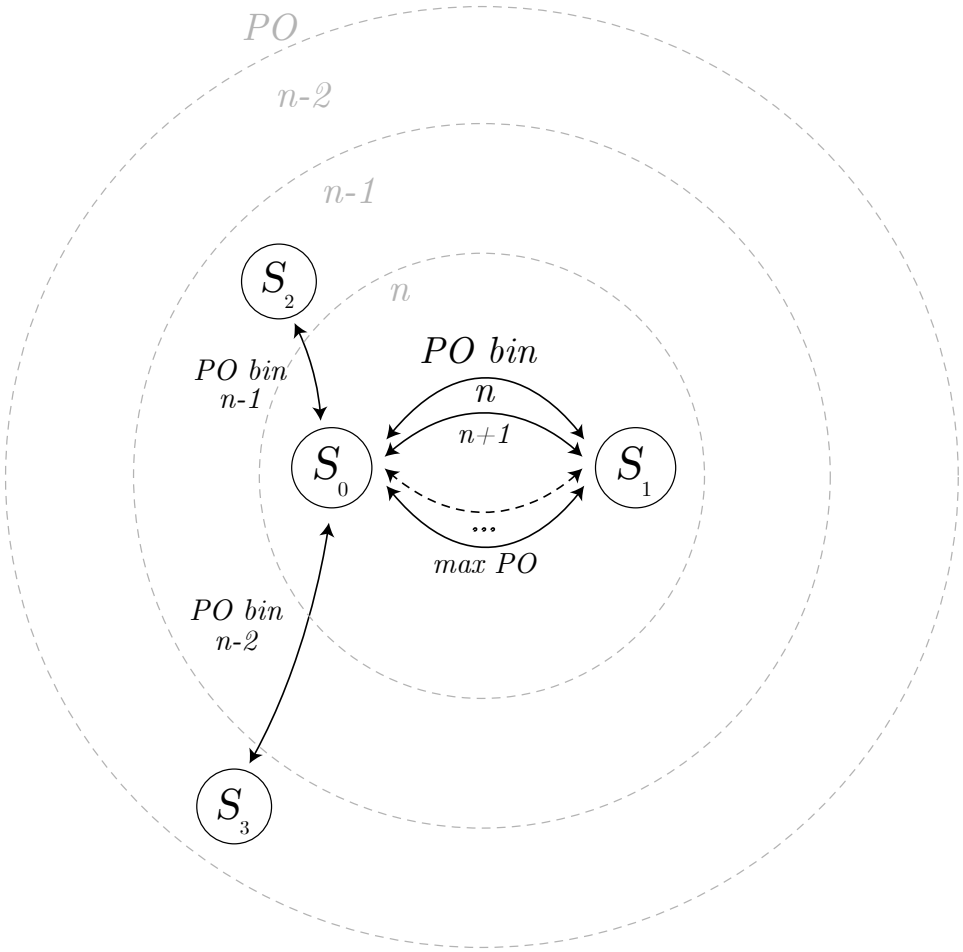


Figure 18: Pull syncing. Nodes continuously synchronise their nearest neighbourhood. If they have free capacity they also pull sync chunks belonging to shallower bins from peers falling outside the neighbourhood depth.

In the context of a peer connection, a client is said to be *synced* if it has synced all the chunks of the upstream peer. Note that due to disk capacity limitations, nodes must impose a value cutoff and as such "all chunks" reads as shorthand for "all chunks

having value greater than v" (v is a constant ranking function, the origin of which is discussed later in 3.3.3). In order for a node to promise they store all chunks with value greater than v, all its neighbours must have stored all chunks greater than value v. In other words, nodes syncing inherit the maximum such value from among their storer peers.

If chunks are synced in the order they are stored, this may not result in the node always having the most profitable (most often requested) chunks. Thus it may be advisable to sync chunks starting with the most popular ones according to upstream peers and finish syncing when storage capacity is reached. In this way, a node's limited storage will be optimised. Syncing and garbage collection are discussed further in 3.3.1 and 3.3.3 and a consolidated client strategy is specified in 9.6.

To conclude this section, we show how the criteria of eventual consistency are met in a healthy Swarm. Chunks found in the local store of any node will become retrievable after being synced to their storers. This is because as long as those as peers in the network pull chunks closer to them than to the upstream peer, each chunk travels a route that would also qualify as valid a forwarding path in the push-sync protocol. If new nodes are added, and old nodes drop out, neighbourhoods change, but as long as local redundancy is high enough that churn can not render previously retrievable chunks non-retrievable, neighbourhoods eventually replicate their content and redundancy is restored. Consider the unlikely event that a whole new neighbourhood is formed and the nodes that originally held the content belonging to this neighbourhood end up outside of it and therefore those chunks are temporarily not available. Even in this scenario, as long as there is a chain of nodes running pull-syncing streams on the relevant bins, redundant retrievability is eventually restored.

2.3.4 *Light nodes*

The concept of a light node refers to a special mode of operation necessitated by poor bandwidth environments, e.g. mobile devices on low throughput networks or devices allowing only transient or low-volume storage.

A node is said to be light by virtue of not participating fully in the usual protocols detailed in the previous sections, i.e. retrieval, push syncing or pull syncing.

A node that has restricted bandwidth environment or in whatever way has limited capacity to maintain underlay connections is not expected to be able to forward messages conforming to the rules of Kademlia routing. This needs to be communicated to its peers so that they do not relay messages to it.

As all protocols in Swarm are modular, a node may switch on or off any protocol independently (depending on capacity and earnings requirements). To give an example: a node that has no storage space available, but has spare bandwidth, may participate as a forwarding node only. Of course, while switching off protocols is technically feasible, a node must at all times take into account the fact that his/her peers expect a

certain level of service if this is advertised and may not accept that some services are switched off and choose not to interact with that node.

Since forwarding can earn revenue, these nodes may still be incentivised to accept retrieve requests. However, if the light node has Kademlia connectivity above proximity order bin p (i.e. they are connected to all storer nodes within their nearest neighbourhood of r peers at depth d, and there is at least one peer in each of their proximity order bin from p to d), they can advertise this and therefore participate in forwarding.

When they want to retrieve or push chunks, if the chunk address falls into a proximity order bin where there are no peers, they can just pick a saturated peer in another bin. Though this may result in a spurious hop (where the proximity of the message destination to the latest peer does not increase as a result of the relaying), the Kademlia assumption that routing can be completed in logarithmic steps still holds valid.

A node that is advertised as a storer/caching node is expected to store all chunks above a certain value. In order to have consistency, they need to synchronise content in their area of responsibility which necessitates their running of the pull-sync protocol. This is also so with aspiring storer nodes, which come online with available storage and open up to pull-sync streams to fill their storage capacity. In the early stages of this, it does not make sense for a node to sync to other full storer nodes. However, it can still be useful for them to sync with other similar newcomer nodes, especially if storer nodes are maxing out on their bandwidth.

The crucial thing here is that for redundancy and hops to work, light nodes with incomplete, unsaturated Kademlia tables should not be counted by other peers towards saturation.

3

INCENTIVES

The Swarm network comprises many independent nodes, running software which implements the Swarm protocol (see chapter 8). It is important to realize that, even though nodes run the same protocol, the emergent behavior of the network is not guaranteed by the protocol alone; as nodes are autonomous, they are essentially "free" to react in any way they desire to incoming messages of peers. It is, however possible to make it profitable for a node to react in a way that is beneficial for the desired emergent behavior of the network, while making it costly to act in a way that is detrimental. Broadly speaking, this is achieved in Swarm by enabling a transfer of value from those nodes who are using the resources of the network (net users) to those who are providing it (net providers).

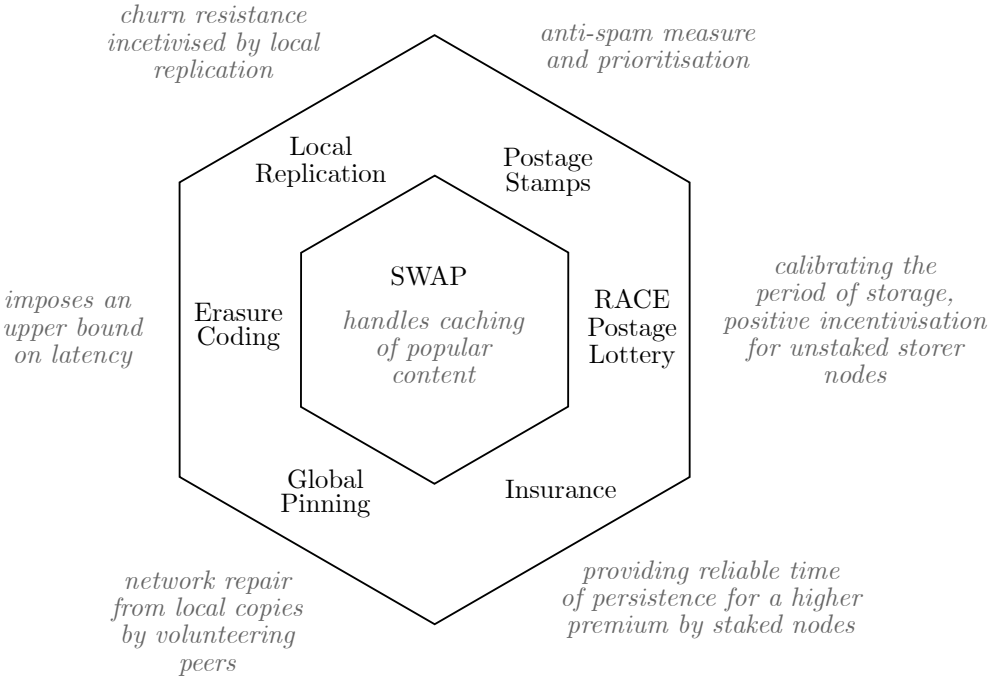


Figure 19: Incentive design

3.1.1 *Incentives for serving and relaying*

Forwarding Kademlia and repeated dealings

Retrieval of a chunk is ultimately initiated by someone accessing content and therefore, all costs related to this retrieval should be borne by them. While paid retrievals may not sound like a popular idea when today's web is "free", many of the problems with the current web stems from consumers' inability to share the costs of hosting and distribution with content publishers directly. In principle, the retrieval of a chunk can be perceived as a functional unit where the storer acts as a service provider and the requestor as consumer. As service is given by provider to consumer, compensation should be given by consumer to provider. Such a direct transaction would normally require that transactors are known to each other, so if we are to maintain the anonymity requirement on downloads, we must conceptualise compensation in a novel way.

As we use Forwarding Kademlia, chunk retrieval subsumes a series of relaying actions performed by forwarding nodes. Since these are independent actors, it is already necessary to incentivise each act of relaying independently. Importantly, if only instances of relaying are what matters, then, irrespective of the details of accounting and compensation (see 3.2.1), transactors are restricted to connected peers. Given the set of ever connected peers is a quasi-permanent set across sessions, this allows us to frame the interaction in the context of repeated dealings. Such a setting always creates extra incentive for the parties involved to play nice. It is reasonable to exercise preference for peers showing untainted historical record. Moreover, since this quasi-permanent set is logarithmic to network size, any book-keeping or blockchain contract that the repeated interaction with a peer might necessitate is kept manageable, offering a scalable solution. Turning the argument around, we could say that keeping balances with a manageable number of peers, as well as the ambiguity of request origination are the very reasons for nodes to have limited connectivity, i.e., that they choose leaner Kademlia bins.

Charging for backwarded response

If accepting a retrieve request already constitutes revenue for forwarding nodes, i.e. an accounting event crediting the downstream peer is triggered before the response is delivered, then it creates a perverse incentive not to forward the requests. Conditioning the request revenue fulfilment on successful retrieval is the natural solution: The accounting event is triggered only when a requested chunk is delivered back to its requestor, see figure 20.

If, however, there is no cost to a request, then sending many illegitimate requests for non-existing chunks (random addresses) becomes possible. This is easily mitigated by

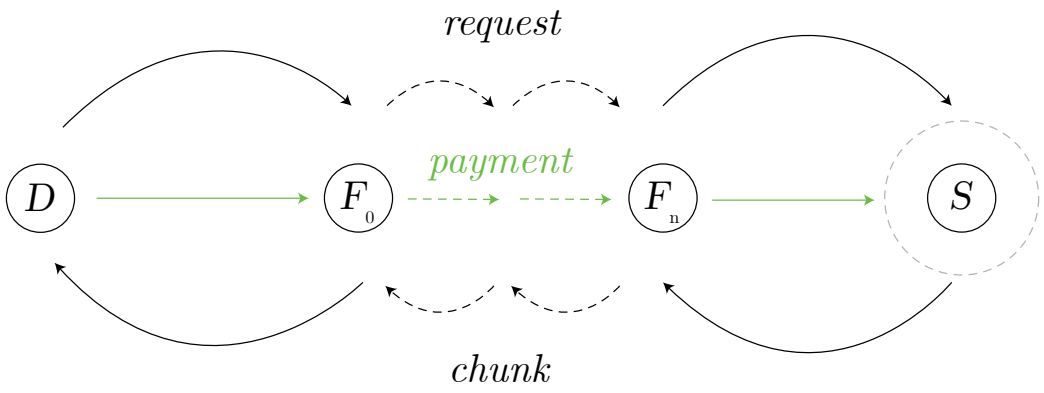


Figure 20: Incentivising retrieval. Node D (Downloader) sends a retrieve request to the chunk's address. Retrieval uses forwarding, so the request is relayed via forwarding nodes F_0, ..., F_n all the way to node S, the storer node closest to the chunk address. The chunk is delivered by being passed back along the same route to the downloader. Receiving the chunk response triggers an accounting event.

imposing sanctions on peers that send too many requests for chunks that do not exist (see 3.2.7).

Once a node initiates (starts or forwards) a request, it commits to pay for that chunk if it is delivered within the defined time to live (TTL), therefore there is never an incentive to block timely deliveries when the chunk is passed back. This commitment also dissuades nodes from frivolously asking too many peers for a chunk, since, if multiple peers respond with delivery, each must be paid.

3.1.2 *Pricing protocol for chunk retrieval*

Next, we describe the protocol which nodes use to communicate their price for delivering chunks in the Swarm network. Building on top of this protocol, strategies can then be implemented by nodes who wish to compete in the market with other nodes in terms of quality of service and price (see 9.3).

Price discovery

The main merit of the protocol is that it allows for the mechanisms of price discovery to be based only on local decisions, which is essential for the following reasons: (1) Bandwidth costs are not homogeneous around the world: Allowing nodes to express their cost structure via their price will enable competition on price and quality, ultimately benefiting the end-user. (2) The demand for bandwidth resource is constantly changing due to fluctuations in usage or connectivity. (3) Being able to react directly to changes creates a self-regulating system.

Practically, without this possibility, a node operator might decide to shut down their node when costs go up or conversely end-users might overpay for an extended period of time when costs or demand decrease and there is no competitive pressure for nodes to reduce their price accordingly.

Bandwidth is a service that comes with "instant gratification" and therefore immediate acknowledgement and accounting of its cost are justified. Since it is hard to conceive of any externality or non-linearity in the overall demand and supply of bandwidth, a pricing mechanism which provides for both (1) efficient and immediate signalling, as well as (2) competitive choice with minimal switching and discovery cost, is most likely to accommodate strategies that result in a globally optimal resource allocation.

To facilitate this, we introduce a protocol message that can communicate these prices to upstream peers (see 8.4). We can conceptualise this message as an alternative response to a request. Nodes maintain the prices associated with each peer for each proximity distance, so when they issue a retrieve request they already know the price they commit to pay in the event that the downstream peer successfully delivers the valid chunk within the time to live period. However, there is no point in restricting the price signal just to responses: For whatever reason a peer decides to change the prices, it is in the interest of both parties to exchange this information even if there is request to respond to. In order to prevent DoS attacks by flooding upstream peers with price change messages, the rate of price messages is limited. Well behaved and competitively priced nodes are favoured by their peers; if a node's prices are set too high or their prices exhibit a much higher volatility than others in the network, then peers will be less willing to request chunks from them.[1]

For simplicity of reasoning we posit that the default price is zero, corresponding to a free service (altruistic strategy, see 9.3).

Differential pricing of proximities

If the price of a chunk is the same at all proximities, then there is no real incentive for nodes to forward requests other than the potential to cache the chunk and earn revenue by reselling it. This option is hard to justify for new chunks, especially if they are in the shallow proximity orders of a node where they are unlikely to be requested. More importantly, if pricing of chunks is uniform across proximity orders, colluding nodes can generate chunk traffic and pocket exactly as much as they send, virtually a free DoS attack (see figure 21).

To mitigate this attack, the price a requestor pays for a chunk needs to be strictly greater than what the storer node would receive as compensation when a request is routed from requestor to storer. We need to have a pricing scheme that rewards forwarding nodes, hence, this necessitates the need for differential pricing by node

1 While this suggests that unreasonable pricing is taken care of by market forces, in order to prevent catastrophic connectivity changes as a result of radical price fluctuations, limiting the rate of change may need to be enforced on the protocol level.

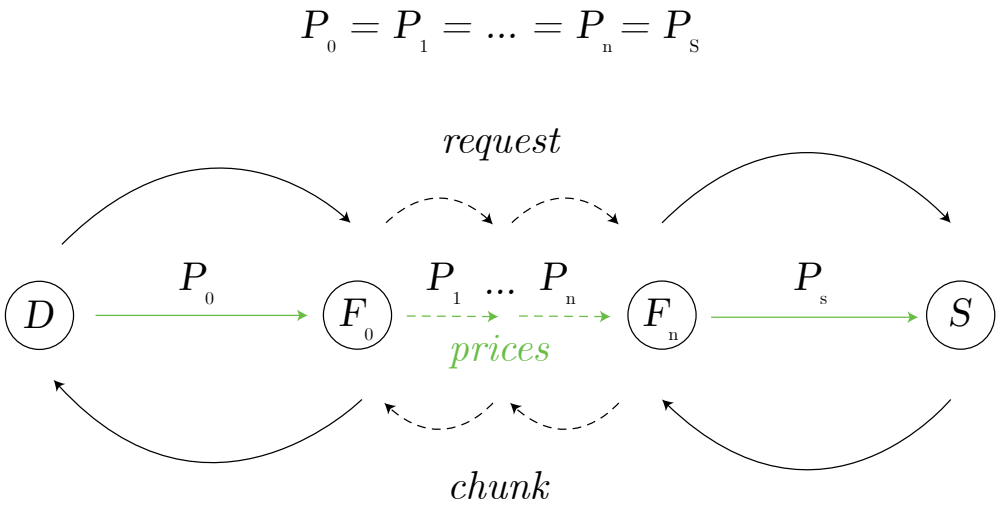


Figure 21: Uniform chunk price across proximities would allow a DoS attack. An attacker can create a flow of traffic between two nodes D and S by sending retrieve requests towards S which only S can serve. If prices are the same across proximities, such an attack would incur no cost for the attacker.

proximity. If the price of delivery is lower as a node gets further from the chunk, then the request can always be sent that way because the forwarder will pocket the difference and therefore make a profit. This means that an effective differential scheme will converge to a pricing model where delivery costs more if the peer is further from the chunk address, i.e. rewards for chunk deliveries are a decreasing function of proximity.

Due to competitive pressure along the delivery path and in the neighborhood, we expect that the differential a node is applying to the downstream price to converge towards the marginal cost of an instance of forwarding. The downstream price is determined by the bin density of the node. Assuming balanced bins with cardinality 2^n, a node can guarantee to increase the proximity order by n in one hop. At the same time it also means that they can spread the cost over n proximity bins pushing the overall price down.

Uniformity of price across peers

Take a node A that needs to forward a request for a chunk which falls into A's PO bin n. Notice that all other peers of A in bins $n+1, n+2, ...$, just like A also have the chunk in their PO n. If any of these peers, say B, has a price for proximity order n cheaper than A, A can lower its price for PO bin n, forward all increased traffic to B and still pocket the difference, see figure 22. Note that this is not ideal for the

network as it introduces a spurious hop in routing, i.e., in relaying without increasing the proximity.

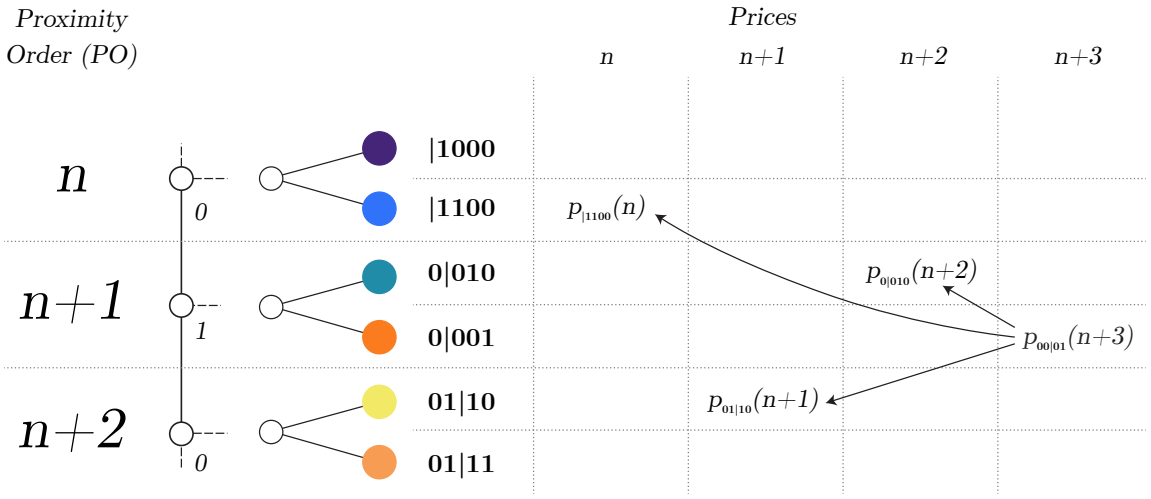


Figure 22: Price arbitrage. Nodes keep a pricetable for prices of every proximity order for each peer. The digaram shows node 0101 trying to forward a retrieve request for 0000. The arrows originate from the closest node, and point to cells where other peers although further from the chunk, offer cheaper to forward. Choosing the cheaper peer will direct traffic away from the overpriced peer and lead to a pressure on both to adjust.

Similarly, peers of A in shallower bins that have lower price than A for their respective bins, e.g. B in bin $n-1$ is cheaper than A in bin n, then A can always forward any request to B and pocket the difference.

Now let's assume that all peers have price tables which are monotonically decreasing as PO decreases. Also assume that shallower bins have higher prices for bins less than n and all deeper peers in bins higher than n have the same prices for n. Let B, C, D and E be the peers in bin n densely balanced. A wants to forward a chunk to a peer so that the PO with its target address increases by 3. If the peers B and C attempt to collude against A and raise the price of forwarding chunks to bin $n+3$, they are still bound by D and E's price on PO bin $n+2$. In particular if they are lower than B and C for $n+3$.

Such price discrepancies offer nodes an arbitrage opportunity; the strategy to forward to the cheapest peer will direct traffic away from expensive peers and increase traffic for cheaper ones. As a consequence prices will adjust.

All else being equal, this price arbitrage strategy will achieve (1) uniform prices for the same proximity order across the network, (2) prices that linearly decrease as a function of proximity (3) nodes can increase connectivity and keep prices lower. In this way, incentivisation is designed so that strategies that are beneficial to individual nodes are also neatly aligned in order to benefit the health of the system as a whole.

Bin density

Charging based on the downstream peer's proximity to the chunk has the important consequence that the net revenue earned from a single act of non-local delivery to a single requestor is a monotonically increasing function of the difference between the chunk's proximity to the node itself vs to the peer the request was forwarded to. In other words, the more distance we can cover in one forward request, the more we earn.

This incentive aligns with downloaders' interest to save hops in serving their requests leading to lower latency delivery and less bandwidth overhead. This scheme incentivises nodes to keep a gap-free balanced set of addresses in their Kademlia bins as deep as possible (see figure 5), i.e, it is better for a node to keep dense Kademlia bins than thin ones.

Nodes that are able to maintain denser bins actually have the same cost as thinner ones, but saving hops will improve latency and make the peer more efficient. This will lead to the peer being preferred over other peers that have the same prices. Increased traffic essentially can also lead to bandwidth contention which eventually allows the raising of prices.

Note that such arbitrage is more efficient in shallow bins bins where the number of peers to choose from is higher. This is in major opposition to deep bins in the area of responsibility. If a node does not replicate its neighbourhoods chunks, some of these chunks will need to be requested by the node closer to the address but further from the node. This will only be possible at a loss. An added incentive for neighbours to replicate their area of responsibility is discussed in 3.3.2. With the area of responsibility stored however, a node can choose to set their price arbitrarily.

Caching and auto-scaling

Nodes receive a reward every time they serve a chunk, therefore the profitability of a chunk is proportional to its popularity: the more often a chunk is requested, the higher the reward relative to the fixed cost of storage per time unit. When nodes reach storage capacity limits and it comes to deciding which chunks to delete, the optimal strategy of a rational profit maximising agent is to remove chunks whose profitability is lowest. A reasonably[2]. good predictor for this is the age of last request. In order to maximise the set of chunks to select from, nodes engage in opportunistic caching of the deliveries they relay as well as the chunks they sync. This then results in popular chunks being more widely spread and faster served, making the entire swarm an auto-scaled and auto-balanced *content distribution network*.

2 Better metrics for predicting chunk profitability than the age of last request will continue to be identified and developed(see also 9.7)

Non-caching nodes

Any scheme which leaves relaying nodes a profit creates a positive incentive for forwarding-only non-caching nodes to enter the network. Such nodes are not inherently beneficial to the network as they are creating unnecessary bandwidth overhead. On the one hand, their presence could in principle unburden storer nodes from relaying traffic, so using them in shallow bins may not be detrimental. On the other hand, closer to neighbourhood depth, their peers will favour a caching/storing node to them because of their disadvantage at least for chunks in their hypothetical area of responsibility. Non-caching nodes can also contribute to increase anonymity (see 2.3.1).

3.1.3 *Incentivising push-syncing*

Push-syncing (see 2.3.2) is the protocol which ensures that chunks that are uploaded into the network arrive at their proper address. In what follows, we will explain how forwarding is incentivised. [3]

The push-sync protocol is analogous to the retrieval protocol in the sense that their respective message exchange sequences travel the same route. The delivery of the chunk in the push sync protocol is analogous to a retrieval request and, conversely, the statement of custody receipt in push sync is analogous to the chunk delivery response in retrieval.

Push-syncing could in principle be left without explicit forwarding incentives. Due to the retrieval protocol, as nodes expect chunks to be found in the neighbourhood of their address, participants in swarm are at least weakly incentivised to help uploaded chunks get to their destination by taking part in the protocol. However, we need to provide the possibility that chunks are uploaded via nodes further from it than the requestor (light nodes or retries). Thus, if push-syncing was free, nodes could generate wasteful amounts of bandwidth.

Requiring payment only for push-sync delivery by the downstream peers would put the forwarder in a position to bargain with a storer node on the delivery of the chunk. The possession of a chunk is valuable for the prospective storer node because there is also a system of rewards for storage (see 3.3.2). Given this, the forwarder node could in theory hold onto the chunk unless the storer node pays marginally more than what the possession of this chunk is worth for them factoring in the profit potential due to storage incentives. In particular, since forwarders on the route from the uploader are not numerous, any profit coming from a reward mechanism for storage could then be captured by the forwarding nodes.

Instead, in push-sync, by making the statement of custody receipt a paid message, roles switch. Now, a forwarder node is not in the position to bargain. To see why,

3 To complement our solution for bandwidth compensation, further measures are needed for spam protection and storage incentivisation which are discussed later in 3.3.1 and 3.3.2, respectively.

consider what would happen if a forwarding node tries to hold on to a chunk to get a price for pushing this chunk to a storer node. In this case, the uploader will not get a statement of custody receipt in due time, assume the attempt has failed and re-upload the chunk via a different route. Now, suddenly the original forwarding node is forced to compete with another forwarding node in getting compensation for their bandwidth costs. Since all forwarding nodes know this, emergent behavior will produce a series of peers that are willing to forward the chunk to the storer node for a only small compensation and the bandwidth costs incurred. Now there is no need for the original forwarding node to try and bargain with the storer node in the first place: Instead, they can make a small profit immediately when they pass back the statement of custody receipt.

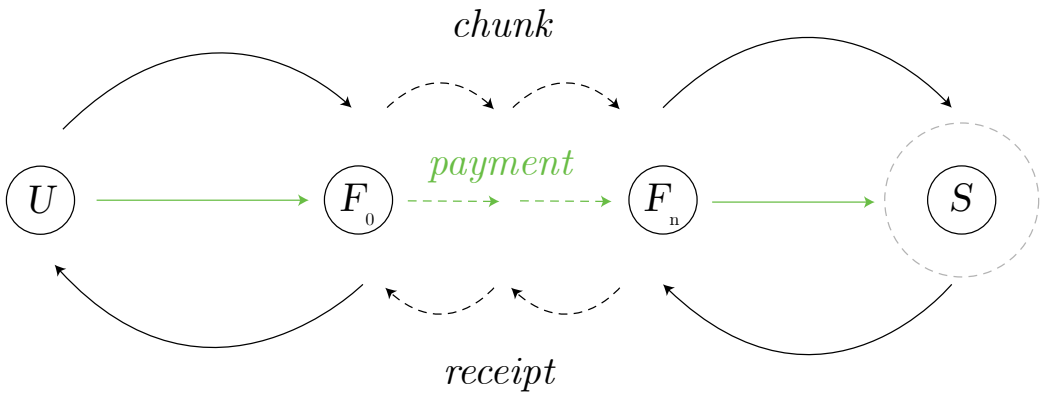


Figure 23: Incentives for push-sync protocol. Node U (uploader) sends the chunk towards its address, the closest node to which is node S (storer) via forwarding nodes $F_0, \ldots F_n$. The storer node responds with a statement of custody receipt which is passed back to the uploader via the same forwarding nodes $F_n, \ldots F_0$. Receiving the statement of custody receipt triggers an accounting event.

This scheme makes it clear why the incentivisation of the two protocols relies on the same premises: there are many sellers (forwarders) and only one buyer (uploader) for a homogeneous good (the statement of custody receipt). This drives the price of the service (delivering the chunk to the storer) to the sum of all marginal cost of forwarding for each node along the route, while simultaneously allowing the storer node to capture all profits from the storage compensation scheme.

In this way, we can make sure that (1) storers actually respond with receipts, and (2) have a way to detect timed out or unsolicited receipt responses to protect against DoS, see figure 23.

Just as in the retrieval protocol, the pricing is expected to be different for different proximities (see 3.1.2 and as the costs of the nodes in the network change (depending on capacity utilization and efficiency of nodes), the pricing will be variable over time as well. Since at the point of accounting the compensation is due for one chunk and

one shorter message (retrieve request and custody receipt), we can safely conclude that the price structure for forwarding in the case of the two protocols are identical and therefore one generic forwarding pricing scheme can be used for both (see 3.1.2) What makes a difference is that unlike the retrieval protocol where the chunk is delivered back and its integrity can be validated, the accounting event in pushsync is a statement of custody which can be spoofed. With the forwarding incentive nodes will have the motivation not to forward and impersonate a storer node and issue the statement of custody. This makes it advisable to query (retrieve) a chunk via alternative routes. If such retrievals fail, it may be necessary to try to push-sync chunks via alternative routes.

3.2 SWAP: ACCOUNTING AND SETTLEMENT

This section covers aspects of incentivisation relating to bandwidth sharing. In 3.2.1, we introduce a mechanism to keep track of the data traffic between peers and offer peer to peer accounting for message relaying. Subsequently, in 3.2.2, we describe the conditions of compensating for unbalanced services and show how settlement can be achieved. In particular we introduce the concept of a. cheques and the chequebook contract. In 3.2.3, we discuss waivers, a further optimisation allowing for more savings on transaction cost. In 3.2.5 we discuss how an incentivised service of sending in cashing transactions enables zero-cash entry to Swarm and, finally, in 3.2.7 we will discuss the basic set of sanctions that serve as a fundamental incentive for nodes to play nice and adhere to the protocols.

3.2.1 *Peer to peer accounting*

[Tron et al., 2016] introduces a protocol for peer-to-peer accounting, called swap. Swap is a tit-for-tat accounting scheme that scales microtransactions (see 8.7). The scheme allows directly connected peers to swap payments or payment commitments. The major features of the system are captured playfully with different mnemonic resolutions of the acronym SWAP:

- *Swarm accounting protocol* for *service wanted and provided* – Account service for service exchange.
- *settle with automated payments* – Send cheque when payment threshold is exceeded.
- *send waiver as payment* – Debt can be waived in the value of un-cashed cheques.
- *start without a penny* and *send with a peer* – Zero cash entry is supported by unidirectional swap.

Service for service

swap allows service for service exchange between connected peers. In case of equal consumption with low variance over time, bidirectional services can be accounted for without any payments. Data relaying is an example of such a service, making Swap ideally suited for implementing bandwidth incentives in content delivery or mesh networks.

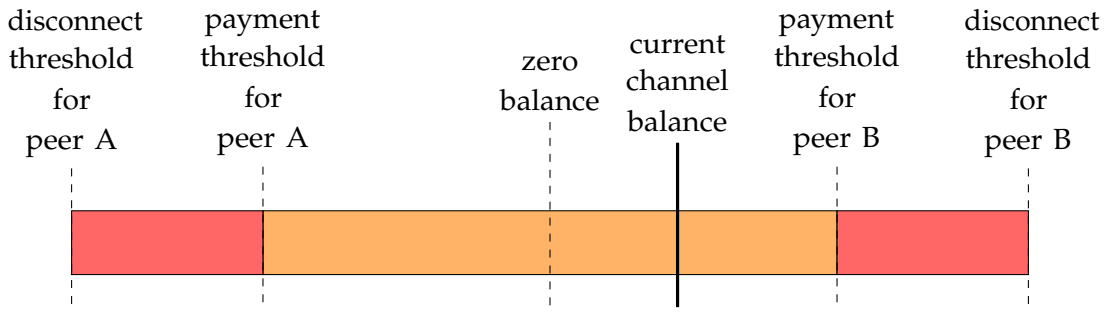


Figure 24: Swap balance and swap thresholds. Zero balance in the middle indicates consumption and provision are equal. The current channel balance represents the difference in uncompensated service provision: If to the right of zero, the balance tilts in favour of A with peer B being in debt, whereas to the left the balance tilts in favour of B with A being in debt. The orange interval represents loss tolerance. If the balance goes over the payment threshold, the party in debt sends a cheque to its peer, if it reaches the disconnect threshold, the peer in debt is disconnected.

Settling with payments

In the presence of high variance or unequal consumption of services, the balance will eventually tilt significantly toward one peer. In this situation, the indebted party issues a payment to the creditor to return the nominal balance to zero. This process is automatic and justifies swap as *settle (the balance) with automated payments* (see figure 24). These payments can be just commitments.

Payment thresholds

To quantify what counts as "significant tilt", the swap protocol requires peers to advertise a payment threshold as part of the handshake (8.7): When their relative debt to their peer goes above this threshold, they send a message, containing a payment to their peer. It is reasonable for any node to send a message at this level, as there also exist a disconnect threshold. The disconnect threshold is set freely by any peer, but a reasonable value is such that the difference between the payment threshold and the disconnect threshold accounts for the normal variance in accounting balances of the two peers. (see 9.4).

Atomicity

Sending the cheque and updating the balance on the receiving side cannot be made an atomic operation without substantial added complexity. For instance, a client could crash between receiving and processing the message, so even if the sending returns with no error, the sending peer can not be sure the payment was received, this can result in discrepancies in accounting on both sides. The tolerance expressed by the difference between the two thresholds (*DisconnectThreshold* − *PaymentThreshold*) guards against this, i.e. if the incidence of such crashes is not high and happen with roughly equal probability for both peers, the resulting minor discrepancies are filtered out. In this way, the nodes are shielded from sanctions.

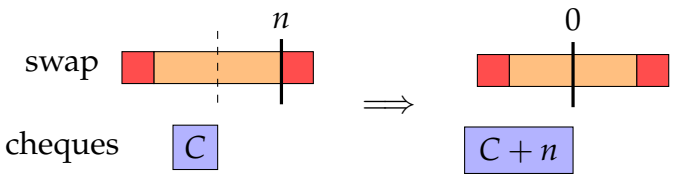


Figure 25: Peer B's swap balance (with respect to A) reaches the payment threshold (left), B sends a cheque to peer A. B keeps the cheque and restores the swap balance to zero.

3.2.2 *Cheques as off-chain commitments to pay*

One of the major issues with direct on-chain payment in a blockchain network is that each transaction must be processed by each and every node participating in the network, resulting in high transaction costs. It is however, possible to create a payment without presenting this payment on-chain. Such payments are called second-layer payment strategies. One such strategy is to defer payments and process them in bulk. In exchange for reduced cost, the beneficiary must be willing to incur higher risk of settlement failure. We argue that this is perfectly acceptable in the case of bandwidth incentivisation in Swarm, where peers will engage in repeated dealings.

The chequebook contract

A very simple smart contract that allows the beneficiary to choose when payments are to be processed was introduced in [Tron et al., 2016]. This chequebook contract is a wallet that can process cheques issued by its owner. These cheques are analogous to those used in traditional financial transactions: The issuer signs a *cheque* specifying a *beneficiary*, a *date* and an *amount*, gives it to the recipient as a token of promise to pay at a later date. The smart contract plays the role of the bank. When the recipient

wishes to get paid, they "cash the cheque" by submitting it to the smart contract. The contract, after validating the signature, date and the amount specified on the cheque, transfers the amount to the beneficiary's account (see figure 26). Analogous to the person taking the cheque to the bank to cash it, anyone can send the digital cheque in a transaction to the owner's chequebook account and thus trigger the transfer.

The swap protocol specifies that when the *payment threshold* is exceeded, a cheque is sent over by the creditor peer. Such cheques can be cashed immediately by being sent to the issuer's chequebook contract. Alternatively, cheques can also be held. Holding a cheque is effectively lending on credit, which enables the parties to save on transaction costs.

The amount deposited in the chequebook (global balance) serves as collateral for the cheques. It is pooled over the beneficiaries of all outstanding cheques. In this simplest form, the chequebook has the same guarantee as real-world cheques: None. Since funds can be freely moved out of the chequebook wallet at any time, solvency at the time of cashing can never be guaranteed: If the chequebook's balance is less than the amount sanctioned by a cheque submitted to it, the cheque will bounce. This is the trade off between transaction costs and risk of settlement failure.

While, strictly speaking, there are no guarantees for solvency, nor is there an explicit punitive measure in the case of insolvency, a bounced cheque will affect the issuer's reputation as the chequebook contract records it. On the premise that cheques are swapped in the context of repeating dealings, peers will refrain from issuing cheques beyond their balance. In other words, a node's interest in keeping a good reputation with their peers serves as a incentive enough to maintain its solvency.

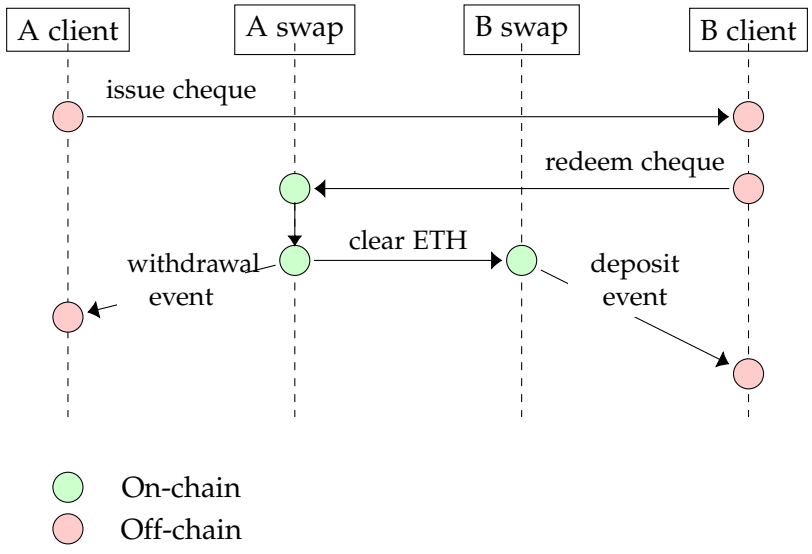


Figure 26: The basic interaction sequence for swap chequebooks

Double cashing

Since these digital cheques are files and can therefore be copied, care must be taken that the same cheque cannot be cashed twice. Such "double cashing" can be prevented by assigning each cheque given to a particular beneficiary a serial number which the contract will store when the cheque is cashed. The chequebook contract can then rely on the serial number to make sure cheques are cashed in sequential order, thus needing to store only a single serial number per beneficiary.

An alternative strategy to prevent double cashing, when repeated payments are made to the same beneficiary, is that the cheques contain the *cumulative* total amount ever credited to the beneficiary. The total amount that has been cashed out is stored in the contract for each beneficiary. When a new cheque is submitted, the contract ignores cheques with amount equal to or less than the stored total, but it will transfer the difference if it receives a cheque with a higher total.

This simple trick also makes it possible to cash cheques in bulk because only the current "last cheque" need ever be processed. This achieves the reduction of transaction costs alluded to above.

Cashing without Ether

Not all peers in Swarm are expected to have the Ether needed to pay for the transaction costs to cash out a cheque. The chequebook allows third parties to cash cheques. The sender of the transaction is incentivised with a reward for the service performed.

3.2.3 *Waivers*

If the imbalance in the swap channel is the result of high variance as opposed to unequal consumption, after a period of accumulating cheques the channel balance starts tilting the other way. Normally, it is now up to the other party to issue cheques to its peer resulting in un-cashed cheques accumulating on both sides. To allow for further savings in transaction costs, it could be desirable to be able to "play the cheques off against each other".

Such a process is possible, but it requires certain important changes within the chequebook contract. In particular, cashing cheques can no longer be immediate and must incur a security delay, a concept familiar from other payment channel implementations.

Let us imagine a system analogous to cheques being returned to the issuer. Assume peer A issued cheques to B and the balance was brought back to zero. Later the balance tilts in A's favour but the cheques from A to B have not been cashed. In the traditional financial world, user B could either simply return the last cheque back to A or provably destroy it. In our case it is not so simple; we need some other mechanism by which *B commits not to cash* that particular cheque. Such a commitment could take several forms; it could be implemented by B signing a message allowing A to issue a

new 'last cheque' which has a lower cumulative total amount than before, or perhaps B could issue some kind of 'negative' cheque for A's chequebook that would have the effect as if a cheque with the same amount had been paid.

What all these implementations have in common, is that the chequebook can no longer allow instantaneous cashing of cheques. Upon receiving a cheque cashing request, the contract must wait to allow the other party in question to submit potentially missing information about cancelled cheques or reduced totals. To accommodate (semi-)bidirectional payments using a single chequebook we make the following modifications:

1. All cheques from user A to user B must contain a serial number.
2. Each new cheque issued by A to B must increase the serial number.
3. A's chequebook contract records the serial number of the last cheque that B cashed.
4. During the cashing delay, valid cheques with higher serial number supersede any previously submitted cheques regardless of their face value.
5. Any submitted cheque which decreases the payout of the previously submitted cheque is only valid if it is signed by the beneficiary.

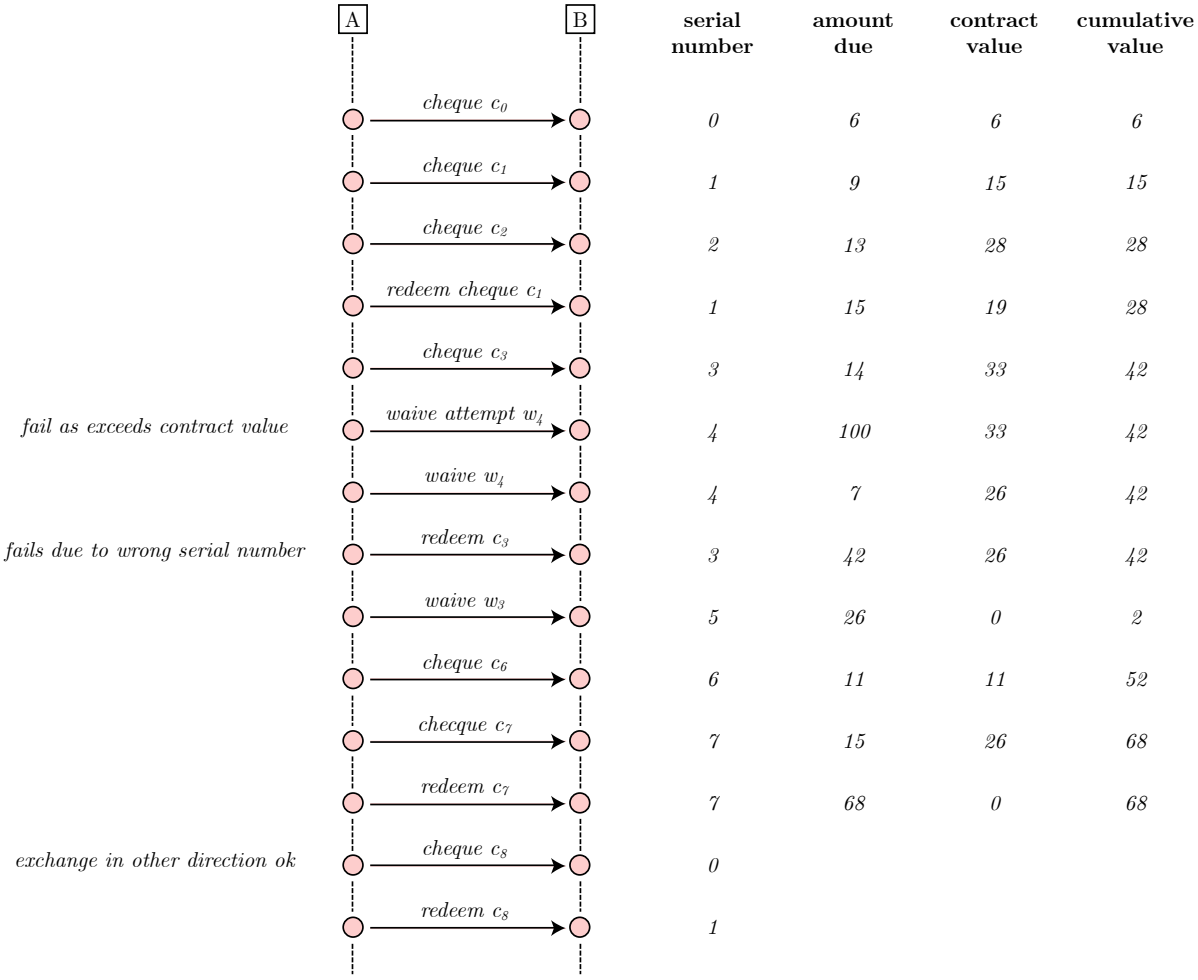
With these rules in place it is easy to see how cheque cancellation would work. Suppose user A has issued cheques $c_0 \dots c_n$ with cumulative totals $t_0 \dots t_n$ to user B. Suppose that the last cheque B cashed was c_i. The chequebook contract has recorded that B has received a payout of t_i and that the last cheque cashed had serial number i.

Let us further suppose that the balance starts tilting in A's favour by some amount x. If B had already cashed cheque c_n, then B would now have to issue a cheque of her own using B's chequebook as the source and naming A as the beneficiary. However, since cheques $c_{i+1} \dots c_n$ are un-cashed, B can instead send to A a cheque with A's chequebook as the source, B as the beneficiary, with serial number $n+1$ and cumulative total $t_{n+1} = t_n - x$. Due to the rules enumerated above, A will accept this as equivalent to a payment of amount x by B. In this scenario, instead of sending a cheque to A, B waives part of their earlier entitlement. This justifies SWAP as *send waiver as payment*.

This process can be repeated multiple times until the cumulative total is brought back to t_i. At this point all outstanding debt has effectively been cancelled and any further payments must be made in the form of a proper cheque from B's chequebook to A (see figure 27).

3.2.4 *Best effort settlement strategy*

Abels text



serial number	amount due	contract value	cumulative value
0	6	6	6
1	9	15	15
2	13	28	28
1	15	19	28
3	14	33	42
4	100	33	42
4	7	26	42
3	42	26	42
5	26	0	2
6	11	11	52
7	15	26	68
7	68	0	68
0			
1			

Figure 27: Example sequence of mixed cheques and waivers exchange

3.2.5 *Zero cash entry*

Swap accounting can also work in one direction only. If a party enters the system with zero liquid capital (a newcomer), but connects to a peer with funds (an insider), the newcomer can begin to provide a service (and not use any) in order to earn a positive Swap balance.

If the insider has a chequebook they are able to simply pay the newcomer with a cheque. However, this has a caveat: The newcomer will be able to earn cheques for services provided, but will not have the means to cash them. Cashing cheques requires sending a transaction to the blockchain, and therefore requires gas, unless the node can convince one of its peers to send the transaction for them. To facilitate this, nodes are able to sign off on a structure that they want to be sent, and then extend the Swap contract with a preprocessing step, which triggers payment to the newcomer covering the transaction's gas cost plus a service fee for the transaction's sender. The newcomer's cheque may be cashed by any insider (see figure 28). This feature justifies SWAP as *start without a penny, send with a peer*.

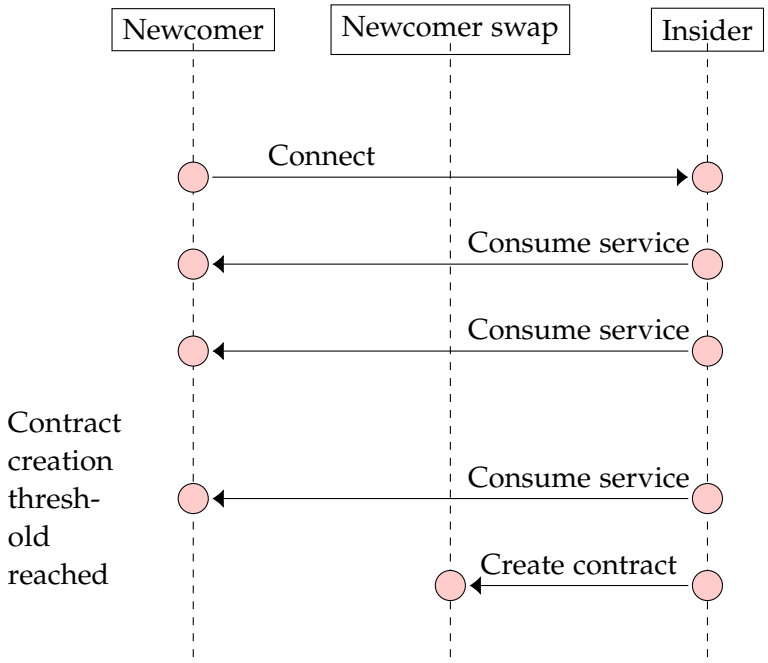


Figure 28: Bootstrapping or how to launch as a swap capable node consuming and providing a service and earn money.

The possibility to earn small amounts of money without starting capital is crucial, as it provides a way for new users to get access to Swarm without the need to purchase the token. This benefit extends to the Ethereum ecosystem in general: using Swarm, anybody can earn small amounts of money to start paying the gas to fuel their

dapps, without the need to go through a painful process of acquiring tokens prior to onboarding.

3.2.6 *Cashing out and risk of insolvency*

3.2.7 *Sanctions and blacklisting*

This section complements the SWAP scheme with additional incentives and protection against foul play.

Protocol breach

In a peer to peer trustless setting, it is difficult to have nuanced sanctions against undesired peer behaviour, however, when the basic rules of interaction are violated, the node that detects it can simply disconnect from that peer. In order to avoid deadlocks due to attempted reconnection, the sanctions imposed on transgressive nodes also include recording the peer's address into a blacklist. This simple measure is enough to provide a clear disincentive to nodes seeking to exploit the protocol.

Excessive frivolity

Both retrieval and push-sync protocols have an incentive structure where only the response incurs a source of income. Although this forms a strong incentive to play ball, it may also be necessary to take measures to ensure that nodes are not able to spam the network with frivolous requests which have no associated cost. In the case of push-syncing it is especially important not to allow chunks to expunge others at no cost. This will form the topic of a later section where we introduce postage stamps (see 3.3.1).

In the case of pull-sync retreival, the attack consists of requesting non-existing chunks and causing downstream peers to initiate a lot of network traffic, as well as some memory consumption, due to requests being persisted during the time to live period. Surely, it could happen that one requests non-existing chunks and what is more, the requested chunk could be garbage collected in the network, in which case, the requestor may have acted in good faith.

To mitigate this, each node keeps a record of the number of retrieve requests from each of its peers and then updates the relative frequency of failed requests, i.e. requests that have timed out even though the node in question has forwarded it. In the case that the proportion of these failed requests relative to successful requests is too high, sanctions are imposed on the peer: it is disconnected and blacklisted.

By remembering the requests they have forwarded, nodes can distinguish legitimate responses from a potential DoS attack: for retrieval, if the chunk delivered does not fulfil an open request, it is considered unsolicited; for push-sync, if a statement

of custody response does not match an existing entry for forwarded chunk, it is considered unsolicited.

Timeouts are crucial here. After the time to live period for a request has passed, the record of the open request can be removed and any subsequent response will therefore be treated as unsolicited, as it is indistinguishable from messages that were never requested.

To allow for some tolerance in time measurement discrepancies, once again a small percentage of illegitimate messages are allowed from a peer before they are disconnected and blacklisted.

Quality of service

Beyond the rate of unsolicited messages, nodes can cause grievances on other ways, such as by having high prices, low network throughput or long response latencies. Similarly to excessively frivolous requests, there is no need for a distinction between malicious attacks or sub-optimal (poor quality, overpriced) service provided in good faith. As a result mitigating quality of service issues is discussed in the context of peer selection strategies in forwarding ((see 9.2) and connectivity (see 9.1).

Blacklisting

Blacklisting is a strategy that complements disconnection as a measure against peers. It is supposed to extend our judgment expressed in the act of disconnection that the peer is unfit for business. In particular blacklists should be consulted when accepting incoming connections as well as in the peer suggestion strategy of the connectivity driver. On the one hand blacklisting can save the node from being deadlocked in a cycle of malicious peers trying to reconnect. On the other hand, care must be taken not to blacklist peers acting in good faith and hurt network connectivity.

3.3 STORAGE INCENTIVES

In 3.3.1, we introduce postage stamps, primarily as a measure for spam protection. Then, in 3.3.2, we turn to the postage lottery, explaining how postage stamps used for spam protection can be modified to create positive incentives for storer nodes to store files and for those uploading them to indicate their importance. In 3.3.3 we describe how the pricing mechanism can be used to signal capacity pressure of the network and how the incentives work to rectify this. Finally, 3.3.4 shows how the positive rewards provided by the postage lottery can be complemented by introducing staked insurers who stand to lose their deposit if they lose the chunks they have insured. We argue that such punitive measures are crucial in mitigating the tragedy of commons problem, which can afflict systems implementing positive storage incentives only.

3.3.1 *Spam protection with postage stamps*

Syncing involves transferring chunks from the uploader to storers, i.e. from where they enter the network, to the neighbourhood where the chunk falls within the nodes' areas of responsibility. A storer node's role is to serve content by responding to retrieve requests with chunk data. All else being equal, given Kademlia routing, the closer a node is to the chunk address, the more likely it is that a request for that chunk will end up with them. This creates a weak incentive for storer nodes to sync content. However, it presupposes that the chunk has the promise of some profit. This assumption is not warranted if an adversary can spam the network with chunks (maybe randomly generated) that are never requested. From the network's point of view, this means that useless chunks would simply replace useful ones. By attaching a cost to uploading a chunk, Swarm can mitigate such an attack.

Postage stamps

Taking inspiration from international mail delivery, the entire delivery path (as well as storage) can be pre-paid. The proof of this payment is called a postage stamp, and must be attached to the payload by the sender.

This cost need not necessarily be borne by the uploader, but they are the ones that will need to make sure a postage stamp is attached to each chunk, otherwise the upload will not succeed. Conversely, this down-payment need not necessarily be paid out as revenue to anyone in particular, i.e. it could be burnt or otherwise redistributed. The actual cost of uploading a chunk can serve as a signal of its relative importance (somewhat analogously to priority mail), that storer nodes can then use to rank chunks when selecting which ones to retain and serve, and which ones to garbage collect (see 9.7) in the event of capacity shortage.

A postage stamp is modelled as a proof of payment associated with a chunk using a witness. The witness is implemented as a digital signature issued by a third party entity who is designated by the payer.

Proof of payment

The proof of payment can lend itself to a number of different implementations. The most obvious choice is to make payments to a central postage stamp issuer smart contract on the blockchain.[4] However, because of the high transaction cost, requiring an on-chain payment for each chunk would be prohibitively expensive. Instead, we need a solution that allows the uploader to purchase the postage stamps in a postage batch and then re-use it over many chunks.

4 Using a cheque seems like one option, however, since cheques are signed against cumulative debt and assume the single beneficiary is able to reconstruct the added value over the previously sent cheque. In other words, cheques are not an appropriate means to communicate value to non-peers.

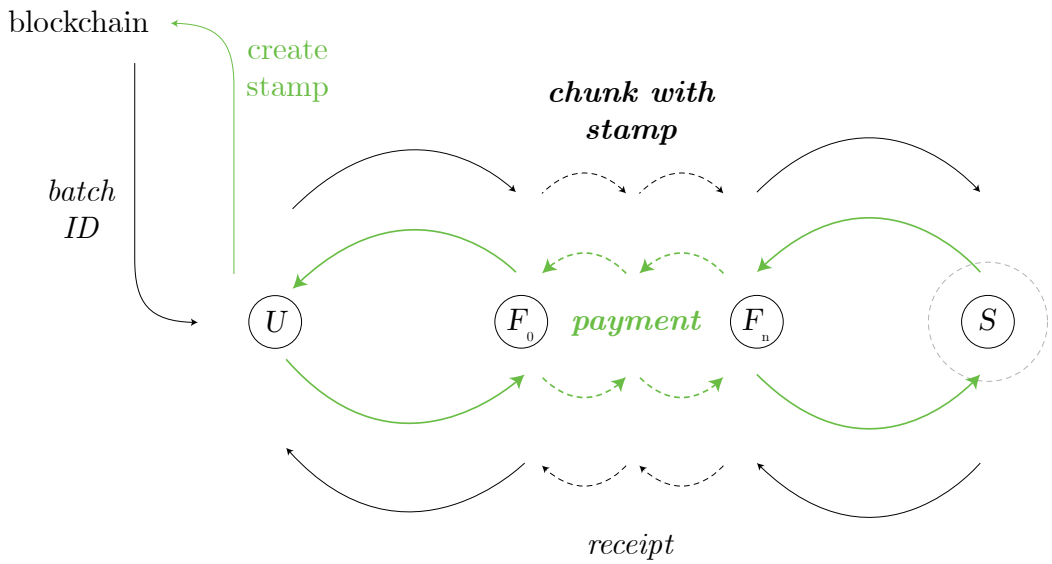


Figure 29: Postage stamps

The batches are created by a central postage smart contract when a transaction is sent to its creation endpoint, together with an amount of BZZ tokens and the following transaction data:

- *owner address* – The owner that is entitled to use the batches created to stamp chunks.
- *number of batches* – Number of batches to be created by this payment.
- *batch depth* – Logarithm of the number of chunks that can be stamped with each batch created.

This postage payment then results in the following information being recorded:

- *payment reference ID* – A random ID that is generated as reference for this payment.
- *per-chunk balance* – The total amount, equally allocated for each chunk covered by the batches created from this payment.
- *owner address* – The owner that is entitled to use the batches created to stamp chunks.
- *number of batches* – The number of batches created by this payment.
- *batch depth* – Logarithm of the number of chunks that can be stamped with each batch created.

The owner is the address that is specified in the transaction data and recorded as the party authorised to use the batches created to stamp chunks; if not specified, it is assumed to be the transaction sender by default.

A random identifier is generated to provide a reference to the payment. The payment transaction will create multiple batches the number of which is specified in the transaction. The batch depth is eventually recorded for each batch separately. New batches can be created for each payment. The number of chunks covered by this payment can then be calculated as the sum of the batch sizes. The initial batch depth is meant to apply to all batches created by the payment transaction. The amount of BZZ tokens sent with the transaction is then allocated equally to all chunks covered by the payment, i.e. the total amount of tokens, divided by the number of chunks covered is assigned to the payment ID to represent the per-chunk balance of the batches. Anyone can then choose to top up this balance at a later date.

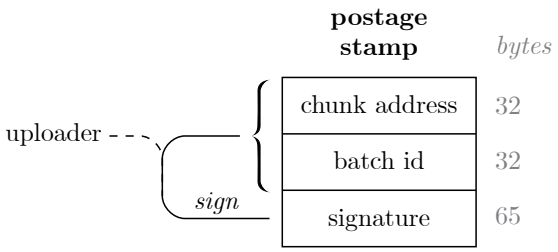


Figure 30: Postage stamp is a data structure comprised of the postage contract batch id, the chunk address and a witness signature attesting to the association of the two. Uploaders and forwarders must attach a valid postage stamp to every chunk uploaded.

The postage stamp attached to a chunk is a data structure comprising the following fields (see figure 30 and the specification in 7.7):

- *chunk address* – The address the stamp is attached to.
- *batch identifier* – Composed of a payment identifier and a batch index validity of which can be checked with the postage smart contract.
- *witness* – The owner's signature, linking the batch identifier and the owner's address.

The *value* of a postage stamp is the per-chunk balance associated with the batch. Similarly to stamps used for postal mail, a single chunk can have multiple postage stamps attached to it. In this case the value conferred to the chunk by multiple valid postage stamps are added up to form the total postage stamp value.

Validity of postage stamp

A postage stamp's validity can be checked by ensuring the following criteria are met:

- *authentic* – The batch identifier is valid, i.e. the payment ID exists and is registered, and the batch index is less than the number of batches associated with the payment ID.
- *authorised* – The witness is signed by the address specified as the owner of the batch.
- *funded* – The referenced batch has not yet exhausted its balance and is sufficiently funded to cover at least a single storage epoch for the latest price.

All this can be easily checked by the smart contract itself. Validating that less than the total amount of chunks that are allowed to be stamped by a batch are contained within it is crucial. Without further measures, there could be a possibility for an overspend attack in which the uploader reuses the stamp over more chunks than the batch size would warrant and thereby trick the unsuspecting storers into underpaid extra work.

Protection against such *overissuance* is not trivial: In the absence of global visibility of all chunks signed with a particular batch, nodes cannot directly verify the size, as they have no access to postage stamps attached to chunks that are not relayed through them. Therefore, there needs to be a way for nodes to prevent overissuance collectively, while each must determine how to act based only on locally available information.

Limiting batch size by constraining prefix collisions

A solution is to impose an explicit uniformity requirement on the batches: a constraint that chunks signed with the same batch identifier have no prefix collision longer than the depth. A postage stamp with a batch size of 2^d can be thought of as a balanced binary tree of depth d, where the leaves correspond to maximum length collision slots of the batch. If the depth of a batch is greater than the depth of the network (log of the number of nodes in the network), then all chunks matching the same collision slot are guaranteed to land in the same neighbourhoods, and, as a result, "violations" of the uniformity requirement can be locally detected by nodes (see figure 31). Storer nodes then are expected to correct the depth recorded with a batch. They can do the correction by reporting two postage stamps issued by the owner against a batch that are closer than the currently recorded depth for the batch. The incentive to do this comes from the possibility for a storer to obtain the extra storage cost above the one already paid out.

As chunk addresses are unlikely to be perfectly distributed over the collision slot of a batch, an uploader must keep more than one batch of depth d to sign 2^d chunks. In general, the most efficient utilisation of each stamp is by filling all the different collision slots (see 6.1). Put differently, continued non-uniformity will lead to underutilised

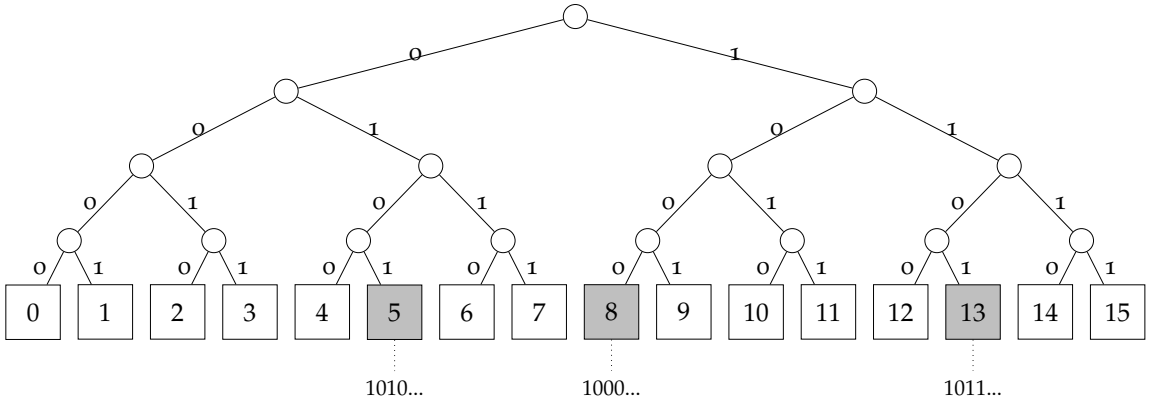


Figure 31: The postage batch is represented as a binary tree and is maps the stamps issued to the batch. The leaf nodes are at a level from the root which corresponds to the depth of the postage batch as set at the time of payment. The boxes on the leaf nodes can be thought of as the prefix collision slots. In this figure, the gray boxes represent already filled slots, so the next chunk address should be made to fall into a white one.

stamps, and therefore a higher average unit price for uploading and storing each chunk. This solution has the desired side effect that it imposes an upfront cost to non-uniform uploads: the more focused our upload is on a neighbourhood, the more slots of the postage stamps remain unused. In this way, we ensure that directing too many uploads towards a particular neighbourhood is expensive.[5] This will be significant for the later discussion of decentralised file insurance (see 5.3).

Another advantage of limiting batch size based on prefix collisions is that with prefix collisions, the absolute value of a postage stamp can be estimated. This comes in handy later when designing the postage lottery (see: 3.3.2).

In order for these collisions to be detectable, the batch size needs to be higher than the estimated neighbourhood depth over the whole lifetime of the batch. In any case, it is not in the interest of users to have batches much deeper than that, since the longer the collision prefixes, the more difficult it becomes to achieve uniformity and fill the entire batch (see A.4 of the appendix).

Mining chunks using encryption

One neat solution to completely fill up postage batches relies on the insight hinted at in 2.2.4: Choosing the encryption key allows us to *mine* a chunk to a specific postage batch.

The process of finding an encryption key to generate a content hash close to an address is analogous to mining blocks in a blockchain. Encryption of chunks requires a 32-byte key, which plays the role of the nonce in a block, that is: It provides enough entropy to guarantee that one is able to find an encryption key such that the hash digest

5 The cost of targeted attacks DoS-ing neighbourhoods is exponential with depth.

of the resulting encrypted chunk produce an address which falls into a particular slot within an open postage batch. The difficulty of mining is determined by the batch depth.

Consider the thrifty uploader that only has as many postage collision slots as chunks to send. Given a postage batch with depth d, they encrypt the chunk with a key chosen such that the resulting encrypted chunk's address fills a free collision slot of an open postage batch. As shown in the analysis found in appendix (A.4), this strategy of filling a batch requires an average of $0.69d + 1$ trials per chunk, i.e. 8, 15, 22 for a thousand, million and billion nodes respectively. This is found to be of a reasonable complexity.

3.3.2 *Postage lottery: positive incentives for storage*

As discussed in 3.2.1, the primary incentive mechanism in Swarm is providing compensation for retrieval, where nodes are rewarded for successfully having served a chunk. This reward mechanism has the added benefit of encouraging opportunistic caching. Profit-maximising storage nodes serve chunks that are often requested from them and as a result, ensure that popular content becomes widely distributed across the network and thus also the retrieval latency is decreased.

The flipside of using only this incentive, is that chunks that are rarely retrieved may end up being lost: If a chunk is not being accessed for a long time, then as a result of limited storage capacity, it will eventually end up garbage collected to make room for new arrivals. In order for the swarm to guarantee long-term availability of data, the incentive system needs to make sure that additional revenue is generated for chunks that would otherwise be deleted. In other words, unpopular chunks that do not generate sufficient profit from retrievals should compensate the nodes that store them for their opportunities forgone. The postage lottery presented in this section provides such a compensation through redistributing the revenue coming from postage stamps among the storer nodes in a fair way.

Central to the postage lottery scheme is the realisation that probabilistic payments can be used to create revenue for storer nodes. Using a lottery allows us to calibrate earnings that in the long run will have the same payout as if there was an actual payment for each chunk, yet save on transaction costs and provides the randomness which is necessary to serve as spot-checks.

Race: raffle, apply, claim and earn

The postage lottery takes place on an EVM blockchain and is managed by a smart contract. The lottery process is defined by the following protocol (see formally in 7.7).

Every N-th[6] block on the blockchain marks the beginning of a new global lottery round. The subsequent N blocks are composed of three phases representing periods during which the participants in the raffle are expected to send transactions to interact with the raffle smart contracts:

1. *pre-committal* – nodes pre-commit to the raffle challenge by sending in a price offer. As a result of valid pre-committal transactions, a set of applicants is defined. The end of this period is marked by the selection of the witness batches.
2. *submission* – nodes send a list of chunks and their proofs of custody as dictated by the witness batches. As a result of valid claim transactions, a set of claimants are defined. The end of this period is marked by the selection of challenges.
3. *refutation* – nodes send refutations of the challenges. As a result of valid refutations, a set of final prize winning claimants are determined. The end of this period is marked by the payout to the winners.

The moment the last period ends a new raffle round begins. The lottery and its phases can also be characterised by the acronym race, with, of course, an mnemonic resolution detailing the phases: raffle–apply–claim–earn (race). The timeline of events is depicted in figure 32.

Automatic raffle rounds

The RACE process is initialised with the raffle phase. First, the hash of the starting block is hashed together with integers from 0 up to $n-1$ to obtain n *winning tickets* $(r_0, \ldots r_{n-1})$ each of which represents an independent raffle draw.[7]. Once these winning tickets have been produced, nodes whose addresses are in the close proximity of the ticket r_i are then able to apply to participate in the applicable raffle.

From the network perspective, these raffles serve as spot checks on storer nodes, whereas from the storer nodes' perspective, raffles represent the means of securing compensation for providing network storage within the swarm.

Apply: precommit to raffle challenge

In the first interactive phase, in order to apply for the prize, all nodes in the proximity of the winning ticket must send a transaction to the lottery contract including a price offer for storing a chunk per block. By submitting such a price, a node implicitly declares that it has in its possession all chunks of all batches with value higher than the price within their radius of responsibility. This radius of responsibility is self-claimed by the nodes and is part of the pre-committal.

6 N is chosen so that the period between two rounds comfortably accommodates all phases of the lottery process.

7 Multiple raffles each round are needed to calibrate the expected frequency for a node to win a raffle. By increasing n as the network grows the expected time needed to get compensated for storage services can be kept constant.

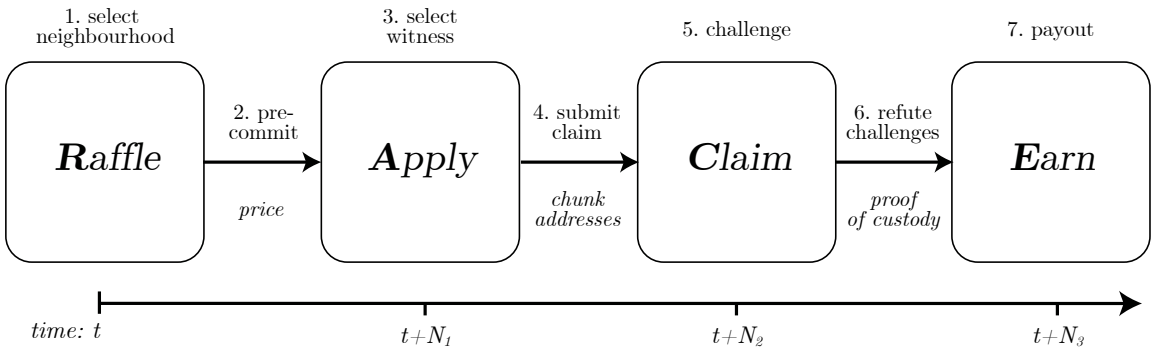


Figure 32: Timeline of events in a raffle round.

Claim: submit proofs to win

After this application deadline has passed, the raffle enters into the claim phase by designating a witness batch for each applicant. This witness batch serves as a spot check for the applicant's claim that they store all the chunks that they are responsible for. Analogous to the winning tickets, the witness batch is selected based on randomness provided by the blockhash of the block that closes the application round h_1. The spot check is implemented by the following procedure. First create an anchor by hashing together the blockhash h_1 and the applicant's address, and then choose the batch which has the id closest to the anchor and has a value above the price submitted in the application. This batch serves as the random witness batch against which applicants will submit evidence that they are still in custody of all their allocated chunks.

As we have discussed previously, stamps attached to a chunk endow that chunk with a value, as defined by the per-chunk balance of the batch. Pursuing a value-consistent garbage collection strategy (see 9.7) implies that the minimum postage stamp value accepted on a chunk will coincide with the storer node's garbage collection cutoff value. Nodes that follow this strategy faithfully, know that they have never deleted a chunk whose postage value was above their minimum required postage value. In this way, they can be confident that they are able to include all relevant chunks above that value, no matter which batch is selected to be the witness.

The claim phase is separated from the application phase exactly in order to force applicants to pre-commit to any spot check, since: if the witness was known at the time of application, frivolous nodes could simply claim the prize even if they just serendipitously stored only the chunks of the witness batch. In order to discourage such opportunistic behaviour, participation in the raffle requires a deposit sent with the precommittal. This amount is added to the reward pool if the node fails to survive the remaining phases of the raffle round, otherwise given back to the applicant.

To claim the raffle prize, a node must show that they have every witness chunk, i.e. every chunk stamped with the witness batch that falls within their radius of

responsibility. In order to prove possession of chunk data, nodes must submit a batch proof of custody, a canonically ordered list of BMT proofs (see 7.3.3), one for each witness chunk (see figure 33). Applicants have N_2 blocks to submit their claim, after which the raffle enters the challenge phase. Applicants that applied but failed to submit a claim are treated as if they had submitted an invalid claim.

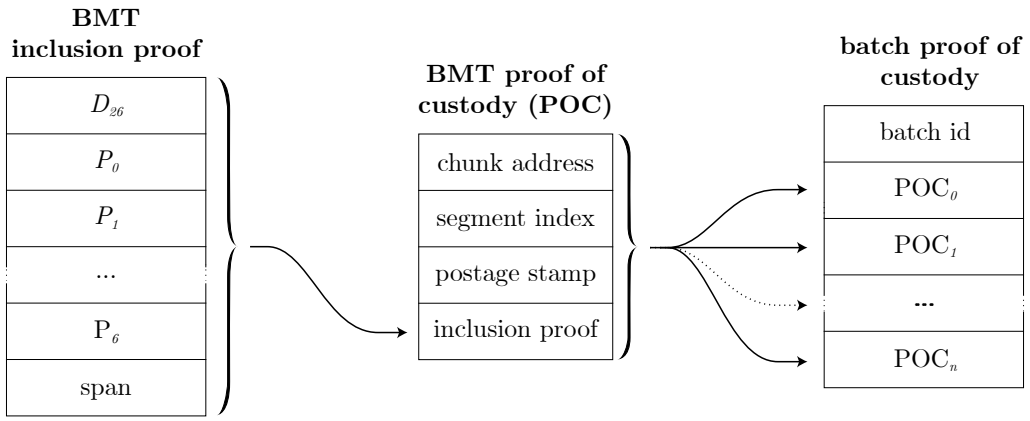


Figure 33: Batch proof of custody. A Proof of custody is BMT inclusion proof wrapped with the chunk address, the segment index and the postage stamp. Multiple POCs are packaged with a witness batch id.

Earn: challenging the applicants

Choosing the winner

After the challenge phase ended (N blocks after the beginning of the raffle round), from amongst the applicants that have survived the challenge period, for each raffle, the ones that offered the lowest prices will win the round. The price itself is the lowest price that any of the neighbourhood's winners applied with.

The lottery prize payout needs to take into account the number of winners, the number of chunks, the proportion of a nodes' storage with respect to the total volume stored in Swarm. The period that the payout is for is equally distributed among winners. For ease of reasoning, it is possible to calibrate blocks = winners, so that we can say a node when it wins gets paid for all chunks for a block. The winner is entitled to collect the price for each chunk whose price is not lower than the minimum required by the winner. The sum of all winners' proceeds is subtracted from the remaining balance of each batch. The postage batches whose balance is dried out after the payout are removed from the contract.[8]

While winning nodes could try to benefit from a higher price for a bigger win, they are competing with each other to try to win the raffle, so they are incentivised to

8 In practice, such a balance adjusment is unnecessary, see 7.7).

secure their win by offering the actual cutoff value they used for garbage collection. As a result of competitive payouts, it is expected that the unit prices will converge on the marginal cost of storing one more chunk.

Batch depth and volume adjusted payouts

So far, for ease of reasoning, we assumed that the depth of each batch was given. The requirements of batch depth are that it is deeper than the neighbourhood depth so that prefix collisions are always caught by storer nodes, yet not too much deeper so that it would make batches too difficult to fill completely (see A.4). If a batch of depth b has been filled completely, then a storer node with radius of responsibility d is expected to store 2^{b-d} chunks belonging to this batch. Half of these with proximity of order d to the winner's address.

In order to incentivise r redundant replicas of chunks, each stored at one of the nodes in a neighbourhood, winners are expected to store the same chunks. Moreover, partial storage of a batch's chunks should be discouraged. Both are taken care of if nodes are incentivised to present the most chunks they possibly can for a batch. This incentive translates to partial storage of local chunks of a match leads to lower winnings. This is achieved if each batch detected depth as well as actual size are both reported as part of the precommit. Falsely committing to more chunks than stored by the node for a batch runs the risk of an unrefutable challenge on the batch (if designated as a witness) resulting in losing the price one-time raffle ticket on top of not earning the raffle prizepool. All this aligns uploaders and storers on attempting to use completely filled batc hes (the collision slots constituting a balanced binary tree of depth b.) Since the actual sizes constitute an estimation of actual chunk volume is swarm.

The best arrangement of storer is if the In particular we propose that a volume-adjustment scheme is used when applicants

This is implicit in the require Conversely, based on the actual number of chunks per batch presented to the contract by winners, C we can make an estimate of the current neighbourhood depth in Swarm as $d' \stackrel{\text{def}}{=} b - log_2(C)$. This gives the chance for the lottery smart contract to serve as a depth oracle. The oracle value can also be used as the minimum depth for any postage batch in the raffle calculations. Note that if self-claimed depth of a batch can be lower than the typical neighbourhood depth, then nodes are expected to have at most one chunk of a batch in their storage and therefore unable to report overissued batches. Therefore active batches has the minimum size of 2^{d+1}.

Practically old completely filled postage batches can either be retained or the content they cover can be re-insured. In the former case, a node would want to retain the batch in case the network wide neighbourhood depth grows and exceeds the original depth of the batch, in which case the owner is able to add chunks to it legitimately.

In order to add new chunks to a batch without collisions, one must remember the chunks they have already associated with the batch.[9]

Note that owners can choose to pay an amount for postage that, even if the swarm grows a thousand fold, the balance would run dry only after a desired period. This becomes important in the context of upload and disappear. Such deep batches require the same $O(log(n))$ computational overhead (see A.4). This essentially means a discount requiring bulk purchase/use.

Incentives implicit in the lottery

This procedure incentivises nodes to adhere to the following behaviors:

- *stay online* – Otherwise the raffle is missed.
- *have a redundantly synced area of responsibility* – Remain fully synced with their neighbourhood. Otherwise the claim may be incomplete and the volume-adjustment leaves the node with less.
- *have a fully indexed local store* – To be able to list all chunks of a batch, nodes need to preserve postage stamps and keep an associated set of chunk addresses.
- *perform value consistent garbage collection* – In order to tell if the node locally stores all previously receipted chunks of a batch they must perform garbage collection value-consistently,[10] i.e. the minimum value accepted coincides with the maximum value ever deleted.
- *store the data* – In order to provide BMT proofs the node must possess the data in the chunk content.[11]

3.3.3 *Price signalling capacity pressure*

Storage capacity

Kademlia topology and the redundancy parameter determine the node's neighbourhood depth (see 2.2.5). The neighbourhood depth delineates the area of responsibility. As a result of postage lottery, the nodes are incentivised to have a uniform neighbourhood size. The number of chunks uploaded to that area is simply proportional to the number of all chunks uploaded to Swarm C and inversely proportional to the number of nodes in Swarm N. The number of chunks stored by a node is on average

9 In fact is actually sufficient to just remember which collision slots were filled up to a depth that could possibly be reached by the growing network. In practice, as an optimisation, potential collision slots for depth $b' > b$ can be recorded using a bit vector of length $2^{b'-1}$

10 or store indexes of deleted chunks which is inefficient

11 In the special case of addressed envelopes with pre-paid postage, no payout can be collected before the envelope has been filled and sent (see 4.4.3).

CR/N, where R is the neighbourhood replication factor, measuring the degree of redundancy. This expresses the density of chunks in any neighbourhood. Given a particular connectivity and fixed storage quantity for a neighbourhood, C/N captures the *storage capacity pressure*.

If we assume that chunks are meant to be preserved only for a limited period of time, then within some variance the pressure stays constant. In other words, we can find a network size[12] such that all content that users want preserved is preserved.

If the rate of new uploads to Swarm is higher than the rate of expiry (postage stamp balances going below the current price), then a fixed network will experience increasing pressure. As a consequence, without added capacity, after a while, content that is meant to be preserved will be garbage collected.

Such capacity shortage is solved if added storage is invited by incentivising new storer nodes to join the network or rising demand is resisted by raising prices. Conversely, if the rate of expiry is faster than new uploads, pressure decreases and there may well be surplus capacity. The resulting situation with underutilised resources is remedied if added content is invited by incentivising users to upload their content or by some nodes dropping out disincentivised by lower prices. Thinking in terms of supply and demand, we can reach a self-regulating market simply if storage capacity pressure is signalled in the price of storage, i.e. the minimum value for postage stamp per chunk.

Garbage collection strategy and postage value

The emergence of this self-regulating market is exactly what happens if we ensure that the garbage collection queue is prioritised by descending postage value (see 9.7).

Note that when a raffle round pays out, the same amounts get subtracted from all stamp balances. Therefore, the ordering by value does not change no matter how many raffle rounds have happened. However, in order to insert a newly created postage stamp, we will need to know the overall amount paid out so far and then add it to the new item's value.

A postage stamp must promise sufficient earnings with its value for the storer so that it outperforms the lowest quantile of competitors. When a node appends chunks to the bottom of the garbage collection queue, the postage value is updated by checking if there was a top-up on the blockchain.[13] If there was a change, the chunk will need to be reinserted in the queue.

12 in the sense of storage capacity; or in the sense of number of nodes assuming a minimum storage capacity per node.

13 In order to avoid checking updates on the blockchain for payout rate change, a node may just want to update the rate as a response to a payment event logged on the lottery contract.

	Chunk Address	Batch	Payment ID	Value ▲	Index
Retained	a582...	35	e752...		0
	74a9...	35	9b99...		1
	25e1...	35	3624...		2
	ae4e...	35	2a47...		3
	...	...	...	...	...
	23ba...	76	57c6...	323	79998
	0f34...	38	64a0...	299	79999
Garbage Collected	2d1a...		c903...	298	80000
			...		
	f2a6...		a729		99999

Figure 34: Garbage collection: chunks are ordered by profitability which comes from the postage value and a predicted retrieval revenue.

Combining retrieval popularity and postage value

Besides expected lottery payouts, estimation of profitability needs to factor in earnings from serving popular chunks. If we record the number of times a chunk is served for every raffle epoch, then one can apply a predictive model to these datapoints to forecast the earnings for a future period. Fortunately, we only really need to decide which are the least profitable chunks.

When we iterate over the chunks with the smallest postage value, we must decide if each chunk should survive the next epoch or if it has no real promise of being requested. The garbage collection process will then terminates once number of chunks matching the volume of a quantile have been deleted (see figure 34).

Uniformity of prices

When push-syncing chunks, each forwarding node accepts chunks only if their postage value is higher than that node's advertised minimum. Practically this means that the initial postage value cannot be lower than the maximum such value in the network, otherwise syncing is not successful. This means that nodes keeping their postage price offer for the lottery low will not receive more traffic, they just have more chance of winning and need to have more storage capacity than their more expensive neighbours

to be able to store the extra chunks whose value reached below their neighbour's minimum.

If neighbourhoods manage to keep their prices high, it will attract new nodes who will under-offer. If the high prices were genuine due to actual capacity shortage, then the new node adds storage space and corrects the non-uniformity. If the neighbourhood kept high prices because they cartellised, then the new node is able to disrupt this. Over a longer time period, therefore, prices are expected to converge.

3.3.4 *Insurance: negative incentives*

The storage incentives presented so far refer to the ability of a system to encourage the preservation of content through monetary rewards given to storers. This was achieved using a postage lottery which instrumented the fair redistribution of postage payments to storers. With this scheme, we provided positive incentivisation on a collective level. Such a system, however, is suspect to the tragedy of the commons problem in that disappearing content will have no negative consequence to any one storer node. The lack of individual accountability renders the storage incentivisation limited as a security measure against data loss. Introducing competitive insurance, on the other hand, adds an additional layer of negative incentives and forces storers to be very precise with their commitments to provide users with reliability. Particular attention is required in the design of the incentive system to make sure that failure to store every last bit promised is not only unprofitable but outright catastrophic to the insurer.

Punitive measures

Unlike in the case of bandwidth incentives where retrievals are immediately accounted and settled, long-term storage guarantees are promissory in nature and it can only be decided if the promise has been kept at the end of its validity. Loss of reputation is not sufficient as a deterrent against foul play in these instances: since new nodes must be allowed to provide services right away, cheaters could just resort to new identities and keep selling (empty) storage promises.

We need the threat of punitive measures to ensure compliance with storage promises. These will work using a *deposit system*. Nodes wanting to sell promissory storage receipts should have a stake verified and locked-in at the time of making their promise. This implies that nodes must be registered in advance with a contract and be made to put up a security deposit. Following registration, a node may sell storage promises covering the time period for which their funds are locked. While their registration is active, if they are found to have lost a chunk that was covered by their promise, they stand to lose their deposit.

Requirements

Let us start from some reasonable guiding principles:

- Owners need to express their risk preference when submitting to storage.
- Storers need to express their risk preference when committing to storage.
- There needs to be a reasonable market mechanism to match demand and supply.
- There needs to be guarantees for the owner that its content is securely stored.
- There needs to be a litigation system where storers can be charged for not keeping their promise.

Owners' risk preference consist of the time period covered as well as a preference for the *degrees of reliability*. These preferences should be specified on a per-chunk basis and they should be completely flexible at the protocol level.

Satisfying storers' risk preferences means that they have ways to express their certainty of preserving what they store and factor that in their pricing. Some nodes may not wish to provide storage guarantees that are too long term while others cannot afford to stake too big of a deposit. This differentiates nodes in their competition for service provision.

A *market mechanism* means there is flexible *price negotiation* or discovery or automatic feedback loops that tend to respond to changes in supply and demand.

In what follows we will elaborate on a class of incentive schemes we call swap, swear and swindle due to the basic components:

swap Nodes are in quasi-permanent long term contact with their registered peers. Along these connections the peers are swapping chunks and receipts triggering swap accounting (see 3.2.1).

swear Nodes registered on the Swarm network are accountable and stand to lose their deposit if they are found to violate the rules of the Swarm in an on-chain litigation process.

swindle Nodes monitor other nodes to check if they comply with their promise by submitting challenges according to a process of litigation.

Contracts through receipts

A litigation procedure necessitates that there are contractual agreements between parties ultimately linking an owner who pays for securing future availability of content and a storer who gets rewarded for preserving it and making it immediately accessible at any point in the future. The incentive structure needs to make sure that litigation is a last resort option.

The simplest solution to manage storage deals is to use direct contracts between owner and storer. This can be implemented by ensuring storers return signed receipts of chunks they accept to store and owners pay for the receipts either directly or via escrow. In the latter case, the storer is only awarded the locked funds if they are able to provide proof of storage. This procedure is analogous to the postage stamp lottery process. Insurance can be bought in the form of specially marked postage stamps, statements of custody receipts can close the loop and represent a contract between uploader and storer. Outpayments conditional on proofs of custody can be implemented the same way as the lottery.

Failure to deliver the stored content is penalised even when the consumer who tried to access the chunk but was unable, was not party to the agreement to store and provide the requested content. Litigation is therefore expected to be available to third parties wishing to retrieve content.

If the pairing of chunks and receipts is public and accessible, then consumers/downloaders (not only creators/uploaders) of content are able to litigate in case a chunk is found to be missing (see 5.3).

Registration

Before a node can sell promises of long-term storage, it must first register via a contract on the blockchain we call the SWEAR (Secure Ways of Ensuring ARchival or Swarm Enforcement And Registration) contract. The SWEAR contract allows nodes to register their public key to become accountable participants in the Swarm. Registration is done by sending the deposit to the SWEAR contract, which serves as collateral in case the terms that registered nodes "swear" to keep are violated (i.e. nodes do not keep their promise to store). The *registration* is valid only for a set period, at the end of which a Swarm node is entitled to their deposit. Users of Swarm should be able to count on the loss of deposit as a disincentive against foul play for as long as enrolled status is granted. Because of this the deposit must not be refunded before the registration expires. The expiry of the insurance period should therefore include a final period during which the node is not allowed to issue new receipts but can still be challenged.

When a registered insurer node receives a request to store a chunk that is closest to them, it can acknowledge it with a signed receipt. It is these signed receipts that are used to enforce penalties for loss of content. Because of the locked collateral backing them, the receipts can be viewed as secured promises for storing and serving a particular chunk.

Submitting a challenge

If a node fails to observe the rules of the Swarm they swear to keep, the punitive measures need to be enforced which is necessarily preceded by a litigation procedure. The implementation of this process is called SWINDLE (Secured With INsurance Deposit Litigation and Escrow).

When a user attempts to retrieve insured content and fails to find a chunk, they can report the loss by submitting a challenge. This scenario is the typical context for starting litigation. This is analogous to a court case in which the issuers of the receipts are the defendants who are guilty until proven innocent. Similarly to a court procedure, public litigation on the blockchain should be a last resort when the rules have been abused despite the deterrents and positive incentives.

The challenge takes the form of a transaction sent to the SWINDLE contract in which the challenger presents the receipt(s) for the lost chunk. Any node is allowed to send a challenge for a chunk as long as they have a valid receipt for it (although it may not have necessarily been issued to them). The same transaction also sends a deposit covering the price of the upload of a chunk. The validity of the challenge as well as its refutation need to be easily verifiable by the contract. The contract verifies if the receipt is valid, i.e. 1) authentic, 2) active and 3) funded, by checking the following conditions:

- *authentic* – The receipt was signed with the public key of a registered node.
- *active* – The expiry date of the receipt has not passed.
- *funded* – Sufficient funds are sent alongside it to compensate the peer for uploading the chunk in case of a refuted challenge.

The last point above is designed to disincentivise frivolous litigation, i.e. bombarding the blockchain with bogus challenges and potentially causing a DoS attack.

The contract comes with an accessor for checking that a given node is challenged (potentially liable for penalty), so the challenged nodes can be notified that they must present the chunk in a timely fashion. The challenge is then kept open for a fixed amount of time, the end of which essentially is the deadline to refute the challenge.

Fingerpointing or proof of storage

The node implicated can refute the challenge by sending a transaction to the blockchain with either the direct refutation (a proof of custody or the chunk itself depending on the size) or a receipt for the same chunk signed by another node. This receipt needs to be issued by a nearer neighbour (a registered peer closer to the chunk address than the node itself). In other words, if a node is accused with a receipt, they can shift the blame and provide a valid receipt from a nearer neighbour.[14] Thus litigation can trigger a chain of challenges with receipts pointing from the initially challenged node all the way to a node that can shift the blame no further and therefore must present the chunk or be punished. This way of refuting a challenge is called fingerpointing.

Upon verifying the format of the refutation, the contract checks its validity by checking the hash of the chunk payload against the hash that is litigated or validating the proof of custody.

14 The contract can easily verify if the newly challenged node (the one that signed the receipt submitted as a refutation) is closer to the chunk address than the originally challenged node.

If a challenge is refuted within the period the challenge is open, the deposit of the accused node remains untouched. The cost of uploading the chunk must be reimbursed to the uploader from the deposit of the challenge, but in order to prevent DoS attacks, this deposit should actually be substantially higher than this in any case (e.g. a small integer multiple of the corresponding gas price). After successful refutation the challenge is cleared from the blockchain state.

This challenge scheme is the simplest way (1) for the defendants to refute the challenge as well as (2) to make the actual data available for the nodes that needs it.

Successful challenge and enforcement

If the deadline passes without successful refutation of the challenge, then the charge is regarded as proven and the case enters into the enforcement stage. Nodes that are proven guilty of losing a chunk lose their deposit. Enforcement is guaranteed to be successful by the fact that deposits are kept locked up in the SWEAR contract.

If, on litigation, it turns out that a chunk (that was covered by a receipt) was lost, the deposit must be at least partly *burned*. Note that this is necessary because, if penalties were paid out as compensation to holders of receipts of lost chunks, it would provide an avenue of early exit for a registered node by "losing" bogus chunks that had been deposited by colluding users. Since users of Swarm are interested in their information being reliably stored, their primary incentive for keeping the receipts is to keep the swarm motivated to do so, not the potential for compensation in the case they do not. If deposits are substantial, we can get away with paying out compensation for initiating litigation, however we must have the majority (say 95%) of the deposit burned in order to make sure this easy exit route remains closed.

Punishment can entail *suspension*, meaning a node found guilty is no longer considered a registered Swarm node. Such a node is only able to resume selling storage receipts once they create a new identity and put up a deposit once again. Note that the stored chunks are in the proximity of the address, so having to create a new identity will also imply expending extra bandwidth to replenish storage. This is an extra pain inflicted on offending nodes.

Deposit

Another important decision to take is whether maximum deposits staked for a single chunk should vary independently of price. It is hard to conceptualise what this would mean in the first place. Assume that a nodes' deposit varies and affects the probability that they are chosen as storers: a peer is chosen whose deposit is higher out of two advertising the same price. In this case, the nodes have an incentive to up the ante, and start a bidding war. In the case of normal operation, this bidding would not be measuring confidence in quality of service but would simply reflect the wealth of the prospective storers. We conclude that prices should be variable and entirely up to the node, but higher confidence or certainty should also be reflected directly in the

amount of deposit that they stake: deposit staked per chunk should be a constant multiple of the price.

Assuming s is a system-wide security constant dictating the ratio between price and deposit staked in case of loss, for an advertised price of p, the minimum deposit is $d = s \cdot p$. Price per chunk per epoch is freely configurable and dictated by supply and demand in the free market. Nodes are therefore free to follow any price oracle or form cartels agreeing on price.

Incentivising promissory services

Delayed payments without locked funds leave storers vulnerable to non-payment. Advance payments (i.e. payments settled at the time of contracting, not after the storage period ends) on the other hand, leave the buyers vulnerable to cheating. Without limiting the total value of receipts that nodes can sell, a malicious node can collect more than their deposit and disappear. Having forfeited their deposit, they still walk away with a profit even though they broke their promise. Given a network size and a relatively steady demand for insured storage, the deposit could be set sufficiently high so this attack is no longer economical.

Locking the entire amount eliminates the storer's distrust due to potential insolvency of the insured party. When paying for insurance, the funds should cover the total price of storing a chunk for the entire storage period. This amount is locked and is released in installments contingent on the condition that the node provides a proof of custody. On the other hand, since payment is delayed it is no longer possible to collect funds before the work is complete, which eliminates a collect-and-run attack entirely.

3.4 SUMMARY

In the first two chapters of the architecture part of the book we introduced the core of swarm: the peer to peer network layer described in chapter 2 implements a distributed immutable storage for chunks which is complemented by the incentive system described in the following chapter. The resulting base layer system provides:

1. permissionless participation and access,
2. zero cash entry for node operators,
3. maximum resource utilisation,
4. load-balanced distribution of data,
5. scalability,
6. censorship resistance and privacy for storage and retrieval,
7. auto-scaling popular content,

8. basic plausible deniability and confidentiality,
9. churn resistance and eventual consistency in a dynamic network with node dropouts,
10. sustainability without intervention due to built-in economic incentives,
11. robust private peer-to-peer accounting,
12. incentivised bandwidth sharing,
13. off-chain micro-commitments with on-chain settlement,
14. DoS resistance and spam protection,
15. positive (i.e., motivated by reward) incentives for storage,
16. negative (i.e., discouraged through threat of punitive measures) incentives against data loss.

4

HIGH-LEVEL FUNCTIONALITY

This chapter is building on the distributed chunk store and introduces data structures and processes enabling higher level functionality to offer a rich experience handling data. In particular we show how chunks can be organised to represent files (4.1.1), how files can be organised to represent collections (4.1.2), introduce key–value maps (4.1.4) and then briefly discuss the possibility of arbitrary functional data structures. We then turn to giving our solution to providing confidentiality and access control (4.2).

In 4.3, we introduce Swarm feeds, which are suitable for representing a wide variety of sequential data, such as versioning updates of a mutable resource or indexing messages for real-time data exchange: offering a system of persisted pull messaging. To implement push-notifications of all kinds, 4.4 introduces the novel concept of Trojan chunks that allows messages to be disguised as chunks be directed to their desired recipient in the swarm. We explain how trojan chunks and feeds can be used together to form a fully fledged communication system with very strong privacy features.

4.1 DATA STRUCTURES

In the first two chapters, we assumed that data is in the form of chunks, i.e. fixed size data blobs. We now present the algorithms and structures which make it possible to represent data of arbitrary length. We then introduce Swarm manifests which form the basis of representing collections, indexes, and routing tables allowing Swarm to host websites and offer URL-based addressing.

4.1.1 *Files and the Swarm hash*

In this section we introduce the *Swarm hash* which is a means to combine chunks to represent larger sets of structured data such as files. The idea behind the Swarm hashing algorithm is that chunks can be arranged in a Merkle tree such that leaf nodes correspond to chunks from consecutive segments of input data, while intermediate nodes correspond to chunks which are composed of the chunk references of their children, packaged together to form another chunk (see 35).

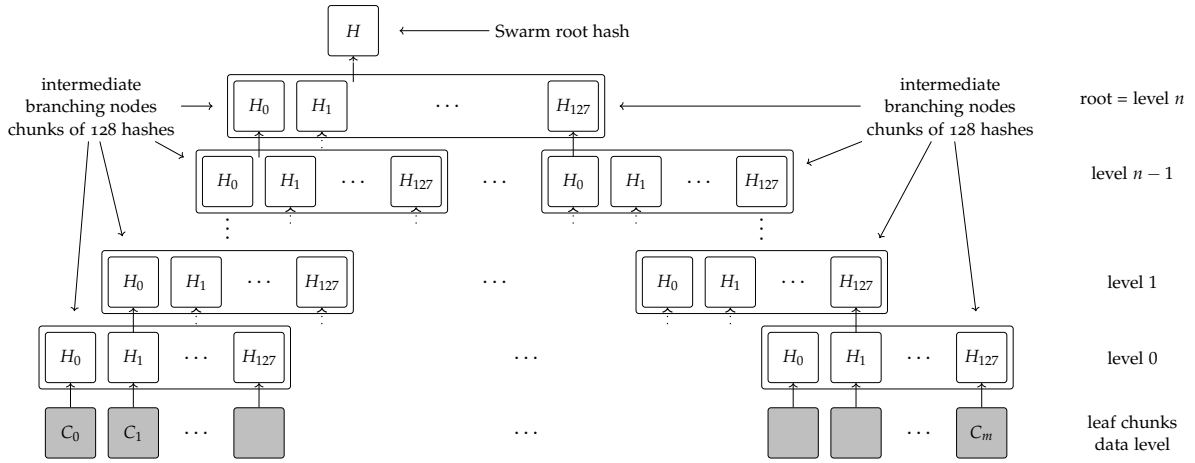


Figure 35: Swarm hash: data input is segmented to 4-kilobyte chunks (gray), that are BMT hashed. Their hashes are packaged into intermediate chunks starting on level 0, all the way until a single chunk remains on level n.

Branching factor and reference size

The branching factor of the tree is calculated as the chunk size divided by the reference size. In the case of unencrypted content, the chunk reference is simply the BMT hash of the chunk (see 7.3.3) which is 32 bytes, so the branching factor is just 4096/32 = 128. A group of chunks referenced under an intermediate node is referred to as a batch. If the content is encrypted, the chunk reference becomes the concatenation of the chunk hash and the decryption key. Both are 32 bytes long so an encrypted chunk reference will be 64 bytes, and therefore the branching factor is 64.

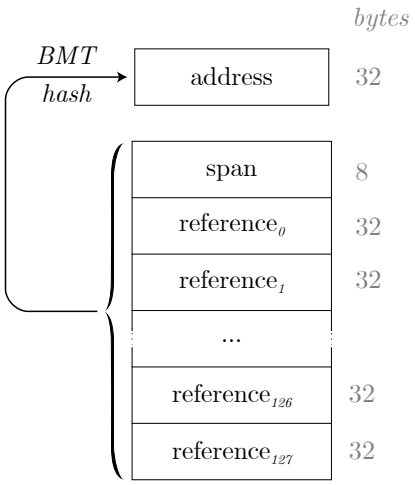


Figure 36: Intermediate chunk. It encapsulates references to its children.

Thus, a single chunk can represent an intermediate node in the Swarm hash tree, in which case, its content can be segmented to references allowing retrieval of their children which themselves may be intermediate chunks, see figure 36. By recursively unpacking these from the root chunk down, we can arrive at a sequence of data chunks.

Chunk span and integrity of depth

The length of data subsumed under an intermediate chunk is called chunk span. In order to be able to tell if a chunk is a data chunk or not, the chunk span, in a 64-bit little endian binary representation is prepended to the chunk data. When calculating the BMT hash of a chunk, this span constitutes the metadata that needs to be prepended to the BMT root and hashed together to give us the chunk address. When assembling a file starting from a hash, one can tell if a chunk is a data chunk or an intermediate chunk simply by looking at the span: if the span is larger than 4K, the chunk is an intermediate chunk and its content needs to be interpreted as a series of hashes of its children; otherwise it is a data chunk.

In theory, if the length of the file is already known, spans of intermediate chunks are unnecessary since we could calculate the number of intermediate levels needed for the tree. However, using spans disallows reinstating the intermediate levels as data layerers. In this way, we impose *integrity of the depth*.

Appending and resuming aborted uploads

The Swarm hash has the interesting property that any data span corresponding to an intermediate chunk is also a file and can therefore be referenced as if the intermediate chunk was its root hash. This has significance because it allows for appending to a file while retaining historical reference to the earlier state and without duplicating chunks, apart from on the incomplete right edge of the Merkle tree. Appending is also relevant for resuming uploads upon crashing in the middle of uploading big files.

Random access

Note that all chunks in a file except for the right edge are completely filled. Since chunks are of fixed size, for any arbitrary data offset one can calculate the path to reach the chunk, including the offset to search within that chunk, in advance. Because of this, *random access to files* is supported right away (see figure 37).

Compact inclusion proofs for files

Suppose we were to prove the inclusion of a substring in a file at a particular offset. We saw that the offset applied to the data maps to a deterministic path traversing the Swarm hash. Since a substring inclusion proof simply reduces to a series of proofs of data segment paths, the chunk addresses are a result of a BMT hash where the

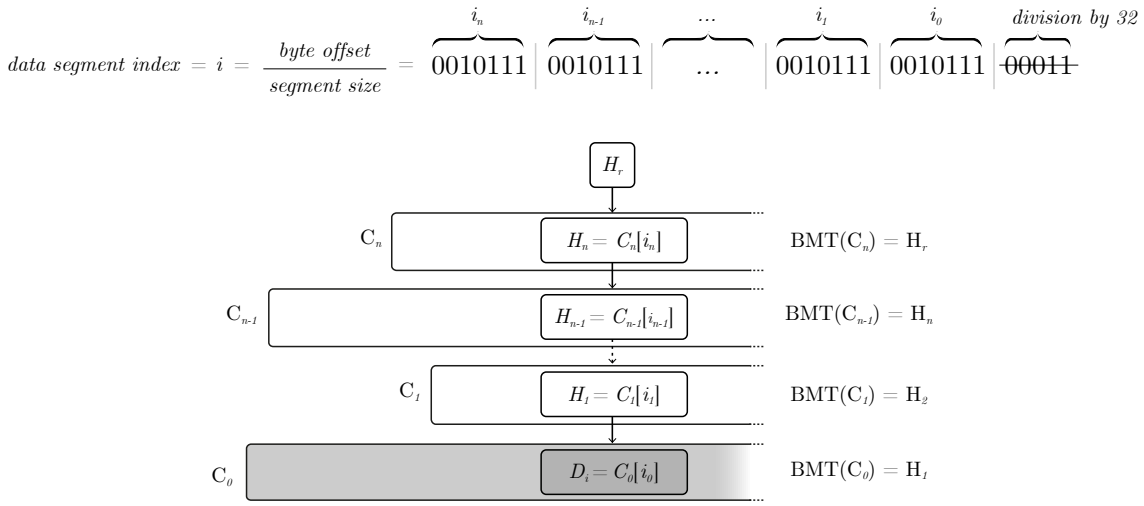


Figure 37: Random access at arbitrary offset with Swarm hash. The arbitrary offset informs us how to traverse the Swarm hash tree.

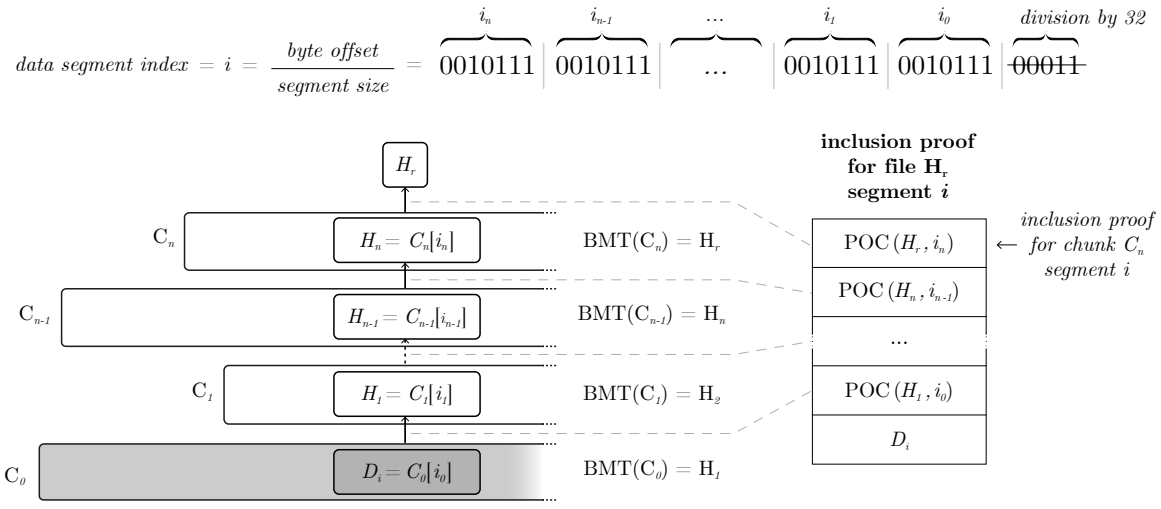


Figure 38: Compact inclusion proofs for files. If we need to prove inclusion of segment i, after division by 32 (within-segment position), we follow groups of 7 bits to find the respective segment of the intermediate node.

base segments are 32-byte long. This means that in intermediate chunks, BMT base segments align with the addresses of children. As a consequence, proving that a the child of an intermediate chunk at a particular span offset is equivalent to giving a segment inclusion proof on the child hash. Therefore, substring inclusion in files can be proved with a sequence of BMT inclusion proofs (see 7.3.3) where the length of the sequence corresponds to the depth of the Swarm hash tree (see figure 38).

Note that such inclusion proofs are possible even in the case of encrypted data since the decryption key at for a segment position can be selectively disclosed without revealing any information that could compromise the encryption elsewhere in the chunk.

In this section, we presented Swarm hash, a data structure over chunks that represents files, which supports the following functionalities:

- *random access* – The file can be read from any arbitrary offset with no extra cost.
- *append* – Supports append without duplication.
- *length preserving edits* – Supports length preserving edits without duplication of unmodified parts.
- *compact inclusion proofs* – Allow inclusion proofs with resolution of 32 bytes in space logarithmic in file size.

4.1.2 *Collections and manifests*

The Swarm manifest is a structure that defines a mapping between arbitrary paths and files to represent collections. It also contains metadata associated with the collection and its objects (files). A manifest entry contains a reference to a file, more precisely a reference to the Swarm root chunk of the representation of file (see 4.1.1) and also specifies the media mime type of file so that browsers will know how to handle it. You can think of a manifest as (1) a routing table, (2) a directory tree, or (3) an index, which makes it possible for Swarm to implement (1) web sites, (2) file-system directories, or (3) key–value stores (see 4.1.4), respectively. Manifests provide the main mechanism to enable URL based addressing in Swarm (see 4.1.3).

Manifests are represented as a compacted trie[1] in which individual trie nodes are serialised as chunks (see 7.4.2). The paths are associated with a manifest entry that specifies at least the *reference*. The reference may then point to an embedded manifest if the path is a common prefix of more than one path in the collection, thereby implementing branching in the trie, see figure 39.

A manifest entry is a data structure that encapsulates all metadata about a file or directory. The information minimally includes a swarm reference to the file, complemented with file information relevant either as a (1) parameter for the downloader

1 see https://en.wikipedia.org/wiki/Trie

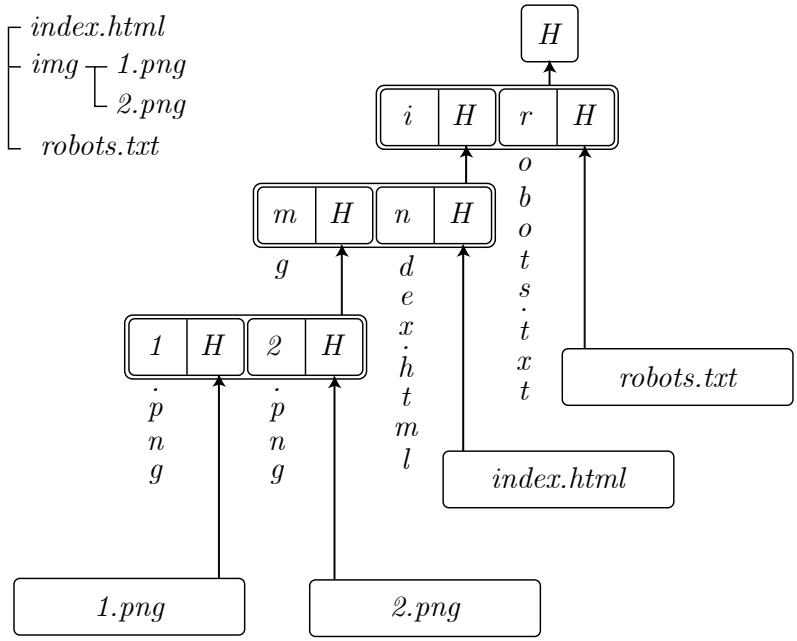


Figure 39: Manifest structure. Nodes represent a generic trie node: it contains the forks which describe continuations sharing a prefix. Forks are indexed by the next byte of the key, the value of which contains the Swarm reference to the child node as well as the longest prefix (compaction).

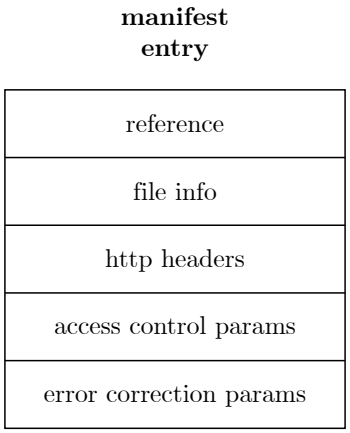


Figure 40: Manifest entry is a data structure that contains the reference to a file including metadata about a file or directory pertaining to the assembler, access control and http headers.

component which assembles chunks into a byte stream or (2) for the client side rendering handled by the browser or (3) for the mapping of manifests to file system directory tree. (1) is exemplified by access control information and erasure coding parameters but also the publisher needed for doing chunk recovery. (2) includes content type headers, or generically HTTP headers that the local swarm client will pick them up by the API and set in the response header when the file is retrieved. and (3) file info mapped to file system when downloading such as file permissions.

The high level API (see 10.2) to the manifests provides functionality to upload and download files and directories. It also provides an interface to add documents to a collection on a path and to 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, i.e. the value that was referenced in the deleted manifest entry remains in Swarm. Swarm exposes the manifest API via the *bzz URL scheme* (see 10.2).

4.1.3 *URL-based addressing and name resolution*

Earlier we introduced the low level network component of Swarm as a distributed immutable store of chunks (DISC, see 2.2.1). In the previous two sections we presented ways in which files (4.1.1) and collections (4.1.2) can be represented in Swarm and referenced using chunk references. Manifests provide a way to index individual documents in a collection and this allows them to be considered a representation of web sites hosted in Swarm. The root manifests serve as the entry-point to virtually hosted sites on Swarm and are therefore analogous to hosting servers. In the current web, domain names resolve to the IP address of the host server, and the URL paths (of static sites) map to entries in the directory tree based on their path relative to the document root set for the host. Analogously, in Swarm, domain names resolve to a reference to the root manifest and the URL paths map to manifest entries based on their path.

When the HTTP API serves a URL, the following steps are performed:

1. *domain name resolution* – Swarm resolves the host part to a reference to a root manifest,
2. *manifest traversal* – recursively traverse embedded manifests along the path matching the URL path to arrive at a manifest entry,
3. *serving the file* – the file referenced in the manifest entry is retrieved and rendered in the browser with headers (notably content type) taken from the metadata of manifest entry.

Swarm supports domain name resolution using the Ethereum Name Service (ENS). ENS is the system that, analogously to the DNS of the old web, translates human-

readable names into system-specific identifiers, i.e. a reference in the case of Swarm.[2] In order to use ENS, the Swarm node needs to be connected to an EVM-based blockchain supporting the Ethereum API (ETH mainnet, Ropsten, ETC, etc). Users of ENS can register a domain name on the blockchain and set it to resolve to a reference. This reference is most commonly the content hash of a public (unencrypted) manifest root. In the case that this manifest represents a directory containing the assets of a website, the default path for the hash may be set to be the desired root html page. When an ENS name is navigated to using a Swarm enabled browser or gateway, Swarm will simply render the root html page and Swarm will provide the rest of the assets provided in the relative path. In this way, users are able to very easily host websites, and Swarm provides an interface to older pre-existing browsers, as well as implementing a decentralised improvement over DNS.

4.1.4 *Maps and key–value stores*

This section describes two ways of implementing a simple distributed key–value store in Swarm. Both rely solely on tools and APIs which have been already introduced.

One technique is by using manifests: Paths represent keys and the reference in the manifest entry with the particular path point to the value. This approach benefits from a full API enabling insert, update and remove through the bzz manifest API (see 10.2). Since manifests are structured as a compacted trie, this key–value store is scalable. Index metadata requires storage logarithmic to the number of key–value pairs. Lookup requires logarithmic bandwidth. The data structure allows for iteration that respects key order.

Single-owner chunks also provide a way to define a key–value store.

This other technique simply posits that the index of the single owner chunk be constructed as a concatenation of the hash of the database name and the key. This structure only provides insert, no update or remove. Both insert and lookup are constant space and bandwidth. However, lookup is not safe against false negatives, i.e., if the chunk representing the key–value pair is not found, this does not mean it has never been created (e.g. it could have been garbage collected). Thus, the single owner chunk based key–value store is best used as (1) a bounded cache of recomputable values, (2) mapping between representations such as a translation between a Swarm hash and a Keccak256 hash as used in the Ethereum blockchain state trie nodes, or (3) conventional relational links, such as likes, upvotes and comments on a social media post.

4.2 ACCESS CONTROL

This section first addresses the confidentiality of content using encryption. Encryption becomes especially useful once users are provided ways to manage other's access

2 RNS, name service for RIF OS on RSK is also supported.

to restricted content. Use cases include managing private shared content as well as authorising access to a member's area of a web application. In this way we provide a robust and simple API to manage access control, something that is traditionally handled through centralised gate-keeping which is subject to frequent and disastrous security breaches.

4.2.1 *Encryption*

This section describes how to achieve confidentiality in a distributed public data storage. In this way, we show how to fulfill the natural requirement for many use cases to store private information and ensure that this is accessible only to specific authorised parties using Swarm.

It is clear that the pseudo-confidentiality provided by the server-based access control predominantly used in current web applications is inadequate. In Swarm, nodes are expected to share the chunks with other nodes, in fact, storers of chunks are incentivised to serve them to anyone who requests them and therefore it is infeasible for nodes to act as the gatekeepers trusted with controlling access. Moreover, since every node could potentially be a storer, the confidentiality solution must leak nothing that allows third party storers to distinguish a private chunk from random data. As a consequence of this, the only way to prevent unauthorized parties from accessing private chunks is by using encryption. In Swarm, if a requestor is authorized to access a chunk, they must be in possession of a decryption key that can be used to decrypt the chunk, while unauthorized parties must not. Incidentally, this also serves as the basis for plausible deniability.

Encryption at the chunk level is described in 2.2.4 and formally specified in 7.3.4. It has the desirable property that it is virtually independent of the chunk store layer, with the exact same underlying infrastructure for storing and retrieving chunks as the case of unencrypted content. The only difference between accessing private and public data is the presence of a decryption/encryption key in the chunk references (see 7.4.1) and the associated minor cryptographic computational overhead.

The storage API's raw `GET` endpoint allows both encrypted and unencrypted chunk references. Decryption is triggered if the chunk reference is double size; consisting of the address of the encrypted chunk and a decryption key. Using the address, the encrypted chunk is retrieved, stored and decrypted using the supplied decryption key. The API responds with the resulting plaintext.

The storage API's `POST` endpoint expects users to indicate if they want to have encryption on the upload or not. In both cases the chunk will be stored and push-synced to the network, but if encryption is desired, the encrypted chunk needs to be created first. If no further context is given, a random encryption key is generated which is used as a seed to generate random padding to fill the chunk up to a complete 4096 bytes if needed, and finally this plaintext is encrypted with the key. In the case of

encryption, the API `POST` call returns the Swarm reference consisting of the Swarm hash as chunk address and the encryption key.

In order to guarantee the uniqueness of encryption keys as well as to ease the load on the OS's entropy pool, it is recommended (but not required) to generate the key as the MAC of the plaintext using a (semi-) permanent random key stored in memory. This key can be permanent and generated using `scrypt` [Percival, 2009] with a password supplied upon startup. Instead of the plaintext, a namespace and path of the manifest entry can be used as context. This use of a key derivation function has the consequence that chunk encryption will be deterministic as long as the context is the same: If we exchange one byte of a file and encrypt it with the same context, all data chunks of the file except the one that was modified will end up being encrypted exactly as the original (see 7.3.4). Encryption is therefore deduplication friendly.

4.2.2 *Managing access*

This section describes the process the client needs to follow in order to obtain the full reference to the encrypted content. This protocol needs basic meta-information which is simply encoded as plaintext metadata and explicitly included in the root manifest entry for a document. This non-privileged access is called root access.

In contrast, granted access is a type of selective access requiring root access as well as access credentials: an authorised private key or passphrase. Granted access gives differentiated privileges for accessing the content by multiple parties sharing the same root access. This allows for updating the content without changing access credentials. Granted access is implemented using an additional layer of encryption on references.

The symmetric encryption of the reference is called the encrypted reference, the symmetric key used in this layer is called the access key.

In the case of granted access, the root access meta-information contains both the encrypted reference and the additional information required for obtaining the access key using the access credentials. Once the access key is obtained, the reference to the content is obtained by decrypting the encrypted reference with the access key, resulting in the full reference composed of the address root chunk and the decryption key for the root chunk. The requested data can then be retrieved and decrypted using the normal method.

The access key can be obtained from a variety of sources, three of which we will define.

First, a session key is derived from the provided credentials. In the case of granting access to a single party, the session key is used directly as the access key, see figure 41. In the case of multiple parties, an additional mechanism is used for turning the session key into the access key.

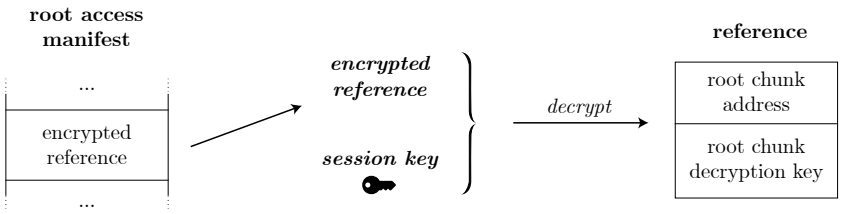


Figure 41: Access key as session key for single party access.

Passphrase

The simplest credential is a *passphrase*. The session key is derived from a passphrase using `scrypt` with parameters that are specified within the root access meta-information. The output of scrypt is a 32-byte key that may be directly used for Swarm encryption and decryption algorithms.

In typical use cases, the passphrase is distributed by an off-band means with adequate security measures, or exchanged in person. Any user knowing the passphrase from which the key was derived will be able to access the content.

Asymmetric derivation

A more sophisticated credential is a *private key*, identical to those used throughout Ethereum for accessing accounts, i.e. an elliptic curve using secp256k1. In order to obtain the session key, an elliptic curve Diffie-Hellman (ECDH) key agreement must be performed between the content publisher and the grantee. The resulting shared secret is hashed together with a salt. The content publisher's public key as well as the salt are included among metadata in the root access manifest. It follows from the standard assumptions of ECDH, that this session key can only be computed by the publisher and the grantee and no-one else. Once again, if access is granted to a single public key, the session key derived this way can be directly used as the access key which allows for the decryption of the encrypted reference. Figure 42 summarises the use of credentials to derive the session key.

4.2.3 *Selective access to multiple parties*

In order to manage access by multiple parties to the same content, an additional layer is introduced to obtain the access key from the session key. In this variant, grantees can be authenticated using either type of credentials, however, the session key derived as described above is not used directly as the access key to decrypt the reference. Instead, two keys: a lookup key and an access key decryption key, are derived from it by hashing it with two different constants (0 and 1, respectively).

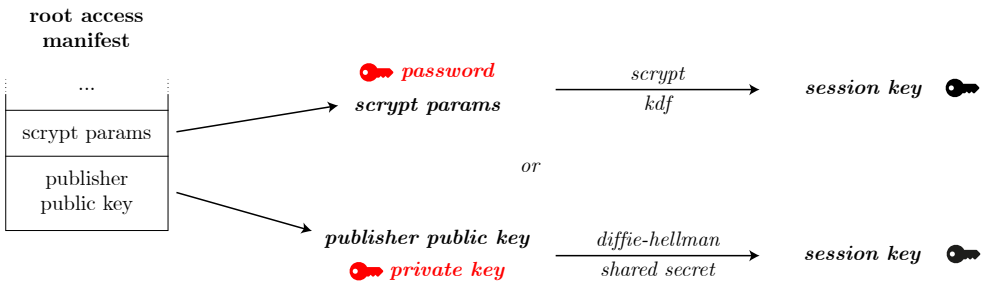


Figure 42: Credentials to derive session key.

When granting access, the publisher needs to generate a global access key to encrypt the full reference and then encrypt it with the access key decryption keys for each grantee. Thereafter, a lookup table is created, mapping each grantees lookup key to their encrypted access key. Then, for each lookup key, the access key is encrypted with the corresponding access key decryption key.

This lookup table is implemented as an access control trie (ACT) in Swarm manifest format with paths corresponding to lookup keys and manifest entries containing the ciphertext of the encrypted access keys as metadata attribute values. The ACT manifest is an independent resource referenced by a URL which is included among the root access metadata so that users know whether or not an ACT is to be used. Its exact format is specified in 7.5.

When accessing content, the user retrieves the root access meta data, identifies the ACT resource and then calculates their session key using either their passphrase and the scrypt parameters or the publisher public key and their private key and a salt. From the session key they can derive the lookup key by hashing it with 0 and then retrieve the manifest entry from ACT. For this they will need to know the root of the ACT manifest and then use the lookup key as the URL path. If the entry exists, the user takes the value of the access key attribute in the form of a ciphertext that is decrypted with a key derived by hashing the session key with the constant 1. The resulting access key can then be used to decrypt the encrypted reference included in the root access manifest, see figure 43. Once we unlock the manifest root, all references contain the decryption key.

This access control scheme has a number of desirable properties:

- Checking and looking up one's own access is logarithmic in the size of the ACT.
- The size of the ACT merely provides an upper bound on the number of grantees, but does not disclose any information beyond this upper bound about the set of grantees to third parties. Even those included in the ACT can only learn that they are grantees, but obtain no information about other grantees beyond an upper bound on their number.

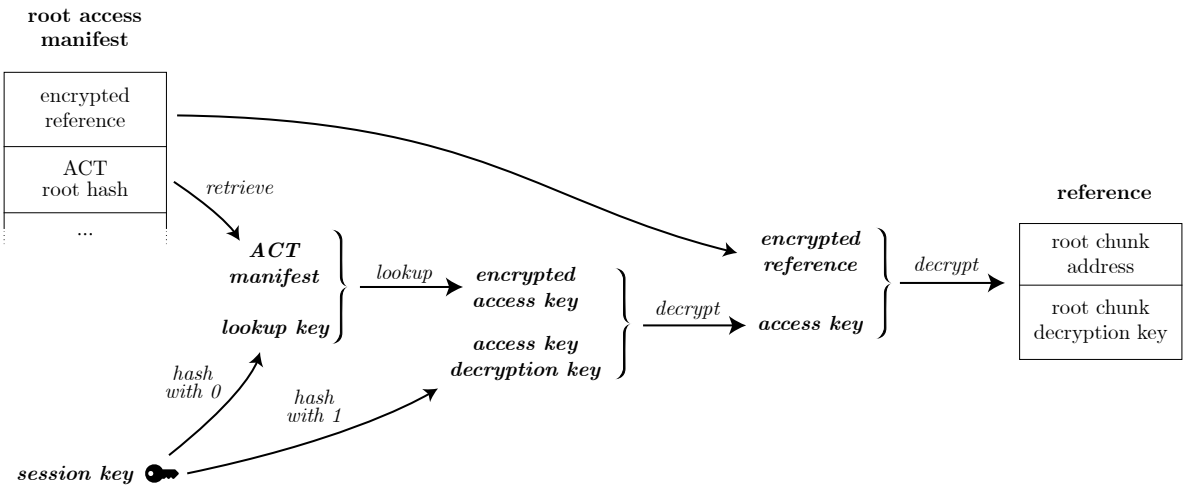


Figure 43: Access control for multiple grantees involves an additional layer to get from the session key to the access key. Each user must lookup the global access key specifically encrypted to them. Both the key to look up and the key to decrypt the access key are derived from the session key which in turn requires their credentials.

- Granting access to an additional key requires extending the ACT by a single entry, which is logarithmic in the size of the ACT.

- Withdrawing access requires a change in the access key and therefore the rebuilding of the ACT. Note that this also requires that the publisher retain a record of the public keys of the grantees after the initial ACT is created.

4.2.4 *Access hierarchy*

In the simplest case, the access key is a symmetric key. However, this is just a special case of the more flexible solution, where the access key consists of a symmetric key and a key derivation path by which it is derived from a root key. In this case, in addition to the encrypted reference, a derivation path may also be included. Any party with an access key whose derivation is a prefix to the derivation path of the reference can decrypt the reference by deriving its key using their own key and the rest of the derivation path of the reference.

This allows for a tree-like hierarchy of roles, possibly reflecting an organizational structure. As long as role changes are "promotions", i.e. result in increase of privileges, it is sufficient to change a single ACT entry for each role change.

The exact format of manifests as well as the serialisation conventions are specified in more detail in 7.5

Feeds are a unique feature of Swarm. They constitute the primary use case for single owner chunks. Feeds can be used for versioning revisions of a mutable resource, indexing sequential updates to a topic, publish the parts to streams, or post consecutive messages in a communication channel to name but a few. Feeds implement persisted pull-messaging and can also be interpreted as a pub-sub system. First, in 4.3.1, we introduce how feeds are composed of single owner chunks with an indexing scheme, the choice of which we discuss in 4.3.2. In 4.3.3, we analyse why feed integrity is relevant and how it can be verified and enforced. 4.3.4 describe epoch-based feeds which provide feeds subject to receiving sporadic updates a way to be searched. Finally, in 4.3.5, we show how feeds can be used as an outbox to send and receive subsequent messages in a communication channel.

4.3.1 *Feed chunks*

A feed chunk is a single owner chunk with the associated constraint that the identifier is composed of the hash of a feed topic and a feed index. The topic is a 32-byte arbitrary byte array, this is typically the Keccak256 hash of one or more human readable strings specifying the topic and optionally the subtopic of the feed, see figure 44.

The index can take various forms defining some of the potentials types of feeds. The ones discussed in this section are: (1) simple feeds which use incremental integers as their index (4.3.2); (2) epoch-based feeds which use an epoch ID (4.3.4); and (3) private channel feeds which use the nonces generated using a double ratchet key chain. (4.3.5). The unifying property of all these feed types is that both the publisher (owner) and the consumer must be aware of the indexing scheme.

Publishers are the single owners of feed chunks and are the only ones able to post updates to their feed. Posting an update requires (1) constructing the identifier from the topic and the correct index and (2) signing it concatenated together with the hash of the arbitrary content of the update. Since the identifier designates an address in the owner's subspace of addresses, this signature effectively assigns the payload to this address (see 2.2.3). In this way, all items published on a particular feed are proven to have been created only by the owner of the associated private key.

Conversely, users can consume a feed by retrieving the chunk by its address. Retrieving an update requires the consumer to construct the address from the owner's public key and the identifier. To calculate the identifier they need the topic and the appropriate index. For this they need to know the indexing scheme.

Feeds enable Swarm users to represent a sequence of content updates. The content of the update is the payload that the feed owner signs against the identifier. The payload can be a swarm reference from which the user can retrieve the associated data.

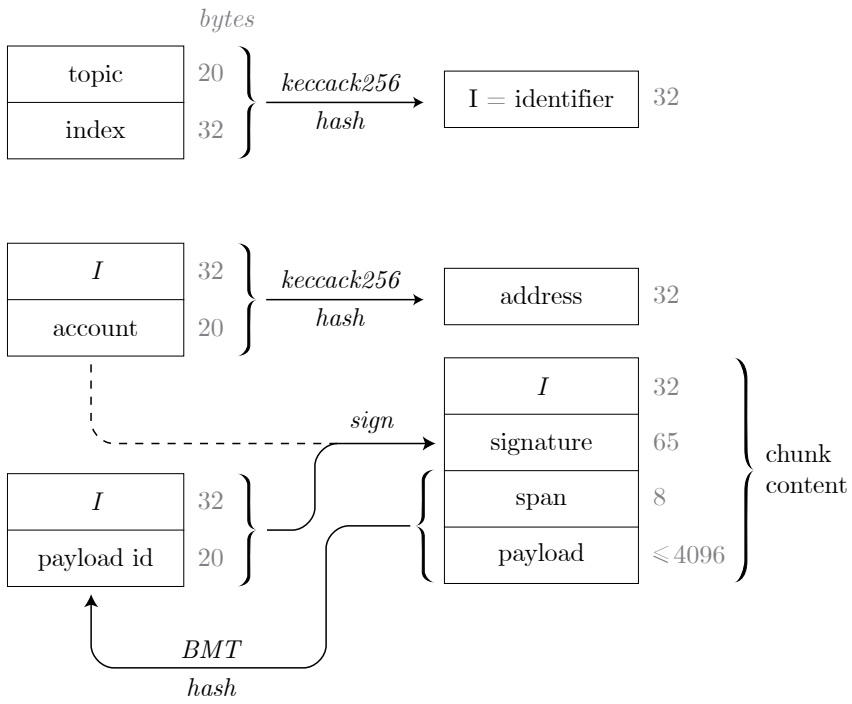


Figure 44: Feed chunks are single owner chunks where the identifier is the hash of the topic and an index. Indexes are deterministic sequences calculated according to an indexing scheme. Subsequent indexes for the same topic represent identifiers for feed updates.

4.3.2 *Indexing schemes*

Different types of feeds require different indexing schemes and different lookup strategies. In what follows, we introduce a few largely independent dimensions in which feeds can be categorised and which appear relevant in making a choice.

The actual indexing scheme used or even whether there is one (i.e. if the single owner chunk is a feed chunk at all) is left unrevealed in the structure of a feed chunk. As this information is not needed for the validation of a chunk by forwarding nodes, having the subtype explicit in the structure would leak information unnecessarily.

Update semantics

Updates of a feed can have three distinct semantics defining three subtypes of feeds. Revisions or mutable resource updates are *substitutive*, series updates are *alternative*, while partition updates are *accumulative*.

Feeds that represent revisions of the same semantic entity are called mutable resource updates. These resources mutate because the underlying semantic entity changes such as versions of your CV or the resource description becomes more elaborate like the Wikipedia entry about a Roman emperor. Users will typically be interested in the latest update of such resources, past versions having only historical significance.

A series of content connected by a common thread, theme or author such as status updates on social media, a person's blog posts or blocks of a blockchain can also be represented with feeds called series. The updates in a series are interpreted as alternative and independent instantiations or episodes manifesting in temporal sequence.

Finally, there are partitions expressed as feeds, updates of which are meant to be accumulated or added to earlier ones, for instance parts of a video stream. These mainly differ from series in that the feed updates are not interpretable on their own and the temporal sequence may represent a processing order corresponding to some serialisation of the structure of the resource rather than temporal succession. When such a feed is accessed, accumulation of all the parts may be necessary even for the integrity of the represented resource.

If subsequent updates of a feed include a reference to a data structure indexing the previous updates (e.g. a key–value store using the timestamp for the update or simply the root hash of the concatenation of update content), then the lookup strategy in all three cases reduces to looking up the latest update.

Update frequency

Within feeds which are updated over time may be several types, there are sporadic feeds with irregular asynchronicities, i.e. updates that can have unpredictable gaps and also periodic feeds with updates published at regularly recurring intervals.

We will also talk about real-time feeds, where the update frequencies may not be regular but do show variance within the temporal span of real-time human interaction, i.e. they are punctuated by intervals in the second to minute range.

Subscriptions

Feeds can be interpreted as pub/sub systems with persistence enabling asynchronous pulls. In what follows we analyse how the implementation of subscriptions to feeds as pub/sub is affected by the choice of indexing scheme.

In order to cater for subscribers to a feed, updates need to be tracked. If we know the latest update, periodic polling needs to be used to fetch the subsequent update. If the feed is periodic one can start polling after a known period. Alternatively, if the feed updates are frequent enough (at most a 1-digit integer orders of magnitude rarer than the desired polling frequency), then polling is also feasible. However, if the feed is sporadic, polling may not be practical and we better resort to push notifications (see 4.4.1 and 4.4.4).

If we missed out on polling for a period of time due to being offline, or just created the subscription, we can also rely on push notifications or use a lookup strategy.

To look up partitions is not a problem since we need to fetch and accumulate each update, so in this case the strategy of just iterating over the successive indexes cannot be improved. For periodic feeds, we can just calculate the index for a given time, hence asynchronous access is efficient and trivial. Looking up the latest version of a sporadically updated feed, however, necessitates some search and hence will benefit from epoch-based indexing.

Aggregate indexing

A set of sporadic feeds can be turned into a periodic one using feed aggregation. Imagine a multi-user forum like Reddit where each registered participant would publish comments on a post using sporadic feeds. In this scenario it is not practical for each user to monitor the comment feed of every other user and search through their sporadic feeds for updates in order to retrieve all the comments on the thread. It is much more efficient to just do it once though for all users. Indexers do exactly that and aggregate everyone's comments into an index, a data structure the root of which can now be published as a periodic feed, see figure 45. The period can be chosen to give a real-time feed experience; even if the rate of change does not justify it, i.e. some updates will be redundant, the cost amortises over all users that use the aggregate feed and therefore is economically sustainable.

Such a service can be provided with arbitrary levels of security, yet trustlessly without resorting to reputation. Using consensual data structures for the aggregation, incorrect indexes can be proved using cheap and compact inclusion proofs (see 4.1.1) and therefore any challenge relating to correctness can be evaluated on chain. The

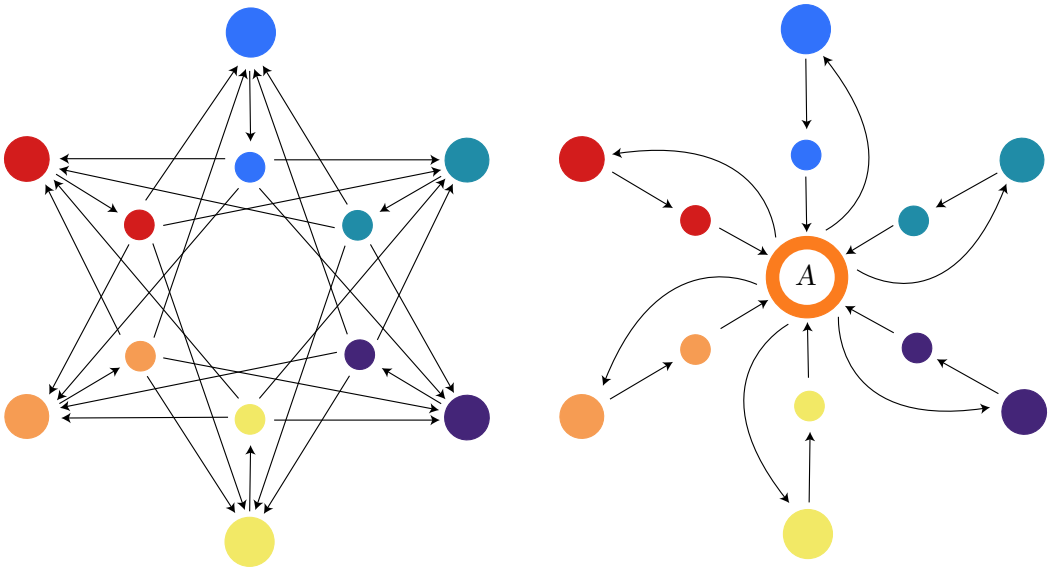


Figure 45: Feed aggregation serves to merge information from several source feeds in order to save consumers from duplicate work. **Left:** 6 nodes involved in group communication (discussing a post, having real time chat, or asynchronous email thread). Each node publishes their contribution as outbox feed updates (small coloured circles). Each participant polls the other's epoch based feeds duplicating work with the lookup. **Right:** the 6 nodes now register as sources with an aggregator which polls the nodes' feed and creates indices that aggregate the sources into one data structure which each participant can then pull.

threat of losing their deposit in case of an unrefuted challenge acts as a strong incentive for providers to maintain a good quality of service.

4.3.3 *Integrity*

We say that a feed has *integrity* if each of its updates has integrity, i.e. the update is unambiguous. Formally this means that for each index, the respective feed identifier is only ever assigned to a single payload. Incidentally, this also implies that the corresponding feed chunk has integrity. As discussed in 2.2.3, this is a prerequisite for consistent retrieval. If the payloads of the successive updates are imagined as blocks of a blockchain, then the criterion of integrity demands that feed owners must not fork their chain.

In fact, the integrity of a feed can only be guaranteed by the owner. But can it be checked or enforced? Owners can commit to the integrity of their feeds by staking a deposit on the blockchain which they stand to lose if they are found to double sign on an update. Even though this may give a strong disincentive to fork a feed for long-term benefits, by itself, it is still unable to give sufficient guarantees to consumers of the feed with respect to integrity. Because of this, we must design indexing schemes which work to enforce this integrity.

Authoritative version history

Mutable resource update feeds track versions pretty much the same way as the Ethereum Name Service does. When the owner consolidates a version (say of a website), they want to register the content address of the current version. In order to guarantee that there is no dispute over history, the payload needs to incorporate the previous payload's hash. This imposes the requirement that the payload must be a composite structure. If we want the payload to just be a manifest or manifest entry, so that it can be rooted to a path matching a URL, or directly displayed, this is not possible. Also, if the feed content is not a payload hash, then ENS registers a payload hash but the chunk may not exist on Swarm, so then the semantics of ENS are violated.

An indexing scheme which incorporates the previous payload hash into the subsequent index behaves like a blockchain in that it expresses the owner's unambiguous commitment to a particular history and that any consumer reading and using it expresses their acceptance of such a history. Looking up such feeds is only possible by retrieving each update since the last known one. The address is that of the update chunk, so registering the update address both guarantees historic integrity and preserves ENS semantics so that the registered address is just a Swarm reference to a chunk. Such feeds implement an authoritative version history, i.e. a secure audit trail of the revisions of a mutable resource.

Real-time integrity check

A deterministically indexed feed enables a real-time integrity check. In the context of feeds that represent blockchains (ledgers/side-chains), integrity translates to a non-forking and unique chain commitment. The ability to enforce this in real-time allows fast and secure definitions of transaction finality.

We illustrate this with an example of an off-chain p2p payment network where each node's locked up funds are allocated to a fixed set of creditors (see more detail in [Tron et al., 2019a]). Creditors of the node need to check the correctness of reallocations, i.e. that the total increases are covered by countersigned decreases. If a debitor keeps publishing a deposit allocation table for an exhaustive list of creditors, by issuing two alternatives to targeted creditors, the debitors will be able to orchestrate a double spend. Conversely, certainty in the uniqueness of this allocation table allows the creditor to conclude finality.

We claim that using Swarm feeds, this uniqueness constraint can be checked real time.

The insight here is that it is impossible to meaningfully control the responses to a single owner chunk request: There is no systematic way to respond with particular versions to particular requestors even if the attacker controls the entire neighbourhood of the chunk address.[3] This is the result of the ambiguity of the originator of the request due to the nature of forwarding Kademlia. Let us imagine that the attacker, with some sophisticated traffic analysis, has the chance of $1/n$ (asymptotic ceiling) to identify the originator and give a differential response. Sending multiple requests from random addresses, however, one can test integrity and consider a consistent response a requirement to conclude finality. The chance that the attacker can give a consistent differential response to a creditor testing with k independent requests is $1/n^k$. With linear bandwidth cost in k, we can achieve exponential degrees of certainty about the uniqueness of an update. If a creditor finds consistency, it can conclude that there is no alternative allocation table.

By requiring the allocation tables to be disseminated as feed updates, we can leverage permissionlessness, availability and anonymity to enforce feed integrity. If the feed is a blockchain-like ledger, a real-time integrity check translates to fork finality.

4.3.4 *Epoch-based indexing*

In order to use single owner chunks to implement feeds with flexible update frequency, we introduce epoch-based feeds, an indexing scheme where the single owner chunk's identifier incorporates anchors related to the time of publishing. In order to be able to find the latest update, we introduce an adaptive lookup algorithm.

3 If the chunks are uploaded using the same route, the chunk that comes later will be rejected as already known. If the two chunks originate from different addresses in the network, they might both end up in their local neighbourhood. This scenario will result in inconsistent retrievals depending on which node the request ends up with.

Epoch grid

An epoch represents a concrete time period starting at a specific point in time, called the epoch base time and has a specific length. Period lengths are expressed as powers of 2 in seconds. The shortest period is $2^0 = 1$ second, the longest is 2^{31} seconds.

An epoch grid is the arrangement of epochs where rows (referred to as levels) represent alternative partitioning of time into various disjoint epochs with the same length. Levels are indexed by the logarithm of the epoch length putting level 0 with 1 second epoch length at the bottom by convention, see figure 46.

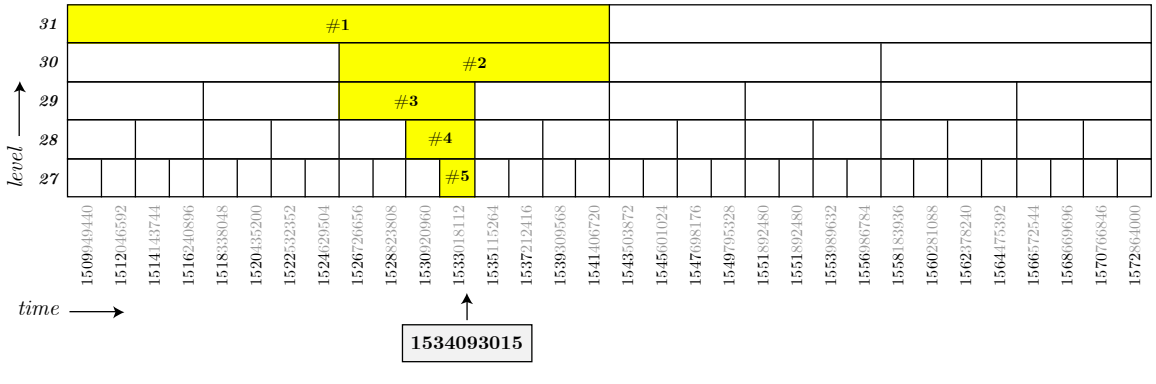


Figure 46: Epoch grid showing the first few updates of an epoch-based feed. Epochs occupied are marked in yellow and are numbered to reflect the order of updates they represent.

When representing a epoch-based feed in an epoch grid, each update is assigned to a different epoch in the grid based on their timestamp. In particular, an update is mapped to the longest free epoch that includes the timestamp. This structure gives the series of updates a contiguous structure which allows for easy search. The contiguity requirement implies that by knowing the epoch of the previous update, a subsequent update can be mapped to an epoch unambiguously.

To identify a specific epoch, we need to know both the epoch base time and the level. This pair is called the epoch reference. To calculate the epoch base time of any given instant in time t at a particular level l, we are dropping the l least significant bits of t. The level requires one byte, and the epoch base time (using linux seconds) 4 bytes so the epoch reference can be serialised in 5 bytes. The epoch reference of the very first update of any epoch-based feed is always the same.

Mapping epochs to feed update chunks

Feed updates can then be mapped to feed chunks using the serialised epoch reference as the feed index. The topic of the feed hashed together with the index results in the feed identifier used in constructing the single owner chunk that expresses the feed chunk.

To determine the epoch in which to store a subsequent update, the publisher needs to know where they stored the previous update. If the publisher does not keep track of this, they can use the lookup algorithm to find their last update.

Lookup algorithm

When consumers retrieve feeds, they typically will either want to look up the state of the feed at a particular time (historical lookup) or to find the latest update.

If historical lookups based on a *target* time are required, the update can incorporate a data structure mapping timestamps to states. In such cases, finding any update later than the target can be used to deterministically look up the state at an earlier time.

If no such index is available, historical lookups need to find the shortest filled epoch whose timestamp is earlier than the target.

To select the best starting epoch from which to walk our grid, we have to assume the worst case scenario, which is that the resource was never again updated after we last saw it. If we don't know when the resource was last updated, we assume 0 as the "last time" it was updated.

We can guess a start level as the position of the first nonzero bit of $lastUpdate \veebar NOW$ counting from the left. The bigger the difference among the two times (last update time and now), the higher the level will be.

In A.7, we walk the reader through an example.

4.3.5 *Real-time data exchange*

Feeds can be used to represent a communication channel, i.e. the outgoing messages of a persona. Such a feed, called an outbox feed can be created to provide email-like communication or instant messaging, or even the two combined. For email-like asynchronicities, epoch-based indexing can be used while for instant messaging, it is best to use deterministic sequence indexing. In group chat or group email, confidentiality is handled using an access control trie over the data structure, indexing each party's contribution to the thread. Communication clients can retrieve each group member's feed relating to a thread and merge their timelines for rendering.

Even forums could be implemented with such an outbox mechanism, however, above a certain number of registered participants, aggregating all outboxes on the client side may become impractical and require index aggregators or other schemes to crowdsource combination of the data.

Two-way private channels

Private two-party communication can also be implemented using outbox feeds, see figure 47. The parameters of such feeds are set as part of an initial key exchange or registration protocol (see 4.4.2) which ensures that the parties consent on the indexing scheme as well as the encryption used.

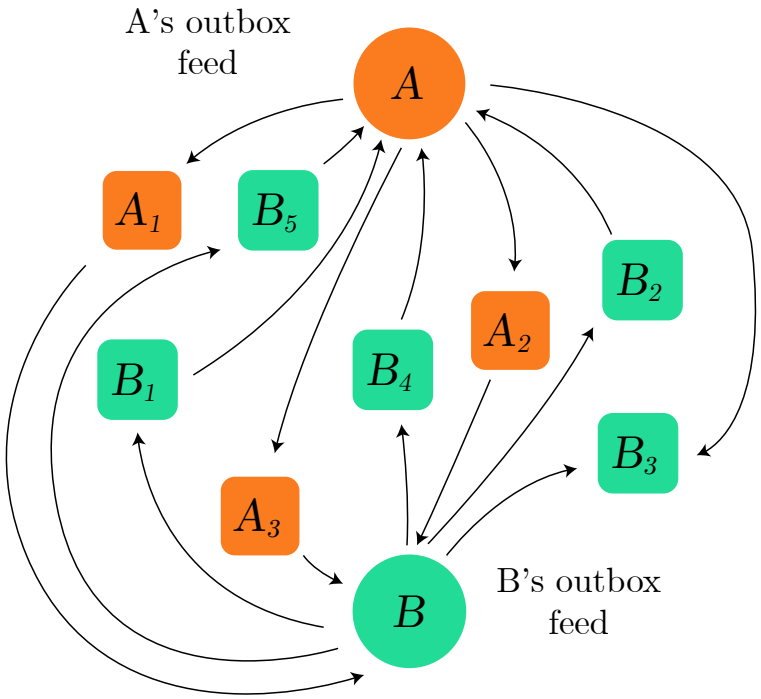


Figure 47: Swarm feeds as outboxes for private communication. Outbox feeds represent consecutive messages from a party in a conversation. The indexing scheme can follow a key management system with strong privacy which obfuscates the communication channel itself and renders interception attacks prohibitively expensive.

The real-time series feed used for instant messaging ought to have an indexing scheme with deterministic continuations for at least a few updates ahead. This enables sending retrieve requests for future updates ahead of time, i.e. during or even prior to processing the previous messages. When such retrieve requests arrive at the nodes whose address is closest to the requested update address, the chunk will obviously not be able to be found as the other party will not have sent them yet. However, even these storer nodes are incentivised to keep retrieve requests alive until they expire (see the argument in 2.3.1). This means that up until the end of their time-to-live setting (30 seconds), requests will behave as subscriptions: the arrival of the update chunk triggers the delivery response to the open request as if it was the notification sent to the subscriber. This reduces the expected message latency down to less than twice the average time of one-way forwarding paths, see figure 48.

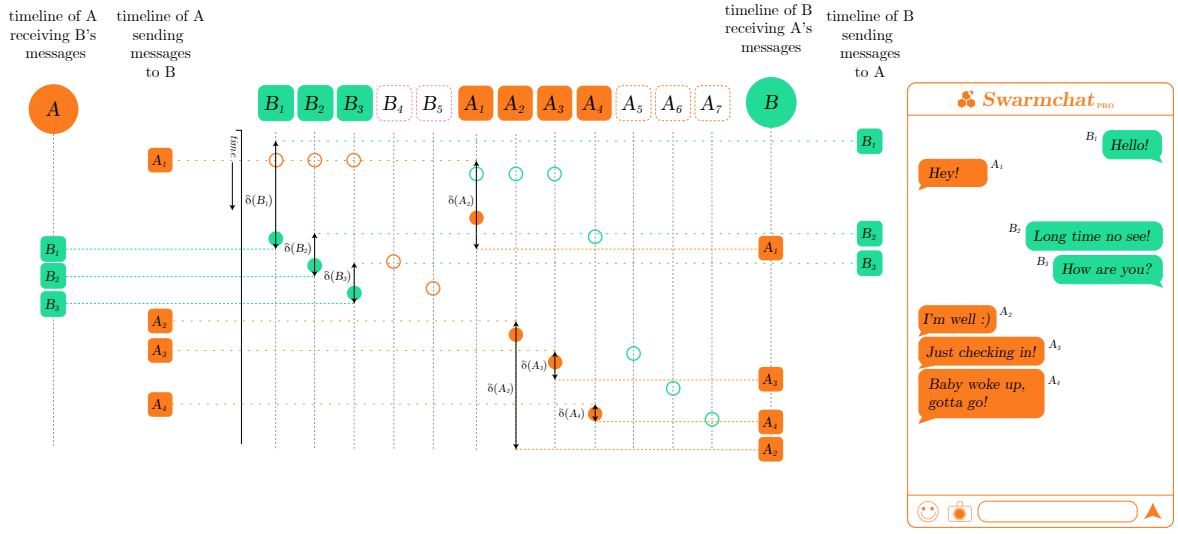


Figure 48: Advance requests for future updates. The diagram shows the time of a series of events during instant messaging between two parties A and B using outbox feeds. The columns designate the neighbourhood locations of the feed update addresses. Circles show the time of protocol messages arriving: colors indicate the origin of data, empty circles are retrieve requests, full circles are push sync deliveries arriving at the respective neighbourhood. Note that the outbox addresses are deterministic 3 messages ahead so retrieve requests can be sent before the update arrives.
Importantly, latency between one party sending a message m and the other receiving it is shown as $\delta(m)$. Messages A_3 and A_4 arrive before A_2 which can be reported and repaired. If address predictability was only possible for 1 message ahead, both B_2 and B_3 would have much longer latencies.
Also note that the latency of B_2 and B_3 are helped by advance requests: the retrieve requests for B_4 and B_5 are sent upon receipt of B_1 and B_2 and arrive at their neighbourhood the same time as the messages B_2 and B_3 arrive at theirs, respectively. If address predictability was only 1 message ahead, this would also cause both B_2 and B_3 to have much longer latencies.

Post-compromise security

A key management solution called double ratchet is the de-facto industry standard used for encryption in instant messaging. It is customary to use the extended triple Diffie–Hellmann key exchange (X3DH) to establish the initial parameters for the double-ratchet key chains (see 4.4.2).

Double-ratchet combines a ratchet based on a continuous key-agreement protocol with a ratchet based on key-derivation function [Perrin and Marlinspike, 2016]. This scheme can be generalised [Alwen et al., 2019] and understood as a combination of well-understood primitives and shown to provide (1) forward secrecy, (2) backward secrecy,[4] and (3) immediate decryption and message loss resilience.

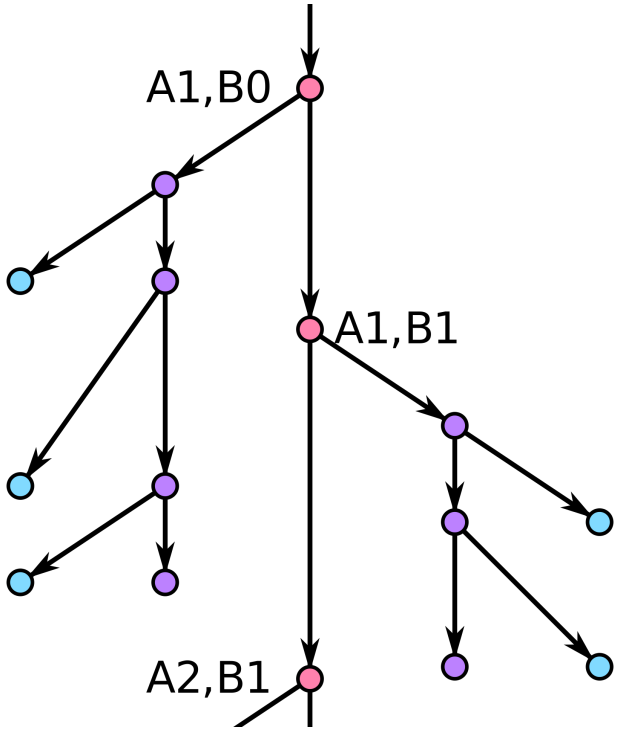


Figure 49: Future secrecy for update addresses

On top of the confidentiality due end-to-end encryption, Swarm offers further resistance to attacks. Due to forwarding Kademlia, the sender is ambiguous and deniable. Due to normal push-sync and pull-sync traffic, messages are also obfuscated. To make it really hard for an attacker, the sequence of indexes can also provide future secrecy if we add more key chains to the double-ratchet machinery. Beside root, sending and receiving encryption key chains, we would need to introduce two more: outgoing and incoming outbox index key chains, see figure 49. As a result of this measure the underlying communication channel is obfuscated, i.e. intercepting an outbox update chunk and knowing its index, reveals nothing about previous or

4 Also known as future secrecy or post-compromise security.

subsequent outbox update indexes. This makes subsequent messages prohibitively difficult and costly to monitor or intercept.

In 4.3.3, we used factoring in the payload hash into the indexing scheme to achieve non-mergeability of chains (unambiguous history). Inspired by this, we propose also to factor in the payload hash into the subsequent feed update index. This results in the additional property called recover security, which, intuitively, ensures that once an adversary manages to forge a message from A to B, then no future message from A to B will be accepted by B. This is guaranteed if the authenticity of A's messages to B affects the subsequent feed index. If there is a mismatch (one of the messages was forged), messages will be looked up at the wrong address and therefore the communication channel will be abandoned and a new one is initiated. Such a communication channel represents a completely confidential zero-leak solution for real-time messaging.

4.4 PSS: DIRECT PUSH MESSAGING WITH MAILBOXING

This section introduces *pss*, Swarm's direct node-to-node push messaging solution. Functionalities of and motivation for its existence are playfully captured by alternative resolutions of the term:

- *postal service on Swarm* – Delivering messages if recipient is online or depositing for download if not.
- *pss is bzz whispered* – Beyond the association to Chinese whispers, it surely carries the spirit and aspiration of Ethereum Whisper.[5] Pss piggy-backs on Swarm's distributed storage for chunks and hence inherits their full incentivisation for relaying and persistence. At the same time it borrows from Whisper's crypto, envelope structure and API.
- *pss! instruction to hush/whisper* – Evokes an effort to not disclose information to 3rd parties, which is found exactly in the tagline for pss: truly zero-leak messaging where beside anonymity and confidentiality, the very act of messaging is also undetectable.
- *pub/sub system* – API allows publishing and subscription to a topic.

First, in 4.4.1, we introduce Trojan Chunks, i.e. messages to storers that masquerade as chunks whose content address happens to fall in the proximity of their intended recipient. 4.4.2 discusses the use of pss to send a contact message to open a real time communication channel. In 4.4.3, we explore the mining of feed identifiers to target a neighbourhood with the address of a single owner chunk and present the construct of an addressed envelope. Finally, building on Trojan chunks and addressed

5 Whisper is a gossip-based dark messaging system, which is no longer developed. It never saw wide adoption due to its (obvious) lack of scalability. Whisper, alongside Swarm and the Ethereum blockchain, was the communication component of the holy trinity, the basis for Ethereum's original vision of web3.

envelopes, 4.4.4 introduces update notification requests. The user experience pss offers is discussed later in 6.3.2.

4.4.1 *Trojan chunks*

Cutting edge systems promising private messaging often struggle to offer truly zero-leak communication [Kwon et al., 2016]. While linking the sender and recipient is cryptographically proven to be impossible, resistance to traffic analysis is harder to achieve. Having sufficiently large anonymity sets requires high volumes available at all times. In the absence of mass adoption, guaranteeing high message rate in dedicated messaging networks necessitates constant fake traffic. With Swarm, the opportunity arises to disguise messages as chunk traffic and thereby obfuscate even the act of messaging itself.

We define a Trojan chunk as a content addressed chunk the content of which has a fixed internal structure, see figure 51:

1. *span* – 8 byte little endian uint64 representation of the length of the message
2. *nonce* – 32 byte arbitrary nonce
3. *Trojan message* – 4064 byte asymmetrically encrypted message ciphertext with underlying plaintext composed of
   a) *length* – 2 byte little endian encoding of the length of the message in bytes $0 \leq l \leq 4030$,
   b) *topic* – 32 byte obfuscated topic id
   c) *payload* – m bytes of a message
   d) *padding* – $4030 - m$ random bytes.

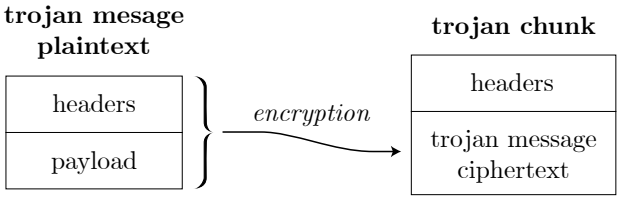


Figure 50: A pss message is a Trojan chunk that wraps an obfuscated topic identifier with a Trojan message, which in turn wraps the actual message payload to be interpreted by the application that handles it.

Knowing the public key of the recipient, the sender wraps the message in a trojan message (i.e. prefixing it with length, then padding it to 4030 bytes) then encrypts it using the recipient's public key to obtain the ciphertext payload of the trojan chunk

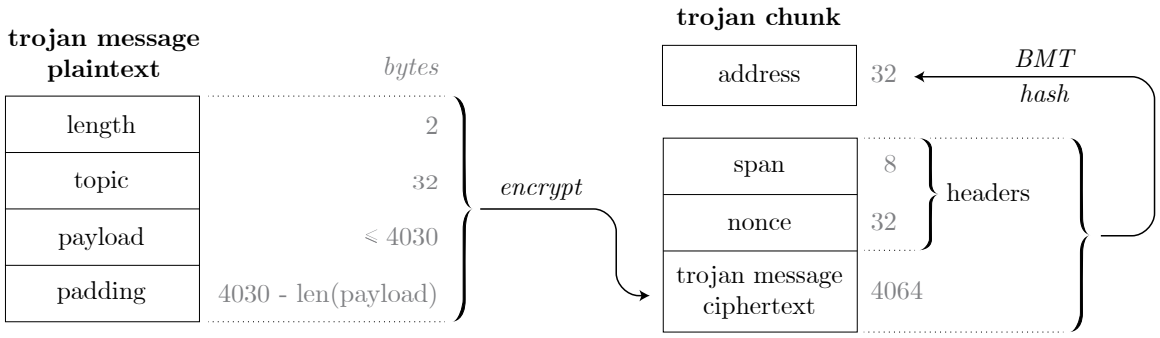


Figure 51: The Trojan chunk wraps an asymmetrically encrypted Trojan message.

by asymmetric encryption. Then the sender finds a random nonce such that when it is prepended to the payload, the chunk hashes to an address that starts with a destination target prefix. The destination target is a bit sequence that represents a specific neighbourhood in the address space. If the target is a partial address derived as a prefix of the recipient overlay address, matching the target means that the chunk falls in the neighbourhood of the recipient. If only the public key is known, it is assumed that it is the bzz account of the recipient, i.e. their overlay address can be calculated from it[6] (see 2.1.2 and 7.2). The sender then uploads the resulting chunk to Swarm with postage stamps of their choice which then ends up being synced to the recipient address' neighbourhood. If the recipient node is online they receive the chunk for certain provided the bit length of the matching target is greater than the recipient's neighbourhood depth. In practice, targets should be $n + c$ bits long where n is the estimated average depth in swarm and c is a small integer.

Receiving Trojan messages

The recipient only knows that a chunk is a pss message if and when they successfully opened the Trojan message with the private key corresponding to the public key that they advertise as their resident key (see 4.4.2) and do an integrity check/topic matching. Nodes that want to receive such Trojan Messages will keep trying to open all messages that they are closest to. Forwarding nodes (or anyone else apart from sender and recipient) have no way to distinguish between a random encrypted chunk and a trojan message, which means that communication is perfectly obfuscated as generic chunk traffic.

After the recipient has opened the envelope using asymmetric decryption, there is a combined step of integrity check and topic matching. Knowing the length of the payload (from the first 2 bytes of the message), the recipient takes the payload slice and calculates the Keccak256 hash of it. Now for each topic the client has a subscription to, it then hashes the payload hash together with the topic. If the resulting segment

6 Alternative overlays can be associated with a public key, and several public keys can be listened on by a node at a particular address.

xor-ed with the topic matches the obfuscated topic id in the message then the message is indeed meant as a message with the said topic and the registered handler is called with the payload as argument.

Mailboxing for asynchronous delivery

If the recipient is not online the chunk will prevail as any other chunk would, depending on the postage stamp it has. Whenever the recipient node comes online, it pull-syncs the chunks from the neighbourhood closest to it, amongst them all the Trojan chunks, and amongst them their own as yet unreceived messages. In other words, through Trojan messages pss automatically provides asynchronous *mailboxing* functionality, i.e. without any further action needed from the sender, even if they are offline at the time that their correspondent has sent them, undelivered messages are preserved and available to the recipient whenever they come online. The duration of mailboxing is controlled with postage stamps in exactly the same way as the storage of chunks, in fact, it is totally indistinguishable.

Mining for proximity

The process of finding a hash close to the recipient address is analogous to mining blocks on the blockchain. The nonce segment in a Trojan chunk also plays exactly the same role as a block nonce: it provides sufficient entropy to guarantee a solution. The difficulty of mining corresponds to the length of the destination target: The minimum proximity order required to ensure that the recipient will receive the message needs to be higher than the neighbourhood depth of the recipient[7] when it comes online, so it is logarithmic in the number of nodes in the network. The expected number of nonces that need to be tried per Trojan message before an appropriate content address is found is exponential in the difficulty, and therefore equal to the number of nodes in the network. As the expected number of computational cycles needed to find the nonce equals the network size, in practice, mining a Trojan will never be prohibitively expensive or slow even for a single node. A small delay in the second range is expected only in a network of a billion nodes and even that is acceptable given that Trojan messages are meant to be used only for one-off instances such as initiations of a channel. All subsequent real-time exchange will happen using the previously described bidirectional outbox model using single owner chunks.

Anonymous mailbox

Asynchronous access to pss messages is guaranteed if the postage stamp has not yet expired. The receiver only needs to create a node with an overlay address corresponding to the destination target advertised to be the recipient's resident address.

7 It makes sense to use the postage lottery batch depth (see 3.3.2) as a heuristic for the target proximity order when mining a Trojan chunk. This is available as a read-only call to the postage stamp smart contract.

One can simply create an anonymous mailbox. An anonymous mailbox can receive pss messages on behalf of a client and then publish those on a separate, private feed so that the intended recipient can read them whenever they come back online.

Register for aggregate indexing

As mentioned in 4.3.2, aggregate indexing services help nodes in monitoring sporadic feeds. For instance, a forum indexer aggregates the contribution feeds of registered members. For public forums, off-chain registration is also possible and can be achieved by simply sending a pss message to the aggregator.

4.4.2 *Initial contact for key exchange*

Encrypted communication requires a handshake to agree on the initial parameters that are used as inputs to a symmetric key generation scheme. The extended triple Diffie–Hellmann key exchange (X3DH) is one such protocol [Marlinspike and Perrin, 2016] and is used to establish the initial parameters for a post-handshake communication protocol such as the *double-ratchet scheme* discussed earlier in the section on feeds (see 4.3.5). In what follows, we describe how the X3DH protocol can be implemented with pss in a serverless setting.

Swarm's X3DH uses the same primitives as are customary in Ethereum, i.e. secp256k elliptic curve, Keccak256 hash, and a 64-byte encoding for EC public keys.

The Swarm X3DH protocol is set out to allow two parties to establish a shared secret that forms the input used to determine the encryption keys which will be used during post-handshake two-way messaging. The *initiator* is the party that initiates a two-way communication with the *responder*. The responder is supposed to advertise the information necessary for parties not previously known to the responder to be able to initiate contact. Zero leak communication can be achieved by first performing an X3DH to establish the seed keys used by the double-ratchet protocol for the encryption of data, as well as the feed indexing methodology used. This will enable the responder to retrieve the updates of the outbox feed.

X3DH uses the following keys:[8]

- responder long-term public identity key - K_r^{ENS},
- responder resident key (aka signed pre-key) - K_r^{Res},
- initiator long-term identity key - K_i^{ID},
- initiator ephemeral key for the conversation - K_i^{EPH}.

8 The protocol specifies one-time pre-keys for the responder, but these can be safely ignored since they only serve as replay protection, which is solved by other means in this implementation.

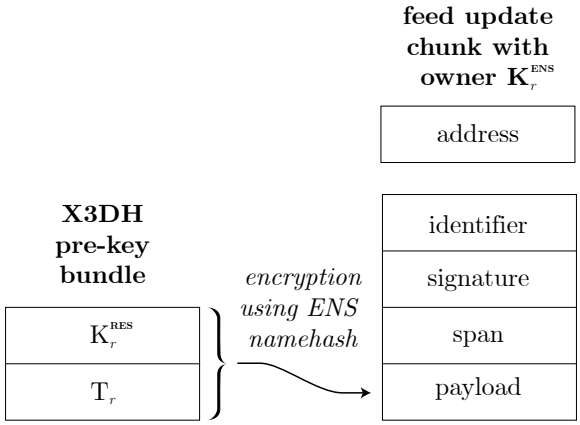


Figure 52: The X3DH pre-key bundle feed update contains the resident key and resident address and is optionally together encrypted with the ENS name hash to prove uniqueness and provide authentication.

A pre-key bundle consists of all information the initiator needs to know about responder. However, instead of being stored on (usually 3rd-party) servers, this information is instead stored in Swarm. For human-friendly identity management, ENS can be optionally used to provide familiar username based identities. The owner of the ENS resolver represents the long-term public identity key of this persona and is considered authenticated. The long-term identity address can be used to construct an epoch-based feed with a topic id indicating it provides the pre-key bundle for would be correspondents. When communication is initiated with a new identity, the latest update from the feed is retrieved by the initiator, containing the current *resident key* (aka *signed pre-key*) and current *addresses of residence*, i.e. (potentially multiple) overlay destination targets where the persona is expecting she may receive pss messages. The signature in the feed update chunk signs both the resident key (cf. signed pre-key) and the destination targets. The public key that is recovered from this signature gives the long-term identity public key, see figure 52.

In order to invite a responder to an outbox-feed based private communication channel, the initiator first looks up the responder's public pre-key bundle feed and sends an initial message to the responder (see figure 53) in which the intent to communicate is indicated. She then shares the parameters required to initiate the encrypted conversation. This consists of the public key of its long-term identity, as well as the public key of the ephemeral key-pair which has been generated just for that conversation. These details are delivered to the potential responder by sending a Trojan pss message addressed to the responder's current address of residence which is also being advertised in their pre-key bundle feed.

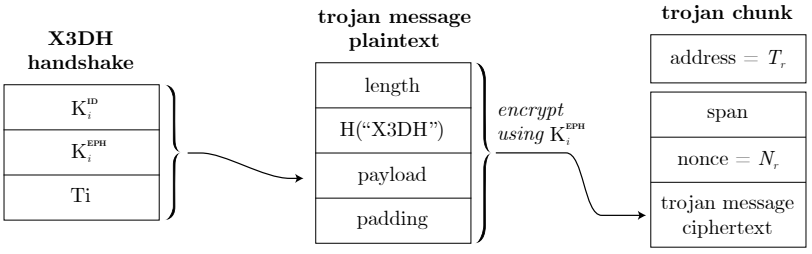


Figure 53: X3DH initial message. Initiator retrieves the ENS owner as well as the latest update of responder's pre-key bundle feed containing the resident key and resident address. Initiator sends their identity key and an ephemeral key to responder's resident address using the resident key for encryption.

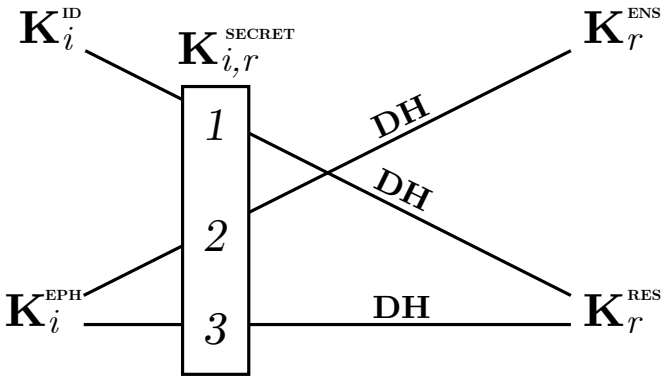


Figure 54: X3DH secret key. Both parties can calculate the triple Diffie-Hellmann keys and xor them to get the X3DH shared secret key used as the seed for the post-handshake protocol.

After the responder receives this information, both parties have all the ingredients needed to generate the triple Diffie-Hellmann shared secret, see figure 54.[9] This shared secret constitutes the seed key for the double-ratchet continuous key agreement protocol as used in the signal protocol. The double-ratchet scheme provides forward secrecy and post-compromise security to the end-to-end encryption. By applying separate key-chains for the outbox feed's indexing scheme, additional recover security, i.e. resilience to message insertion attack, can be achieved. Most importantly, however, by adding forward and backward secrecy to outbox addresses, the communication channel is obfuscated, which renders sequential message interception contingent on the same security assumptions as encryption and therefore eliminates the only known attack surface for double-ratchet encryption. The obfuscation and deniability of the channel based on outbox feeds, together with the initial X3DH message being disguised indistingushable as a chunk warrants designating this as zero-leak communication.

4.4.3 *Addressed envelopes*

Mining single owner chunk addresses

The question immediately arises whether it makes sense to somehow mine single owner chunks. Since the address in this case is the hash of a 32 byte identifier and a 20 byte account address, the id provides sufficient entropy to mine addresses even if the owner account is fixed. So for a particular account, if we find an id such that the resulting single owner chunk address is close to a target overlay address, the chunk can be used as a message in a similar way to Trojan chunks. Importantly, however, since the address can be mined before the chunk content is associated with it, this construct can serve as an addressed envelope.

Let us make explicit the roles relevant to this construct:

- *issuer* (I) - creates the envelope by mining an address.
- *poster* (P) - puts the content into the envelope and posts it as a valid single owner chunk to Swarm.
- *owner* (O) - possesses the private key to the account part of the address and can thus sign off on the association of the payload to the identifier. This effectively decides on the contents of the envelope.
- *target* (T) - the constraint for mining: a bit sequence that must form the prefix of the mined address. It represents a neighbourhood in the overlay address space

9 If the X3DH does not use one-time pre-keys, the initial message can in theory be re-sent by a third party and lead the responder to assume genuine repeated requests. Protocol replay attacks like this are eliminated if the post-handshake protocol adds random key material coming from the responder. But the initial Trojan message can also be required to contain a unique identifier, e.g. the nonce used for mining the chunk. Reusing the id is not possible since it leads to the same chunk.

where the envelope will be sent. The length of the target sequence corresponds to the difficulty for mining. The longer this target is, the smaller the neighbourhood that the envelope will be able to reach.

- *recipient* (R) - the party whose overlay address has the target sequence as its prefix and therefore the destination of the message

The envelope can be perceived as open: since the poster is also the owner of the response single owner chunk, they are able to control what content is put into the chunk. By constructing such an envelope, the issuer effectively allows the poster to send an arbitrary message to the target without the computation requirement needed to mine the chunk. See figure 55.

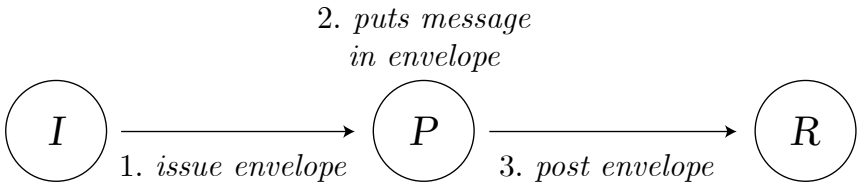


Figure 55: Stamped addressed envelopes timeline of events. Issuer I creates the envelope encrypted for P with an identifier such that P as the single owner of the chunk, produces an address that falls in the recipient R's neighbourhood. As a result (only) P can fill the envelope with arbitrary content and then use simple push-syncing to post it to R.

All the poster needs to do when they wish to send a message to the recipient is to create a Trojan message and sign it against the identifier using the private key of the same account the issuer used as an input when they mined the address. If they do this, the chunk will be valid. See figure 56.

Pre-paid postage

These chunks behave in the same way as normal Trojan messages, with their privacy properties being the same if not somewhat better since the issuer/recipient can associate a random public key with which the message is encrypted, or even use symmetric encryption. If a postage stamp is pre-paid for an address and given to someone to post later, they can use push-sync to send the chunk to the target without the poster needing to pay anything, the cost of this having already been covered by the stamped addressed envelope. Such a construct effectively implements *addressed envelopes with pre-paid postage* and serves as a base layer solution for various high-level communication needs: 1) push notifications about an update to subscribers without any computational or financial postage burden on the sender 2) free contact vouchers 3) zero-delay direct message response.

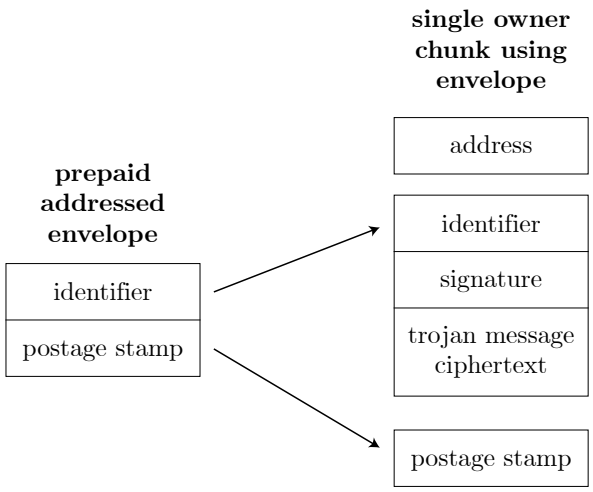


Figure 56: A stamped addressed envelope issued for P and addressed to R consists of an identifier which is mined so that when used to create a single owner chunk owned by P, it produces an address which falls within R's neighbourhood. This allows P to construct a message, sign it against the identifier and using the postage stamp, post it using the network's normal push-syncing to R for free.

Issuing a stamped addressed envelope

Issuing a stamped addressed envelope involves the following process:

1. *assume* issuer I, prospective poster P, and prospective recipient R with public keys K_I, K_P, K_R and overlay addresses A_I, A_P, A_R.

2. *mine* – I finds a nonce N_R such that when used as an identifier to create a single owner chunk, the address of the chunk hashes to H_R which is in the nearest neighbourhood of A_R.

3. *pay postage* – I signs H_R to produce a witness for an appropriate postage payment to produce stamp PS_R.

4. *encapsulate* – package N_R and PS_R which represent the pre-paid envelope pre-addressed to recipient address and encrypt it with K_P then wrap it as a Trojan chunk.

5. *mine* – find a nonce N_P such that the Trojan chunk hashes to H_P which is in the nearest neighbourhood of A_P.

Receiving a stamped addressed envelope

A prospective poster P is assumed to receive a a Trojan message consisting of pre-paid envelope E. In order to open it, she carries out the following steps:

1. *decrypt* message with the private key belonging to K_P
2. *deserialise* unpack and identify PS_R and N_R, extract H_R from PS_R
3. *verify* postage stamp PS_R and check if N_R hashed with the account for K_P results in H_R to ensure the associated address is in fact owned by P.
4. *store* N_R and PS_R

Posting a stamped addressed envelope

When the poster wants to use the envelope to send an arbitrary message M to R (with recipient R potentially unknown to the sender), they must follow the following steps:

1. *encrypt* the message content M with K_R to create a payload and wrap it in Trojan message T
2. *hash* the encrypted Trojan message resulting in H_T
3. *sign* H_T against the identifier N_R using the private key belonging to K_P producing signature W
4. *encapsulate* nonce N_R as id, the signature W and the Trojan message T as the payload of a valid single owner chunk with address H_R
5. *post* the chunk with the valid stamp PS_R

Receiving a posted addressed envelope

When R receives chunk with address H_R

1. *verify* postage stamp PS_R and validate the chunk as a single owner chunk with payload T.
2. *decrypt* T with the private key belonging to K_R.
3. *deserialise* the plaintext as a Trojan message, identify the message payload M and check its integrity.
4. *consume* M.

4.4.4 *Notification requests*

This section elaborates on the concept of addressed envelopes and presents three flavours, each implementing different types of notifications.

Direct notification from publisher

If an issuer wants a recipient to be notified of the next activity on a feed, she must construct a stamped addressed envelope embedded in a regular Trojan message and send it to the publisher, see figure 57. If the issuer is also the recipient, then the account that is used in the request and the one in the response envelope may be one and the same.

When the feed owner publishes an update to her feed, the reference to the update chunk is put into the envelope and is sent to the recipient. More formally, the publisher creates a single owner chunk from the identifier representing the pre-addressed envelope and therefore, as the owner, signs off on the identifier associated with the feed update content as the payload of the single owner chunk. Right after this, the same publisher acting as the poster, push-syncs the chunk to the swarm. This construct is called direct notification from publisher.

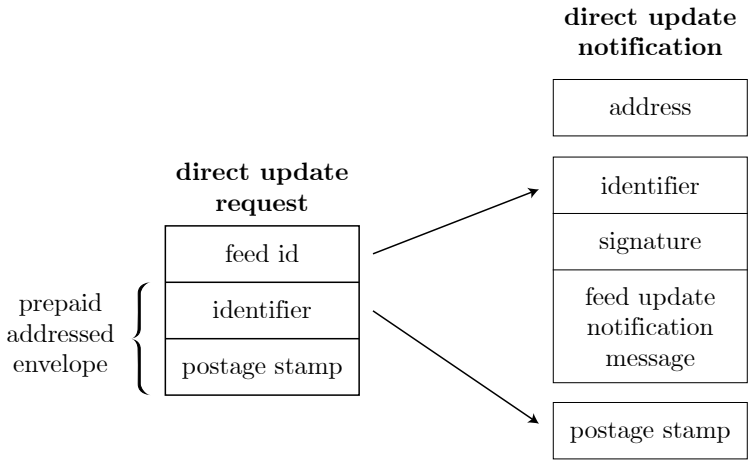


Figure 57: A direct notification request contains a reference to a feed and wraps a pre-paid envelope mined for P (the publisher or a known distributor of the feed) and addressed to recipient R. The response is identical to the process used for generic stamped addressed envelopes and only differs in that the message is supposed to be the feed update or a reference to its content.

The Trojan message specifies in its topic that it is a notification and is encrypted using the public key. As the address is mined to match the recipient overlay address on a sufficiently long prefix, the message ends up push-synced to the recipient's neighbourhood. Whenever the recipient comes online and receives the chunk by push-sync it detects that it is intended as a message. It can be identified as such either by decrypting the chunk content successfully with the key for the address they had advertised, or in case that the recipient has issued the pre-addressed envelope

themselves, simply by looking it up the against the record of the address they saved when it was issued. See figure 58 for the timeline of events.

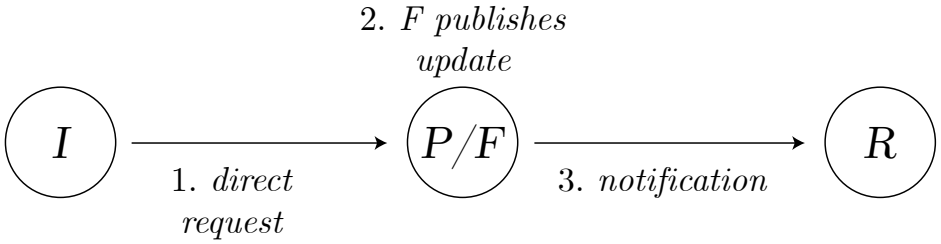


Figure 58: Direct notification from publisher timeline of events. Issuer I constructs a prepaid envelope for the publisher or a known distributor of the feed P/F and addressed to recipient R. Together with a feed topic I, it is sent to P/F wrapped in a pss trojan message. P/F receives it and stores the request. When they publish the update, they wrap it in the envelope, i.e. sign the identifier received from I against the feed update notification message and post it as a chunk, which R will receive and hence, be notified of the feed update.

Notifications coming from publishers directly enable the poster to put arbitrary content into the envelope. The poster is at the same time the owner so that they can sign off on any content against the identifier when posting the envelope. In order for the issuer to be able to create the notification chunk address in advance, the prospective poster's account must be known. However, the feed update address need not be fixed, so this scheme remains applicable to (sporadic) epoch-based feeds.

Notification from neighbourhood

Assume there is a feed whose owner has not revealed their overlay (destination target) or refuses to handle notifications. The feed is updated sporadically using simple sequential indexing and therefore polling is not feasible when it comes to looking up the latest update, especially as the consumer can go offline any time. Is there a way for consumers to still get notified?

We introduce another construct, a neighbourhood notification which works without the issuer of the notification knowing the identity of prospective posters. It presupposes, however, that the content or at least the hash of the content is known in advance so that it can be signed by the issuer themselves.

An issuer wants a recipient to be notified of the next update. The issuer can create a neighbourhood notification request, simply a Trojan chunk wrapping a message. This message contains an addressed envelope (identifier, signature and postage stamp) that the poster can use to construct the single owner chunk serving as the notification. Note that the notification message need not contain any new information, merely the fact of receiving it is sufficiently informative to the recipient. In order to indicate what it is a notification of, the notifications payload contains the feed update address.

Upon receiving the notification, the receiver can then simply send a regular retrieve request to fetch the actual feed update chunk containing or pointing to the content of the message. See figure 59 for the structure of neighbourhood notifications and notification requests.

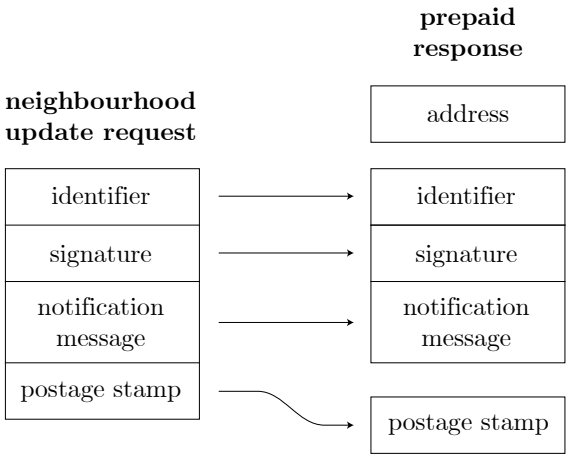


Figure 59: Neighbourhood notification requests are not only including identifier and postage stamp but also a notification message together with the signature attesting it against the address. Thus P needs to construct the actual notification and when publisher F posts their update to P-s neighbour, P can post the notification to recipient R.

If the notification needs only to contain the feed update address as its payload then associating it with the identifier can be signed off by the issuers themselves. This signature together with the identifier should be considered a necessary component of the envelope and must be sent as part of the notification request. Unlike the *open* envelopes used with publishers directly, neighbourhood notifications can be viewed as *closed* envelopes, where the content is sanctioned by the issuer in advance.

Now the issuer and not the poster becomes the owner of the chunk and the signature is already available to the poster when they create and post the notification. As a consequence no public key or account address information is needed from the poster. In fact, the identity of the poster does not need to be fixed, any peer could qualify as a candidate poster. Issuers therefore can send the notification request to the neighbourhood of the feed update chunk. The nearest neighbours will keep holding the request chunk as dictated by the attached postage stamps until they receive the appropriate feed update chunk that indicates the receipt of the notification. See figure 60 for the timeline of events when using neighbourhood notifications.

But how can we make sure that notifications are not sent too early or too late? While the integrity of notification chunks is guaranteed by the issuer as their single owner, measures must also be taken to prevent nodes that manage the notifications from sending them before the update should actually arrive. It would be ideal if

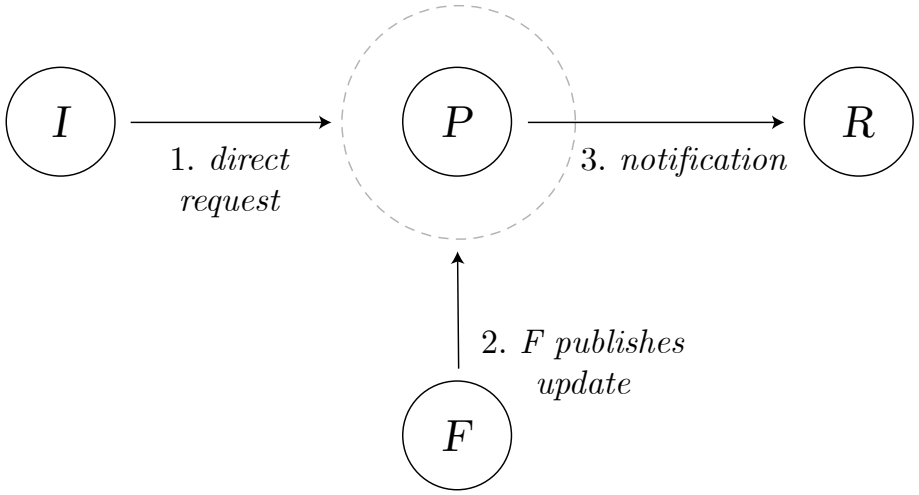


Figure 60: Neighbourhood notification timeline of events. Issuer I mines an identifier for a single owner chunk with themselves as owner such that the chunk address falls in recipient R's neighbourhood. The issuer, being the owner also must sign the identifier against a prefabricated reminder and remember which node should be notified. When P syncs the feed update the notification is sent to R, delivered by the usual push-syncing.

the notification could not be posted before the arrival of the update, otherwise false alarms could be generated by malicious nodes that service the request.

A simple measure is to encrypt the message in the request message symmetrically with a key that is only revealed to the prospective poster when they receive the feed update, for instance, the hash of the feed update identifier. In order to reveal that the notification request needs to be matched on arrival of the feed update, the topic must be left unencrypted, so here we do not encrypt the pss envelope asymmetrically but only the message and symmetrically.

Note that the feed update address as well as identifier can be known if the feed is public, so neighbourhood notifications are to be used with feeds whose subsequent identifiers are not publicly known.

Targeted chunk delivery

Normally, in Swarm's DISC model, chunks are requested with retrieve requests, which are forwarded towards the neighbourhood designated by the requested chunk address (see 2.3.1). The first node on the route that has the chunk will respond with it and the chunk is delivered as a backwarded response travelling back along the same route that the request has taken. In certain cases, however, it may be useful to have a mechanism to request a chunk from an arbitrary neighbourhood where it is known to be stored and send it to an arbitrary neighbourhood where it is known to be needed. A construct called targeted chunk delivery is meant for such a use case: the request is a Trojan pss

message while the response, the delivery, must be a single owner chunk wrapping the requested chunk, with an address mined to fall into the neighbourhood of the recipient.

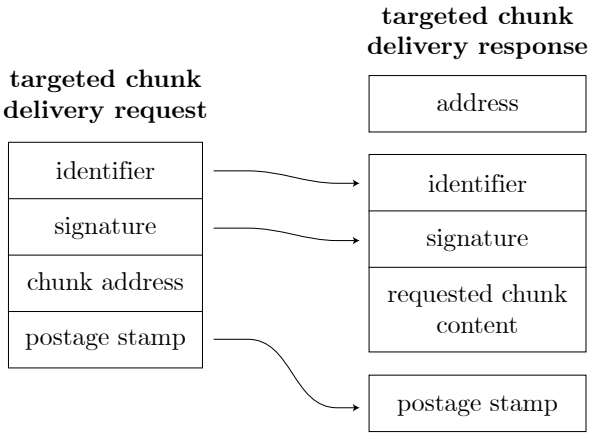


Figure 61: Targeted chunk deliveries are similar to neighbourhood notifications in that they are not only pre-addressed and pre-paid but also pre-signed. Similarly, the identifier is mined so that given issuer I as the owner, it produces an chunk address that falls in recipient R's neighbourhood. Here the issuer signs the identifier against the hash of a content addressed chunk that they want to see posted to R. If the targeted chunk delivery request lands with any node that has the chunk in question, they can use the envelope and the content of the chunk to produce a valid response which is a single owner chunk wrapping a content addressed chunk.

These 'chunk-in-a-soc' responses are structurally similarly to neighbourhood notifications in that the payload's hash is already known (the content address of the chunk requested). The requestor can then sign off on its association with the identifier which, along with the signature, is sent as part of the request, see figure 62. A node that got the request only needs to have the requested chunk stored and they can construct the response as a valid single owner chunk addressed to the recipient's neighbourhood. This makes the request generic, i.e. not fixed to the identity of the prospective poster. Therefore it can be sent to any neighbourhood that requires the content. Even multiple requests can be sent simultaneously: due to the uniqueness of the valid response, multiple responses cannot hurt chunk integrity (see 2.2.3 and 4.3.3). Targeted delivery is used in missing chunk recovery, see 5.2.3.

The difference is that neighbourhood notifications are not giving any new information while targeted chunk delivery supplies the chunk data. Also, a chunk delivery wrapped in a single owner chunk uses no message wrapping and has no topic. Neither request nor response uses encryption. A consequence of not using encryption is that, if there are alternative sources, mining the initial Trojan request has the constraint that the address matches any target.

Table 2 summarises various properties of the three notification-like constructs.

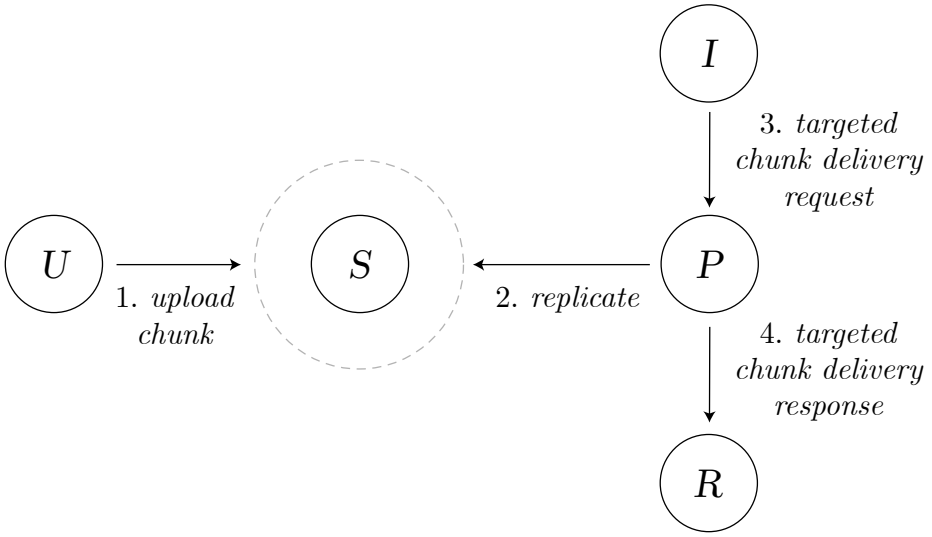


Figure 62: Targeted chunk delivery timeline of events. Uploader U uploads a chunk to Swarm and it lands with a storer node S. Nodes within P replicate (and pin) this chunk. Now, if issuer I wants this chunk delivered to R, it will take the prepaid envelope addressed to R and sends it to the neighbourhood of a known host unencrypted so that anyone who has the chunk can construct the notification and send or store it for later.

type	owner	poster	request encryption	notification
direct	poster	publisher	asymmetric on pss	feed update content
neighbourhood	issuer	any	symmetric on envelope	feed update arrival
targeted delivery	issuer	any	none	chunk content

Table 2: Requests of and responses to types of feed update notifications and targeted chunk delivery.

5

PERSISTENCE

In this chapter we focus on data persistence, i.e. the ways of making sure that content stays available on Swarm. We introduce error coding schemes, which can provide a way to secure availability against churn at a cost of the storage overhead. In particular erasure codes (5.1) and entanglement codes provide redundancy optimised for documents with different access patterns. After introducing the notion of local pinning in 5.2, i.e. the ability to mark content as sticky in your Swarm local storage, we will present ways in which this can help achieve global persistence across the network. We define a missing chunk notification protocol, which allows content maintainers to make sure their published content is restored in the event of some chunks being garbage collected, by proxying retrieval through select pinners of the content.

Finally, in 5.3, we present the holy grail, that is, decentralised file insurance: combining the concepts of the postage lottery and missing chunk notifications, Swarm offers a protocol that enables users to pay other nodes to keep their content available. While ordinary storer nodes are rewarded for storing chunks via the postage lottery, for a higher premium, staked nodes will promise to take responsibility for storing and responding to requests for chunks, accepting the risk of punitive measures that will be imposed on them in the event that the insured data does not remain available to the network.

5.1 REDUNDANCY, LATENCY AND REPAIR

5.1.1 *Error correcting codes*

Error correction codes are commonly used to ensure that data remains available even if parts of the storage system become faulty. In particular, they allow us to construct storage schemes (data encoding and distribution) that solve this problem more efficiently than simple replication. The problem is framed in the context of guaranteeing a certain probability that the data will be retrievable given a model expressing the expected fault conditions in the storage subsystems.

Coding theory in general is often applied to RAIDs and computer hardware architecture synchronising arrays of disks to provide resilient data-centre storage. Erasure codes in particular see the problem as: how does one encode stored data into shards

distributed across n disks so that the entirety of the data remains fully retrievable in the face of a particular probability that one disk becomes faulty. Similarly, in the context of a distributed chunk store, the problem can be reformulated as the question: how does one encode the stored data into chunks distributed across nodes in the network so that the entirety of the data remains retrievable in the face of a particular probability that one chunk is found to be missing.[1]

There are various parameters one can optimise on, principally: storage and bandwidth overhead. Erasure codes are theoretically optimal to minimise storage overhead, but require retrieving large amount of data for a local repair. Entanglement codes, on the other hand, require a minimal bandwidth overhead for a local repair, but at the cost of storage overhead that is in multiples of 100%.

5.1.2 *Redundancy by erasure codes*

The Cauchy-Reed-Solomon erasure code (henceforth CRS, [Bloemer et al., 1995], [Plank and Xu, 2006]) is a *systemic* erasure code which, when applied to a data blob of m fixed-size chunks, produces k extra chunks (so called *parity chunks*) of the same size in such a way that any m out of $n = m + k$ fix-sized chunks are enough to reconstruct the original blob. The storage overhead is therefore given by $\frac{k}{m}$.[2]

Both the encoding and the decoding of CRS codes takes $O(mk)$ time, where m is the number of data chunks, k is the number of additional chunks as well as the maximum number of chunks that can be lost without losing decodeability. If k is defined as a given fraction of m which is necessary for guaranteeing a certain probability of retrievability under the condition of a fixed probability p of losing a single chunk, the time complexity of CRS codes becomes $O(n^2)$, which is unacceptable for large files.

5.1.3 *Per-level erasure coding in the Swarm chunk tree*

Swarm uses a hierarchical Merkle tree [Merkle, 1980] to reorganise data into fixed sized chunks which are then sent off to the Swarm nodes to store. Let h be the byte length of the hash used, and let b be the branching factor. Each vertex represents the root hash of a subtree or, at the last level, the hash of a $b \cdot h$ long span (one chunk) of the document. Generically we may think of each chunk as consisting of b hashes.

During normal Swarm lookups, a Swarm client performs a lookup for a hash value and receives a chunk in return. This chunk in turn constitutes another b hashes to be

1 The distributed chunk store model uses fixed-sized chunks which can only be either completely lost or completely undamaged. Since the Swarm content storage uses the hashes of chunks as their addresses, and since the chunks are stored by custodian nodes that are randomly distributed in the same address space, we are safe to assume that a particular storage allocation is independent of all the other nodes and data. This assures that recovering a chunk at any point can practically be thought of as potentially failing with equal and independent probability.

2 There are several open source libraries that implement Reed Solomon or Cauchy-Reed-Solomon coding. See [Plank et al., 2009] for a thorough comparison.

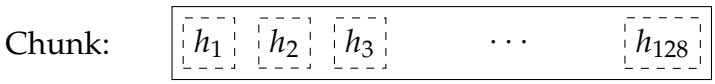


Figure 63: A Swarm chunk consists of 4096 bytes of the file or a sequence of 128 subtree hashes.

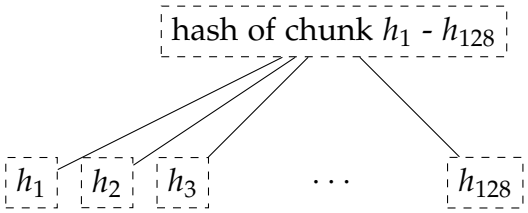


Figure 64: A generic node in the tree has 128 children.

looked up and retrieve another b chunks and so on until the chunks received belong to the actual document (see figure 65).

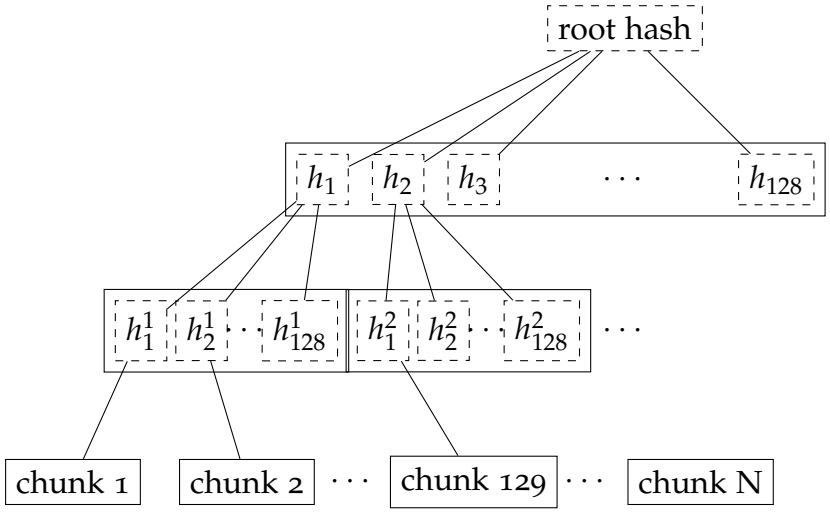


Figure 65: The Swarm tree is the data structure encoding how a document is split into chunks.

While off-the-shelf erasure coding could be used for documents uploaded into Swarm, this solution has immediate problems. Apart from the quadratic complexity of encoding, chunking the CRS-encoded data blob with the BMT chunker would result in certain chunks being more vulnerable, since their retrieval is dependent on the retrieval of the chunks that encode all their ancestor nodes in the chunk tree.

This prompted us to try and align the notion of Swarm chunk with the chunk used in the CRS scheme which led us to encode redundancy directly into the Swarm tree. This is achieved by applying the *CRS scheme* to each set of chunks that are children of a node in the Swarm tree.

The *chunker* algorithm incorporating CRS encoding works in the following way when splitting the document:

1. Set the input to be the data blob.

2. Read the input one chunk (say fixed 4096 bytes) at a time. Count the chunks by incrementing a counter i.

3. Repeat step 2 until either there's no more data (note: the last chunk read may be shorter) or $i \equiv 0 \bmod m$.

4. Use the CRS scheme on the last $i \mod m$ chunks to produce k parity chunks resulting in a total of $n \leq m + k$ chunks.

5. Calculate the hashes of all these chunks and concatenate them to result in the next chunk (of size $i \mod m$ of the next level. Record this chunk as next.

6. If there is more data repeat 2.

7. If there is no more data but the next level data blob has more than one chunk, set the input to this and repeat from 2.

8. Otherwise record the blob to be the root chunk.

A typical piece of our tree would look like this (see figure 66).

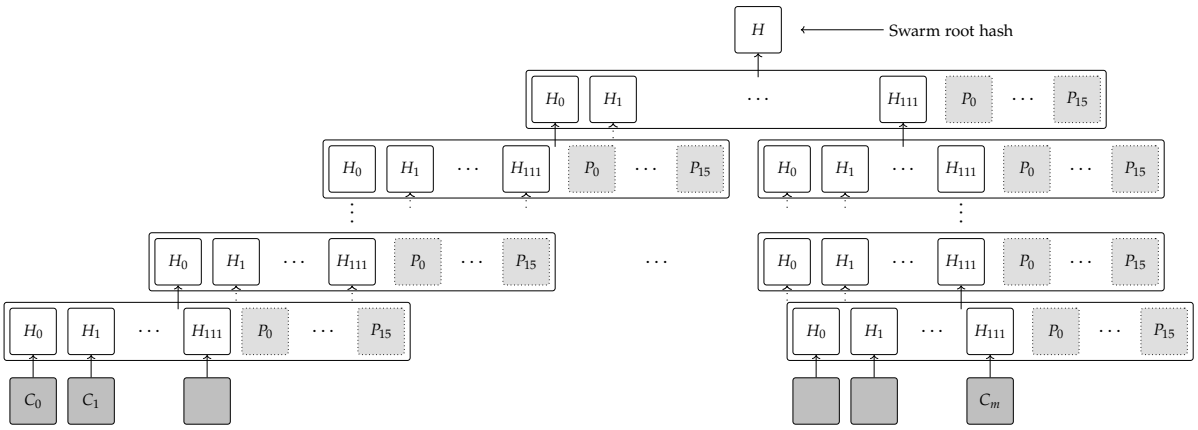


Figure 66: The Swarm tree with extra parity chunks using 112 out of 128 CRS encoding. Chunks p_0 through p_{15} are parity data for chunks H_0 through H_{111} on every level of intermediate chunks.

This pattern repeats itself all the way down the tree. Thus hashes H_{m+1} through H_{127} point to parity data for chunks pointed to by H_0 through H_m. Parity chunks P_i do not have children and so the tree structure does not have uniform depth.

Incomplete batches

If the number of file chunks is not divisible by m, we cannot proceed with the last batch in the same way as the others. We propose that we encode the remaining chunks with an erasure code that guarantees at least the same level of security as the others. Note that this is not as simple as choosing the same redundancy. For example a

50-out-of-100 encoding is much more secure against loss than a 1-out-of-2 encoding even though the redundancy is 100% in both cases. Overcompensating, we still require the same number of parity chunks even when there are fewer than m data chunks.

This leaves us with only one corner case: it is not possible to use our m-out-of-n scheme on a single chunk ($m = 1$) because it would amount to $k + 1$ copies of the same chunk. The problem is that any number of copies of the same chunk all have the same hash and therefore are automatically deduplicated. Whenever a single chunk is left over ($m = 1$) (this is always the case for the root chunk itself), we replicate the chunk as the payload of one of more single owner chunk with an address that is deterministically derivable from the content owners public key and the original root hash, thus enabling us to provide an arbitrary level redundancy for storage of data of any length.

Upper limit of retrieval latencies

When downloading a file with erasure coding, data subsumed under an intermediate chunk can be recovered having any m out of $m + k$ children. A downloader can initiate requests for all $m + k$ and will need to wait only for the first m to be delivered in order to proceed. This technique can effectively shield unavailability of chunks due to occasional faults like network contention, connectivity gaps, and node churn; prohibitively overpriced neighbourhoods or even malicious attacks targeting certain localities. Given a particular fault model for churn and throughput, erasure codes can be calibrated to *guarantee an upper limit on retrieval latencies*, a strong service quality proposition.

5.2 PINNING, REUPLOAD AND MISSING CHUNK NOTIFICATIONS

This section introduces the notion of pinning, i.e. locally sticky content protected from the garbage collection routines of its storage nodes (5.2.1). In 5.2.2, we discuss how pinners of content can play together to pin content for the entire network globally. 5.1 defines a recovery protocol that downloaders can use to notify storers of missing chunks belonging to content they are responsible for pinning globally. With such on-demand repair and the uninterrupted download experience it enables, recovery implements a poor man's persistence measure which can ensure network wide availability of specific chunks without requiring the financial outlay of insurance.

5.2.1 *Local pinning*

Local pinning is the mechanism that makes content sticky and prevents it from being garbage collected. It pins the content only in the node's local storage to enable local persistence of data and speedy retrieval. Pinning is applied at the chunk level in the

local database of the client and exposes an API for users to pin and unpin files and collections in their local node (see 6.1.4 and 10.2.6).

In order to pin all the chunks comprising files, the clients need to keep a reference count for each chunk which incremented and decremented when the chunk is pinned and unpinned respectively. As long as the reference count is non-zero, the chunk is considered to be part of at least one document that is pinned and therefore immune to garbage collection. Once a chunk is pinned, only when the reference count returns to zero, after it has been respectively unpinned for every time it has been pinned, is the chunk once again considered for garbage collection.

Local pinning can be thought of as a feature which allows Swarm users to mark specific files and collections as important, and therefore not removable. Pinning a file in local storage will also make it always accessible for the local node even without an internet connection. As a result using pinned content for storage of local application data enables an offline-first application paradigm, with Swarm automatically handling network activity on re-connection. However, since if the chunk is not in the node's area of responsibility, local pinning by itself is not enough to ensure the chunk generally retrievable to other nodes, since the pinner is not in the Kademlia neighbourhood where the chunk is meant to be stored and hence where it will be searched for when the chunk is requested using the pull-sync protocol. In order to provide this functionality, we must implement a second half of the pinning protocol.

5.2.2 *Global pinning*

If a chunk is removed as a result of garbage collection by storers in its nominated neighbourhood , local pinning in nodes elsewhere in the network will offer no way to retrieve it alone. In order for pinning to aid global network persistence, two issues must be tackled:

- notify global pinners of a missing chunk belonging to the content they pin – so they can re-upload it,
- maintain retrievability of chunks that are garbage collected but globally pinned – so that downloaders experience no interruption.

One naive way to achieve this is to periodically check for a pinned chunk in the network and re-upload the contents if it is not found. This involves a lot of superfluous retrieval attempts, has immense bandwidth overhead and ultimately provides no reduction in latency.

An alternative, reactive way is to organise notifications to the pinner which are somehow triggered when a user accesses the pinned content and finds a chunk to be unavailable. Ideally, the downloader notifies the pinner with a message that triggers (1) the re-upload of the missing chunk, and (2) the delivery of the chunk in response to the request of the downloader.

Fallback to a gateway

Let us assume that a set of pinner nodes have the content locally pinned and our task is to allow fallback to these nodes. In the simplest scenario, we can set up the node as a gateway (potentially load-balancing to a set of multiple pinner nodes): Users learn of this gateway if it is included in the manifest entry for the file or collection. If the user is unable to download the file or collection from the Swarm due to a missing chunk, they can simply resort to the gateway and find all chunks locally. This solution benefits from simplicity and therefore likely the first global persistence milestone to be implemented.

Mining chunks to pinners' neighbourhoods

The second, brute force solution is more sophisticated in that the publisher can construct the chunks of a file in such a way that they all fall within the neighbourhood of the pinner (or of any pinner node in the set). In order to do this, the publisher needs to find an encryption key for each chunk of the file so that the encrypted chunk's address matches one of the pinners on at least their first d bits, where d is chosen to be comfortably larger than that pinner's likely neighbourhood depth.[3]

These solutions can already accomplish persistence, but they reintroduce a degree of centralisation and also require publishers to maintain and control server infrastructure. Surely, they also suffer from the usual drawbacks of server-client architecture, namely decreased fault-tolerance and the likelihood of compromised performance due to the fact that requests will be more concentrated. These solutions also do not address the question of how pinner nodes are provided the content and disregard any privacy considerations. For these reasons, although the low-level pinning API is provided for swarm users, usage of it is considered to be a less desirable alternative to the gold standard of incentivised file insurance discussed elsewhere. As such use cases should be carefully considered to ensure that they would not benefit from the enhanced privacy and resilience provided by the incentivised system.

5.2.3 *Recovery*

In what follows we describe a simple protocol for notifying pinners of the loss of a chunk, which they can react to by (1) re-uploading the lost chunk to the network and at the same time (2) responding to the notifier by delivering to them the missing chunk. The procedures relating to the missing chunk notification protocol are specified in detail in 7.6.5 while related data structures are formalised in 7.6.5.

3 Note that in case that the pinner nodes do not share infrastructure and the mined chunks need to be sent to pinners with the push-sync protocol, then each postage batch used will be used at most n times. Although this non-uniformity implies a higher unit price, if pinners pin out of altruism (i.e. not for compensation), only a minimum postage price needs to be paid.

If replicas of a chunk are distributed among willing hosts (pinners), then downloaders not finding a chunk can fall back to a recovery process requesting it from one of these hosts using the missing chunk notification protocol called prod. As usual, the resolutions of this acronym capture the properties of the protocol:

- *protocol for recovery on deletion* - A protocol between requestor and pinners to orchestrate chunk recovery after garbage collection. The process is triggered by downloaders if they fail to retrieve a chunk.
- *process request of downloader* - Recovery hosts continuously listen for potential recovery requests from downloaders.
- *provide repair of DISC* - Prod provides a service to repair the Swarm DISC, in the event of data loss.
- *pin and re-upload of data* - Hosts pin the data and re-upload it to its designated area when prompted with a recovery request.
- *prompt response on demand* - If the requestor demands a prompt direct response, the host will post the missing chunk using a recovery response envelope and accompanying postage stamp.

Recovery hosts

Recovery hosts are pinners that are willing to provide pinned chunks in the context of recovery. These pinners are supposed to have indicated that they will take on this task to the publisher and to have downloaded all the relevant chunks and have pinned them all in their local instance.

The publisher that wishes to advertise its publication as globally pinned or repairable would then gather the overlay addresses of these volunteering recovery hosts. In fact, it is sufficient to collect just the prefixes of the overlay with a length greater than Swarm's depth. We will call these partial addresses recovery targets.

Publishers will advertise the recovery targets for their content to the consumers of their data via a feed called a recovery feed. This feed can be tracked by consumers of the publisher's data based on the convention of constructing the topic using a simple sequential indexing scheme (see 7.6.5).

Recovery request

Once the recovery hosts overlay targets are revealed to the downloader node, upon encountering a missing chunk, they should "prod" one of the recovery hosts by sending them a notification called a recovery request. This instance of targeted chunk delivery (see 4.4.4) is a public unencrypted Trojan message containing at least the missing chunk address, see figure 68.

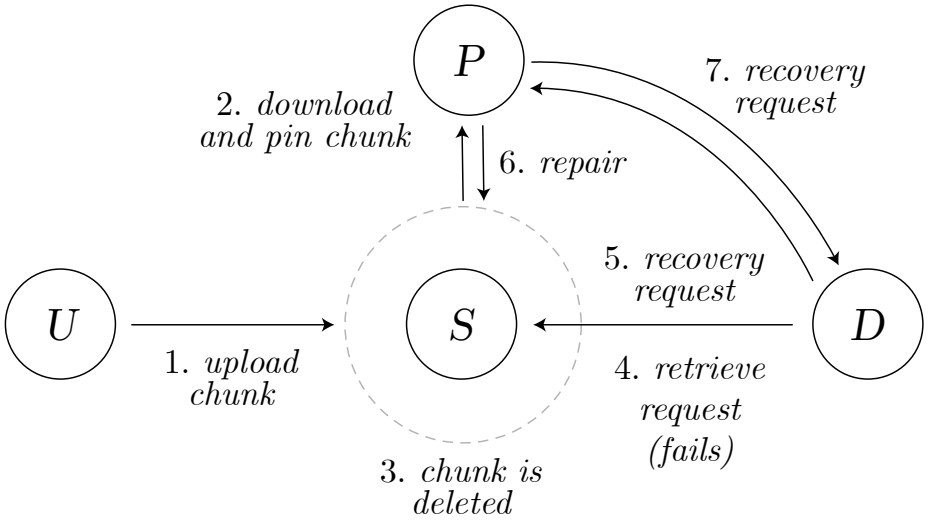


Figure 67: The missing chunk notification process is similar to the targeted chunk delivery. Here a downloader mines the identifier to match their own address, i.e. a self-notification. If downloader D encounters a missing chunk (a request times out), they send a recovery request to one of several neighbourhoods where there is a chance of finding the chunk. A successful response then will comprise a single owner chunk wrapping the missing chunk which is sent to the downloader. The chunk is also re-uploaded to the proper place in the network.

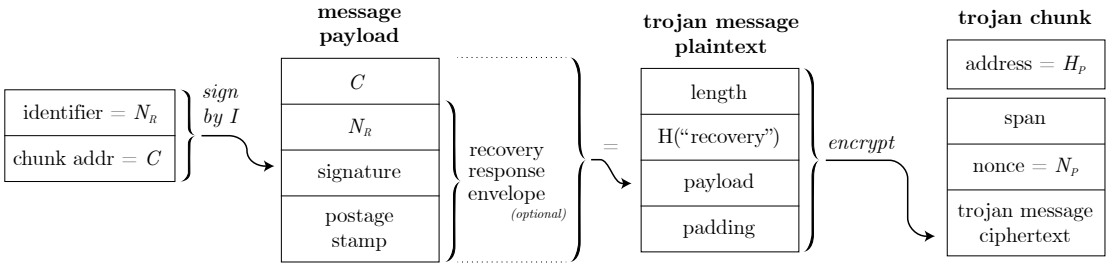


Figure 68: A recovery request is a Trojan chunk that is used as missing chunk notification. It is unencrypted and its payload is structured as a pss message containing the address of the chunk to be recovered. Optionally, it also includes a special recovery response envelope, an identifier with a signature attesting to the association of the identifier and the missing chunk hash.

In order to create a recovery request, a downloader needs to (1) create the payload of the message (2) find a nonce which when prepended to the payload results in a content address that matches one of the recovery targets indicated by the publisher. The matching target means that by push-syncing the chunk, it will be sent to the neighbourhood of the recovery host represented by a target prefix.

If the targeted recovery host is online, they will receive the recovery request, extract the missing chunk address, retrieve the corresponding chunk that is pinned in their local storage and re-upload it to the network with a new postage stamp.

Recovery response envelope

A recovery response envelope serves to provide a way for recovery hosts to be able to respond directly to the originator of the recovery request, promptly and without associated cost or computational burden. It is an instance of targeted chunk delivery response (see 4.4.4), an addressed envelope construct which is neutral to the poster but fixed for the content. The request message includes the components prospective posters will need to create the valid targeted chunk delivery response: an identifier with a signature attesting to the association of the identifier and the missing chunk hash. The identifier is chosen so that the address of the single owner chunk (the hash of the id with the owner account) falls into the requestor's neighbourhood and so it is automatically delivered there by the network's implementation of the push-sync protocol.

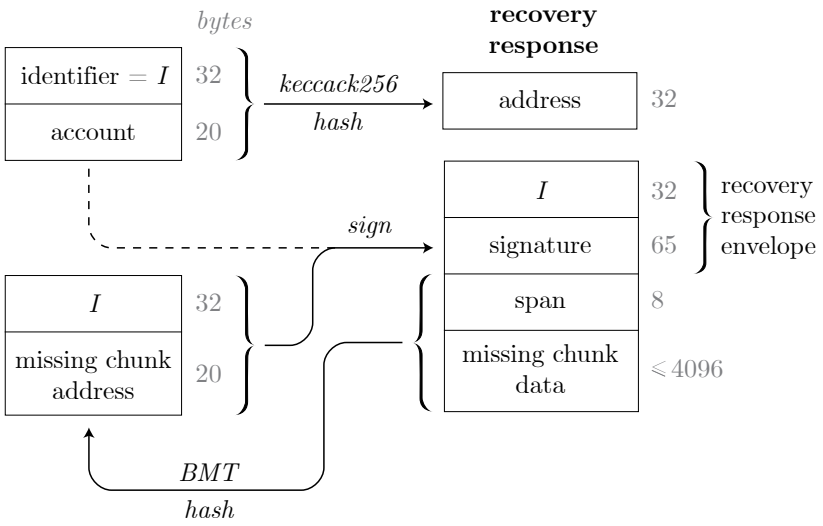


Figure 69: Recovery response is a single owner chunk that wraps the missing chunk. Anyone in possession of the headers and the chunk data is able to construct a valid single owner chunk and send it to the downloader for free.

If the targeted recovery host is online, they will receive the recovery request and extract the missing chunk address as well as the identifier and the signature from the

recovery response. After retrieving the corresponding chunk they pinned in their local storage they can simply create the recovery response. The hash of the identifier and the payload hash concatenated together forms the plain text of the signature. The signature received in the recovery request allows her to recover the public key of the requestor. From this public key they are able to calculate the requestor's address and finally, hashing this with the identifier results in the address of the single owner chunk. This scheme makes more sense if a postage stamp is also attached to the request which can be sent along with the recovery response as requestors are able to cover the Swap costs of returning the chunk which compensates the global pinner. In this way, it is possible for downloaders to intelligently cover the operating costs of nodes using micro-payments, hence ensuring the availability of data for the whole of the network.

5.3 INSURANCE

5.3.1 *Insurance pool*

We saw in 3.3.4 that the reward mechanism for postage lottery can also be used to facilitate and enforce insurance against lost data: where staked nodes take on the responsibility to store certain chunks staked against financial assets using the blockchain.

Finally, we will discuss 1) how chunks are insured 2) how chunks travel across the network to the insurer 3) how later, any third party is able to audit the receipt for a chunk and, if it is found to be missing, initiate a process of litigation.

Let us imagine an insurance pool, a smart contract where registrants contribute stake. The pool represents nodes that, in return for a higher storage premium, stand to lose their stake if they lose a chunk.

5.3.2 *Buying insurance on a chunk*

5.3.3 *Uploading with insurance*

5.3.4 *Accessing receipts*

The manifest entry for the insured file contains the reference to the data structure that maps insurer addresses to their public keys at the time of insuring the file. Once these pivots are known, any third party can calculate which insurer is responsible for each chunk. Once the insurer is known, one can look up their receipt store, the current receipt store root is easily found by simply looking up the insurer's receipt store feed.[4] The actual receipt for the chunk can then be retrieved by traversing the receipt store using the original chunk address as the key. The insurer is implicitly

4 In fact, it is not necessary to have the most up to date root, any state that is later than the start time of insurance should work.

responsible for insuring all the chunks of their receipt store as well as the receipt store feed. So if any chunk is found missing during this lookup procedure, the Merkle proof of segment that corresponds to the address of the missing chunk can be used to litigate. Otherwise, we can assume that the lookup is successful and the receipt is obtained by the user who can thus initiate litigation by submitting the receipt.

Figure 70: From retrieval to litigation

Figure 70 gives a complete timeline of events starting from a third party user retrieving a file all the way to the file original insurer being punished.

6

USER EXPERIENCE

This chapter approaches Swarm features introduced in the previous chapters from the user's perspective. In 6.1, we discuss the user experience related to configuring uploads including aspects of postage, pinning, erasure coding, as well as tracking the progress of chunks propagating into the network with the help of upload tags. Then we turn to the user's experience of the storage API, uploading collections in 6.2 then turn to a description of the options available for communication in 6.3. Finally, in 6.4, we present a complete framework for web3 dapp development.

6.1 CONFIGURING AND TRACKING UPLOADS

Upload is the most important interface for Swarm. Section 6.1.1 presents the request headers (or alternatively, query parameters) used for configuring the upload APIs. In 6.1.2 we introduce upload tags which can be used to track the state of the entire upload and displayed using progress bars together with an estimate of the expected time until the process will finish. Tags also record partial root hashes which can then be used to resume an upload in the eventuality it is interrupted before it is complete. In 6.1.3, we sketch the scenarios relating to payment for upload and dispersal into the network, in particular how users can buy and attach stamps to chunks. Finally, 6.1.4 runs through optional parameters such as encryption, pinning and erasure codes.

6.1.1 *Upload options*

The local http proxy offers the `bzz` URL scheme as a storage API. The API is discussed further in 6.2 and formally specified in 10.2. Requests can specify Swarm specific options such as:

- `tag` – use this upload tag – generated and returned in response header if not given.
- `stamp` – upload using this postage subscription – if not given, the one used most recently is used.

- `encryption` – encrypt content if set, if set to a 64-byte hex value encoding a 256 bit integer, then that is used instead a randomly generated key.

- `pin` – pin all chunks of the upload if set.

- `parities` – apply CRS erasure coding to all intermediate chunks using this number of parities per child batch.

These options can be used as URL query parameters or specified as headers. The name of the header is obtained by capitalising the parameter name and prefixing it with `SWARM-`, i.e. `SWARM-PARITIES`.

6.1.2 *Upload tags and progress bar*

When uploading a file or collection, it is useful to the user to know when the upload is complete in the sense that all newly created chunks are synced to the network and arrived at the neighbourhood where they can be retrieved. At this point, the uploader can "disappear", i.e. can quit their client. A publisher can disseminate the root hash and be assured, the file or collection is retrievable from every node on Swarm.

Since the push-sync protocol provided statements of custody receipts for individual chunks we only need to collect and count those for an upload. Tracking the ratio of sent chunks and returned receipts provides the data to a *progress bar*. An upload tag is an object representing an upload and tracking the progress by counting how many chunks have reached a particular state. The states are:

- *split* – Number of chunks split; count chunk instances.

- *stored* – Number of chunks stored locally; count chunk instances.

- *seen* – Count of chunks previously stored (duplicates).

- *sent* – Number of distinct chunks sent with push-sync.

- *synced* – Number of distinct chunks for which the statement of custody arrived.

With the help of these counts, one can monitor progress of 1) chunking 2) storing 3) push-syncing, and 4) receipts. If the upload tag is not specified in the header, one is randomly generated and is returned as a response header after the file is fully chunked. In order to monitor 1) and 2) during the upload, the tag needs to be created before the upload and supplied in the request header. Thus, the tag can be queried concurrently while the uploaded content is being processed.

Known vs unknown file sizes

If the file size is known, the total number of chunks can be calculated, so that progress of chunking and storage can be meaningful from the beginning in proportion to the total.

If the size and total number of chunks split is not known prior to the upload, the progress of split is undefined. After the chunker has finished splitting, one can set the total count to the split count and from that point on, a percentage progress as well as ETA are available for the rest of the counts.

Note that if the upload also includes a manifest, the total count will serve only as an estimation until the total is set to the split count. This estimation converges to the correct value as the size of the file grows.

Duplicate chunks

Duplicate chunks are chunks that occur multiple times within an upload or across uploads. In order to have a locally verifiable definition, we define a chunk as a duplicate (or seen) if and only if it is already found in the local store. When chunks enter the local store via upload they are push synced, therefore seen chunks need not be push-synced again.

In other words, only newly stored chunks need to be counted when assessing the estimated time of syncing an upload. If we want progress on sent and synced counts, they must report completeness in proportion to stored distinct chunks.

Tags API

The http server's `bzz` URL scheme provides the `tags` endpoint for the tags API. It supports creating, listing and viewing individual tags. Most importantly there is an option to track the changes of a tag in real time using http streams. The tag API is specified in 10.2.5.

6.1.3 *Postage*

To impose a cost on uploads and efficiently allocate storage resources in the network, all uploads must be paid for. This is somewhat unusual for the web, so a novel user experience needs to be developed. The closest and most familiar metaphor is a subscription.

Postage subscriptions

The user creates a subscription for a certain amount of time and storage (e.g. 1 month for 100 megabytes) and pays for it according to a price they learn from the client software (this is similar to how transaction fees are determined for the blockchain). Estimates for price can be read from the postage lottery contract. Subscriptions are

named, but these names are only meaningful locally. The API will offer a few default options for the chosen storage period, on a logarithmic scale, e.g.:

- *minimal (a few hours)* – Useful for immediate delivery of pss messages or single owner chunks part of ephemeral chat or other temporary files.
- *temporary (week)* – Files or mailboxed messages meant not to be stored long but to be picked up by third parties asynchronously.
- *long term (year)* – Default long term storage.
- *forever (10 years)* – Important content not to be lost/forgotten; to survive the growth of the network and subsequent increase of batch depth even if the uploader remains completely offline.

When the user uploads files, they can also indicate which subscription to use as the value of the `stamp` upload parameter. If this is not given, the most recent one is used as default. If the size of the upload is known, the user can be warned if it is larger than the postage capacity.

Postage reuse strategies

As mentioned in 3.3.1, encryption can be used to mine chunks to make sure they fall into collision slots of postage batches. This strategy of mining chunks into batches makes sense for users that only want to upload a particular file/collection for a particular period or want to keep the option open to change the storage period for the file/collection independently of other uploads.

The alternative is to prepay a large set of postage batches and always keep a fixed number of them open for every period. This way we can make sure that we have a free collision slot in one of the batches for any chunk. The postage batches are ordered by time of purchase and when attaching a stamp to a chunk, the first batch is used that has a free slot for the chunk address in question. This more profitable strategy is used most effectively by power users including insurers who can afford to tie up liquidity for long time periods. The choice of strategy must be specified when creating a subscription.

The postage subscription API

In order to manage subscriptions, the `bzz` URL-scheme provides the `stamp` endpoint for the postage API (see 10.2.8). With this API the user can create postage subscriptions, list them, view them, top them up, or drain and expire them.

When checking their subscription(s), the user is informed how much data has already been uploaded to that subscription and how long it can be stored given the current price (e.g. 88/100 megabytes for 23 days). If the estimated storage period is low, the files using the subscription are in danger of being garbage collected and therefore, to prevent that, topping up their subscription is instructive.

6.1.4 *Additional upload features*

Additional features such as swap costs, encryption, pinning and erasure coding can be set on upload using request headers.

Swap costs

Uploading incurs swap costs coming from the push-sync protocol showing up in SWAP accounting (3.2.1). On every peer connection, if the balance tilts beyond the effective payment threshold, a cheque is to be issued and sent to the peer. If, however, the checkbook contract does not exist or is lacking sufficient funds to cover the outstanding debt, the peer connection is blocked. This may lead to unsaturated Kademlia and during that time, the node will count as a light node. The download can continue potentially using a smaller number of peers but, since some chunks will need to be sent to peers not closer to the request address than the node itself, the average cost of retrieving a chunk will be higher.

It is useful to show this average number to the user as well as the average swap balance and the number of peer connections not available due to insufficient funds. This is accomplished through the SWAP API specified in 10.2.7.

Encryption

Encrypted upload is available by setting the upload option `encryption` to a non-zero value. If the value is a hexadecimal representation of a 32 byte seed, that is used as a seed for encryption. Since encryption and decryption are handled by Swarm itself, it must only be used in situations where the transport between the http client and Swarm's proxy server can be guaranteed to be private. As such, this is not be used on public gateways, but only through local Swarm nodes or those that are private with well configured TLS and self-signed certificates.

Pinning

When uploading, the user can specify directly if they want the content to be pinned locally by setting the `pin` upload option to a non-zero value. Beyond this, the local http server's `bzz` URL scheme provides the `pin` endpoint for the pinning API (see 10.2.6). With this API, users can manage pinned content. A `GET` request on the pinning endpoint gives a list of pinned files or connections. The API also accepts `PUT` and `DELETE` requests on a hash (or domain name that resolves to a hash) to pin and unpin content, respectively.

When pinning a file or collection, it is supposed to be retrieved and stored. Pinning triggers traversal of the hash tree and increments the reference count on each chunk; if a chunk is found missing locally, it is retrieved. After pinning, the root hash is saved locally. Unpinning triggers traversal of the hash and decrements the reference count of

each chunk; if a chunk is found missing locally, it is ignored and a warning is included in the response. After unpinning, the hash is removed from the list of pinned hashes.

Erasure coding

Erasure coding (see 5.1) is triggered on an upload by setting the `parities` upload option to the number of parity chunks among the children of each intermediate chunk. It is important that erasure coded files cannot be retrieved using the default Swarm downloader so erasure coding settings should be indicated in the encapsulating manifest entry by setting the `crs` attribute to the number of parity chunks (see 7.4.2).

6.2 STORAGE

In this section, we introduce the storage API provided through the `bzz` family of URL schemes by Swarm's local HTTP proxy.

6.2.1 *Uploading files*

The `bzz` scheme allows for uploading files directly through the `file` API. The file is expected to be encoded in the request body or in a multipart form. All the query paramaters (or corresponding headers) introduced in 6.1.1 can be used with this scheme. The `POST` request chunks and uploads the file. A manifest entry is created containing the reference to the uploaded content together with some attributes reflecting the configuration of the upload, e.g. `crs` specifying the number of parities per batch for erasure codes. The manifest entry is provided as a response. The upload tag that we can use to monitor the status of the upload will contain the reference to the uploaded file as well as the manifest entry.

Appending to existing files

The `PUT` request implements appending to pre-existing data in the swarm and expects the URL to point to that file. If the file is referenced directly, settings such as the number of parities for erasure coding are taken from upload headers otherwise those in the enclosing manifest entry are used. The response is the same as in the case of `POST` request.

Resuming incomplete uploads

As a special case, append is used when resuming uploads after a crash or user initiated abort. For this, one needs to keep track of partial uploads by periodically recording root hashes on the upload tag. When the upload tag specified in the header is not complete, we assume the request is meant to resume the same upload. The last recorded root hash in the tag is used as an append target: the right edge of the existing

file is retrieved to initialise the state of the chunker. The file sent with the request is read from where the partial upload ended, i.e. the offset is set to the span of the root chunk recorded on the tag.

6.2.2 *Collections and manifests*

As described in 4.1.2 and specified in 7.4.2, a manifest can represent a generic index mapping string paths to entries, and therefore can serve as the routing table to a virtual web-site, as the directory tree of a file collection or as a key–value store.

Manifest entries vs singleton manifests

Manifest entries are also needed for single files since they contain key information about the file such as content type, erasure coding which is needed for correct retrieval. A manifest that would contain a single entry to a file on the empty path is called a singleton manifest. This contains no more information than the the entry itself.

The http server provides the `bzz` URL scheme that implements the collection storage API (see 10.2). When a single file is uploaded via a `POST` request the `bzz` scheme, the manifest entry is created and stored and the response contains the reference to it.

Uploading and updating collections

The API provided by the `bzz` URL scheme supports uploading and updating collections too. The `POST` and `PUT` requests expect a multipart with a tar stream. The directory tree encoded in the tar stream is translated into a manifest while all the files are chunked and uploaded and their Swarm reference is included in the respective manifest entry. The `POST` request uploads the resulting manifest and responds with its Swarm reference. The `PUT` request requires the request URL to reference an already existing manifest which is updated with the one coming from the tar stream. Updating here means all the new paths are merged with the paths of the existing manifest. In case of identical paths, the entry coming from the upload replaces the old entry.

The API enables inserting a path into a manifest, updating a path in a manifest (`PUT`) and deleting (`DELETE`) a path from a manifest. In case of a `PUT` request, a file is expected in the request body which is uploaded and its manifest entry is inserted at the path present in the URL. If the path already existed in the manifest, the entry is replaced with the entry generated with the upload, implementing an update; otherwise if the path is new, it implements an insert.

The collection API supports the same headers as the file upload endpoint, namely the ones configuring postage subscription, tags, encryption, erasure coding and pinning.

Updating manifests directly

Manipulating manifests is also supported directly: the `bzz` URL scheme `manifest` endpoint supports the `PUT` and the `DELETE` methods and behaves similarly to the

collection endpoint except that it does not deal with the files referenced in the entries. The URL path is expected to references a manifest with a path p. For `PUT`, the request body requires a manifest entry which will be placed at path p. The chunks needed for the new manifest are created and stored and the root hash of the new manifest is returned in the response. The `DELETE` method expects empty request body and deletes the entry on the path from the manifest: i.e., it creates a new manifest with the referenced path in the URL missing.

The `POST` request directly on the manifest API endpoint installs a manifest entry. Practically, calling manifest `POST` on the output of file `POST` is equivalent to `POST` on the generic storage endpoint.

Given manifest references a and b , sending a `POST` request on `manifest/merge/<a>/<b>` merges b onto a (merge with giving preference to b in case of conflicts), creates and stores all the chunks constituting the merged manifest and returns its root reference as a response.

In case the URL path references a manifest, another manifest is accepted in the request body which is then merged into the one referenced. In case of conflicts, the one that is uploaded wins.

6.2.3 *Access control*

Access control was described in 4.2 and has specs in 7.5. The `bzz` URL scheme provides the `access` API endpoint for access control (AC, see the API specification in 10.2.9). This API is meant as a convenience for users and supports putting a file/collection/site under access control as well as adding and removing grantees.

If the URL path references a collection manifest, then a `POST` request with access control (AC) settings sent as JSON encoded request body will encrypt the manifest reference and wrap it with the submitted AC settings in a so called root access manifest. This manifest is then uploaded and the unencrypted reference to it is returned as the response body.

If the URL path references a root access manifest and the access control settings specify an ACT then this can be created or updated using `POST`, `PUT` and `DELETE` requests. All requests expect a JSON array of grantee public keys or URLs in the request body. If the grantee is referenced by URL, the resolver is used to extract the owner public key through ENS.

The `POST` request will create the ACT with the list of grantees. `PUT` will update an existing ACT by merging the grantees in the body. `DELETE` removes all the grantees listed in the request body. The new ACT root hash is then updated in the root access manifest which is uploaded and its Swarm address is returned in the response. See the specification in 10.2.9 for more detail.

6.2.4 *Download*

Downloading is supported by the `bzz` URL scheme. This URL scheme assumes that the domain part of the URL is referencing a manifest as the entry point.

Although the title of this section is download, the same processes are at work if a file is only partially retrieved. As shown in 4.1.1 and 7.4.1, random access to a file at arbitrary offset is supported at the lowest level. Therefore GET requests on a URL pointing to a file accept range queries in the header. The range queries will trigger the retrieval of all but only those chunks of the file that cover the desired range.

Retrieval costs

Downloading involves retrieving chunks through the network which in turn entails costs showing up as SWAP accounting (3.2.1). On every peer connection, if the balance tilts beyond the effective payment threshold, a cheque is to be issued and sent to the peer. If, however, the checkbook contract does not exist or is lacking sufficient funds to cover the outstanding debt, the peer connection is blocked. This may lead to non-saturated Kademlia and during that time, the node will count as a light node. The download can continue potentially using a smaller number of peers, but since some chunks will need to be sent to peers not closer to the request address than the node itself, the average cost of retrieving a chunk will be higher.

It is useful to show this average number to the user as well as the average swap balance and the number of peer connections not available to receive retrieve requests due to insufficient funds. This is done through the SWAP API specified in 10.2.7 that is provided on the `swap` endpoint of the `bzz` URL scheme.

Domain Name Resolution

The domain part of the URL can be a human readable domain or subdomain with a top level domain (TLD) extension. Depending on the TLD, various name resolvers can be invoked. The TLD `eth` is linked to the Ethereum Name Service contract on the Ethereum main chain. If you register a Swarm hash to an ENS domain, Swarm is able to resolve that by calling the ENS contract as a nameserver would.

Authentication for access control

If the domain resolved referenced a root access manifest, the URL retrieved is under access control (see 4.2). Depending on the credentials used, the user is prompted for a password and possibly a key-pair. The Diffie–Hellmann shared secret is hashed with a constant to derive the lookup key and another one to derive the access key decryption key. The Swarm address of the ACT manifest root chunk is taken from the root access manifest. Next the lookup key is appended the ACT address and the resulting URL is used to retrieve the manifest entry. The reference on this entry is then decrypted

with the access key decryption key, and the resulting access key is used to decrypt the original encrypted reference found in the root access manifest.

Next the manifest entry matching the path of the URL is retrieved. The following attributes are taken into account:

- `crs` – structure with attributes needed to use CRS erasure coding for retrieval, e.g. number of parity chunks.
- `sw3` – structure with attributes needed for litigation, i.e. challenge insurer on a missing chunk.

Erasure coded files

If the `crs` attribute is given, there are two strategies to follow:

- *fallback* – For all intermediate chunks, the CRS parity chunks are ignored first and only if there is a missing non-parity chunk will they be used. This saves some bandwidth and the corresponding costs at the expense of speed: if a chunk is missing, fallback to CRS is delayed.
- *race* – for all intermediate chunks, the CRS parities are retrieved right away together with the rest of the children chunks. The first n (assuming k as value of `crs`, i.e, n out of $n + k$ coding scheme) chunks will be able to reconstruct all the n real children. This is equivalent to saying that the k slowest chunk retrievals can be ignored.

The default strategy choice is `race`, if the user wants to save on download, then the fallback strategy can be enforced by setting the header `SWARM-CRS-STRATEGY=fallback` or completely disabled by setting `SWARM-CRS-STRATEGY=disabled`. The number of parities for an erasure coded batch is taken from the enclosing manifest's `crs` attribute but can be overridden with the header `SWARM-CRS-PARITIES`.

Missing chunks

If a chunk is found missing, we can fall back on the missing chunk notification protocol (see 5.2). Reference to the root of the data structure representing the set of recovery targets is found at the latest update of the recovery feed.

When a chunk request times out, the client can start creating the recovery message using the set of pinner hosts (see 7.6.5). Once created it is sent to the host. If this times out, the next recovery chunk is tried using a different pinner host node. This is repeated until the recovery chunk arrives or all the pinners and insurers are exhausted.

Litigation

If the missing chunk notifications have not returned the chunk, but the file is insured, as a last resort, the client can litigate (see 3.3.4). For this the receipt is needed. To obtain the receipt, we first identify which insurer is responsible for the chunk in question (see 5.3). Using their public key, we can construct the feed addresses that point to the current receipt manifest root. With the chunk address appended to this reference, the client can retrieve the receipt. Once the receipt is obtained, the client is able to start the litigation by submitting a challenge to the *swear* contract. Since the data structure holding the receipts is implicitly insured by the insurer, if any chunk needed to obtain the receipt went missing, the challenge can be submitted as the reference to that. Instead of a receipt signed, the challenge will contain the inclusion proof for the segment containing the reference in the root hash found in the update together with the identifier and the signature in the feed chunk.

6.3 COMMUNICATION

Somewhat surprisingly, Swarm's network layer can serve as an efficient communication platform with exceptionally strong privacy properties. This chapter is an attempt to define a comprehensive set of primitives that serve as building blocks for a base layer communication infrastructure covering the full range of communication modalities including real time anonymous chat, sending and receiving messages from previously not connected, potentially anonymous senders, mailboxing for asynchronous delivery, long term notifications and publish/subscribe interfaces.

Swarm core offers the lowest level entry-points to functionality related to communication.

- pss offers an API to send and receive trojan chunks
- bzz offers a way to upload single owner chunk specifically. Single-owner chunk retrieval requires nothing beyond what is provided by the storage APIs

From a user experience perspective, these primitive interfaces for pss and feeds are too low-level. It is crucial however, that any higher level solution can safely be built on top of the core Swarm infrastructure ideally as packages in the apiary.

6.3.1 *Feeds*

6.3.2 *Pss*

Trojan message sending and receiving is sufficiently different functionality than storage and therefore deserves its own URL-scheme, `pss`. The pss scheme provides an API for sending messages. When sending a `POST` request, the URL is interpreted as referencing

the X3DH pre-key bundle feed update chunk. This contains the public key needed to encrypt as well as the destination targets, one of which the trojan message address should be mined to match (see 4.4.1).

Destination targets are represented as the buzz serialisation of the target prefixes as a file. The URL path point to this file or is a top-level domain. Topic is given as a query parameter and the message as the request body.

Receiving messages is supported only by registering topic handlers internally. In an API context this means push notifications via web-sockets.

6.4 SWARM AS BACKEND FOR WEB3

6.4.1 *Primitives*

6.4.2 *Scripting*

6.4.3 *Built-in composite APIs*

6.4.4 *Standard library*

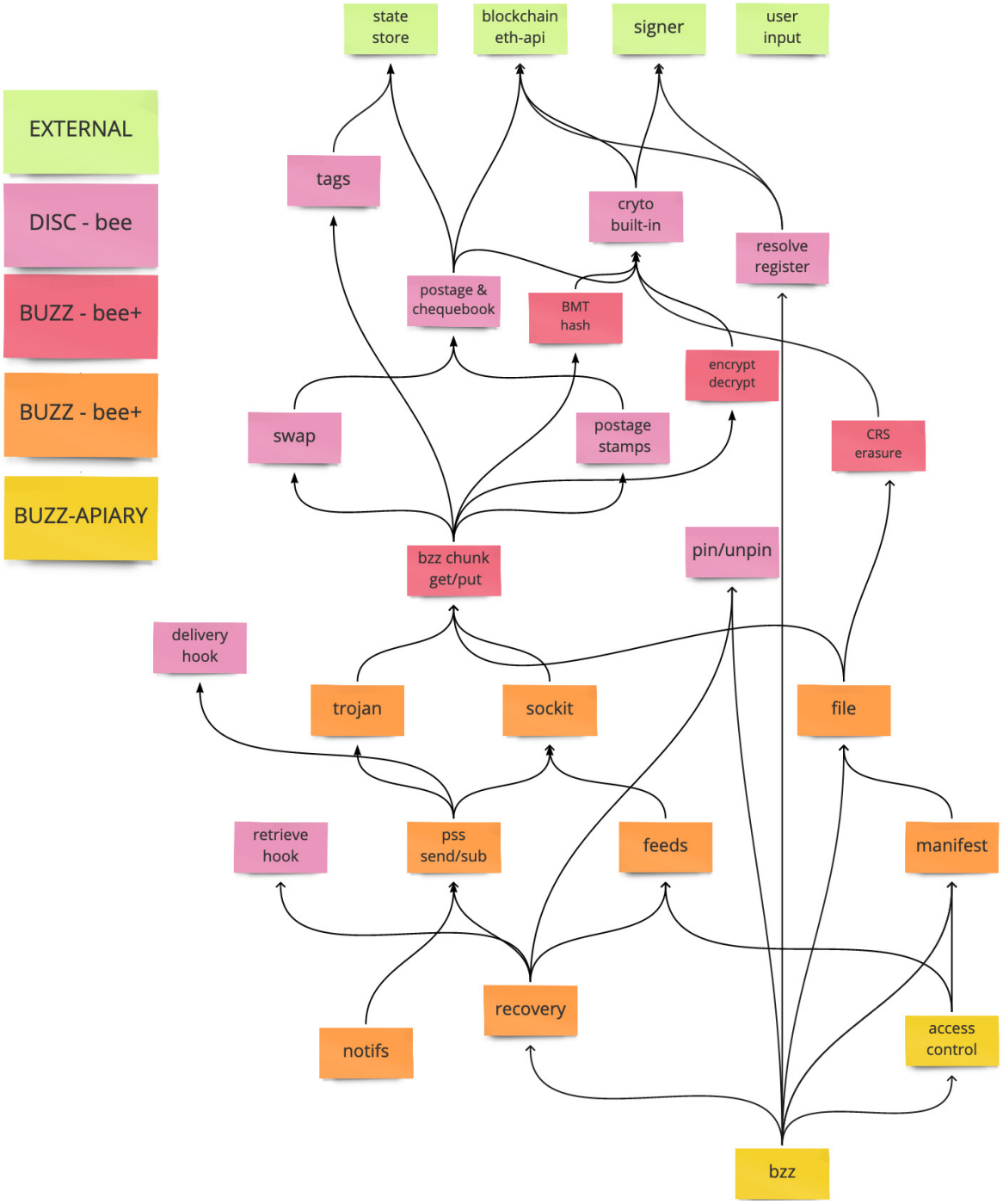


Figure 71: buzz apiary.

Part III

SPECIFICATIONS

The Swarm specification is presented in four chapters. Chapter 7 enlists all the conventions relating data types, formats and algorithms using the `buzz` language as pseudo-code. Chapter 8 (Protocols) specifies the wire protocols: message formats, serialisation and encapsulation are hard requirements that are crucial for cross client compatibility. This chapter benefits from `protobuf`,[1] which is a generic language-neutral, platform-neutral, extensible mechanism for serializing structured data. The chapter on Strategies (9) is meant to present recommended incentive-aligned behaviour. APIs (10) gives a formal specification of the high level interfaces of Swarm. Different client implementations are to be tested against a standard suite of API tests. The API specifications benefit from `OpenAPI`,[2] which is an API description format for REST APIs.

1 https://developers.google.com/protocol-buffers
2 https://swagger.io/

LIST OF DEFINITIONS

7

DATA TYPES AND ALGORITHMS

7.1 BUILT-IN PRIMITIVES

7.1.1 *Crypto*

This section describes the crypto primitives used throughout the specification. They are exposed as buzz built-in functions. The modules are hashing, random number generation, key derivation, symmetric and asymmetric encryption (ECIES), mining (i.e., finding a nonce), elliptic curve key generation, digital signature (ECDSA), Diffie–Hellman shared secret (ECDH) and Cauchy-Reed-Solomon (CRS) erasure coding.

Some of the built-in crypto primitives (notably, sha3 hash, and ECDSA ecrecover) are replicating crypto functionality of the Ethereum VM. These are defined here with the help of ethereum api calls to a smart contract. This smart contract just implements the primitives of "buzz" and only has read methods.

Hashing

The base hash function implements Keccak256 hash as used in Ethereum.

Definition 1 Hashing**.**

```
// /crypto

define function hash @input []byte
    ?and/with @suff
    return segment
as
    ethereum/call "sha3" with @input append= @suff
         on context contracts "buzz"
```

Random number generation

Definition 2 Random number generation**.**

```
// /crypto

define function random type
    return [@type size]byte
```

Scrypt key derivation

The crypto key derivation function implements `scrypt` [Percival, 2009].

Definition 3 Scrypt key derivation.

```
// /crypto

define type salt as [segment size]byte

define type key as [segment size]byte

// params for scrypt key derivation function
// scrypt.key(password, salt, n, r, p, 32) to generate key

define type kdf
    n int // 262144
    r int // 8
    p int // 1

define function scrypt from @password
    with   salt
    using  kdf
    return key
```

Mining helper

This module provides a very simple helper function that finds a nonce that when given as the single argument to a mining function returns true.

Definition 4 Mining a nonce.

```
// /crypto

define type nonce as [segment size]byte

define function mine @f function of nonce return bool
as
```

```
    @nonce = random key                                    7
    return @nonce if call @f @nonce                        8
    self @f                                                9
```

Symmetric encryption

Symmetric encryption uses a modified blockcipher with 32 byte blocksize in counter mode. The segment keys are generated by hashing the chunk-specific encryption key with the counter and hash that again. This second step is required so that a segment can be selectively disclosed in a 3rd party provable way yet without compromising the security of the rest of the chunk.

The module provides input length preserving blockcipher encryption.

Definition 5 Blockcipher**.**

```
// /crypto                                                                  1
                                                                            2
// two-way (en/de)crypt function for segment                                3
define function crypt.segment segment                                      4
    with key                                                                5
    at @i uint8                                                             6
as                                                                          7
    hash @key and @i          // counter mode                              8
        hash                  // extra hashing                             9
        to @segment length    // chop if needed                            10
        xor @segment          // xor key with segment                      11
                                                                            12
// two-way (en/de)crypt function for arbitrary length                       13
define function crypt @input []byte                                        14
    with key                                                                15
    return [@input length]byte                                              16
as                                                                          17
    @segments = @input each segment size    // iterate segments of input   18
        go crypt.segment at @i++ with @key  // concurrent crypt on          19
           segments
    return wait for @segments               // wait for results            20
        join                                // join (en/de)crypted         21
           segments
```

Elliptic curve keys

Public key cryptography is the same as in Ethereum, it uses the secp256k1 elliptic curve.

Definition 6 Elliptic curve key generation**.**

```
// /crypto
define type pubkey  as [64]byte
define type keypair
    privkey [32]byte
    pubkey

define type address as [20]byte

define function address pubkey
    return address
as
    hash pubkey
        from 12

define function generate
p    ?using entropy
as
    @entropy = random segment if no @entropy
    http/get "signer/generate?entropy=" append @entropy
        as keypair
```

Asymmetric encryption

Asymmetric encryption implements ECIES based on the secp256k1 elliptic curve.

Definition 7 Asymmetric encryption**.**

```
// /crypto

define function encrypt @input []byte
    for pubkey
    return [@input length]byte

define function decrypt @input []byte
    with keypair
    return [@input length]byte
```

Signature

Crypto's built-in signature module implements secp2156k1 elliptic curve based ECDSA. The actual signing happens in the external signer running as a separate process (possibly within the secure enclave). As customary in Ethereum, the signature is represented and serialised using the r/s/v format,

Definition 8 Signature.

```
// /crypto

define type signature
    r segment
    s segment
    v uint8
    signer private keypair


define type doc
    preamble []byte
    context  []byte
    asset    segment

define function sign @input []byte
    by keypair
    return signature
as
    @doc = doc{ "swarm signature", context caller, @input }
    @sig = http/get "signer/sign?text=" append @doc
        append "&account=" append @keypair pubkey address
            as signature
    @sig signer = @keypair
    @sig

define function recover signature
    with @input []byte
    from @caller []byte
    return pubkey
as
    @doc =  doc{ "swarm signature", @caller, @input } as bytes
    ethereum/call "ecrecover" with
        on context contracts "buzz"
            as pubkey
```

Diffie-Hellmann shared secret

The shared secret module implements elliptic curve based Diffie–Helmann shared secret (ECDH) using the usual secp256k1 elliptic curve. The actual DH comes from the external signer which is then hashed together with a salt.

Definition 9 Shared secret.

```
// /crypto

define function shared.secret between keypair
    and pubkey
    using salt
    return [segment size]byte
as
    http/get "signer/dh?pubkey=" append @pubkey append "&account="
       @keypair address
        hash with @salt
```

Erasure coding

Erasure coding interface provides wrappers `extend/repair` for the encoder/decoder that work directly on a list of chunks.[1]

Assuming n out of m coding. `extend` takes a list of n data chunks and an argument for the number of required parities. It returns the parity chunks only. `repair` takes a list of m chunks (extended with *all* parities) and an argument for the number of parities $p = m - n$, that designate the last p chunks as parity chunks. It returns the list of n repaired data chunks only. The encoder does not know which parts are invalid, so missing or invalid chunks should be set to `nil` in the argument to repair. If parity chunks are needed to be repaired, you call `repair @chunks with @parities; extend with @parities`

Definition 10 CRS erasure code interface definition.

```
// /crypto/crs

define function extend @chunks []chunk
    with @parities uint
    return [@parities]chunk

define function repair @chunks []chunk
    with @parities uint
   return [@chunks length - @parities]chunk
```

1 Cauchy-Reed-Solomon erasure codes based on https://github.com/klauspost/reedsolomon.

Definition 11 CRS erasure coding parameters.

```
// /crypto/crs
define strategy as "race"|"fallback"|"disabled"

define type params
    parities uint
    strategy
```

7.1.2 *State store*

Definition 12 State store.

```
// /statestore

define type key []byte
define type db  []byte
define type value []byte

define function create db

define function destroy db

define function put value
    to db
    on key

define function get key
    from db
    return value
```

7.1.3 *Local context and configuration*

Definition 13 Context.

```
// /context

define type contract as "buzz"|"chequebook"|"postage"|""

define type context
```

7.2 BZZ ADDRESS

7.2.1 *Overlay address*

Swarm's overlay network uses 32-byte addresses. In order to help uniform utilisation of the address space, these addresses must be derived using a hash function. A Swarm node must be associated a Swarm base account or bzz account, which is an Ethereum account that the node (operator) must possess the private key for. The node's overlay address is derived the public key of this account.

Definition 14 Swarm overlay address of node A**.**

$$overlayAddress(A) \stackrel{\text{def}}{=} Hash(ethAddress|bzzNetworkID) \tag{1}$$

where

- *Hash* is the 256-bit Keccak SHA3 hash function
- *ethAddress* - the ethereum address (bytes, not hex) derived from the node's base account public key: $account \stackrel{\text{def}}{=} PubKey(K_A^{bzz})[12:32]$, where
  - *PubKey* is the *uncompressed* form of the public key of a keypair *including* its 04 (uncompressed) prefix.
  - K_A^{bzz} refers to the node's bzz account key pair
- *bzzNetworkID* is the bzz network id of the swarm network serialised as a little-endian binary *uint64*.

In a way, deriving the node address from a public key means the overlay address space is not freely available: to occupy an address you must possess the private key of the address which other nodes need to verify. Such authentication is easy using a digital signature of a shared consensual piece of text, see 7.2.

7.2.2 *Underlay address*

To enable peers to locate the a node on the network, the overlay address is paired with an underlay address. The underlay address is a string representation of the node's network location on the underlying transport layer. It is used by nodes to dial other nodes to establish keep-alive peer to peer connections.

Definition 15 Swarm underlay multiaddress**.**

7.2.3 *BZZ address*

Bzz address is functionally the pairing of overlay and underlay addresses. In order to ensure that an overlay address is derived from an account the node possesses as well as verifiably attest to an underlay address a node can be called on, bzz addresses are communicated in the following transfer format:

Definition 16 Swarm bzz address transfer format**.**

```
// ID: /swarm/handshake/1.0.0/bzzaddress

syntax = "proto3";

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

Here the signature is attesting to the association of an overlay and an underlay address for a network.

Definition 17 Signed underlay address of node A**.**

$$signedUnderlay(A) \stackrel{\text{def}}{=} Sign(underlay|overlay|bzzNetworkID) \tag{2}$$

of the underlay with overlay and bzz network ID appended as plaintext and hashes the resulting public key together with the bzz network ID.

Definition 18 Node addresses: overlay, underlay, bzz address**.**

```
// /bzz

define type overlay as [segment size]byte
define type underlay []byte

define function overlay.address of pubkey
    within @network uint64
as
    hash @pubkey address and @network
        as overlay

define function valid bzz.address
    within @network uint64
```

```
as                                                                    14
    assert @bzz.address overlay == overlay.address of crypto/recover  15
       from @bzz.address signature with @bzz.address @underlay
        within @network                                               16
                                                                      17
define function bzz.address of overlay                               18
    from underlay                                                     19
    by @account ethereum/address                                      20
as                                                                    21
    @sig = crypto/sign @underlay by @account                          22
    bzz.address{ @overlay, @underlay, @sig }                          23
```

In order to get the overlay address from the transfer format peer info, one recovers from signature the peer's base account public key using the plaintext that is constructed as per 17. From the public key, the overlay can be calculated as in 14. The overlay address thus obtained needs to be checked against the one provided in the handshake.

Signing of the underlay enables preflight authentication of the underlay of a trusted but not connected node.

Since underlays are meant to be volatile, we can assume and in fact expect multiple underlays signed by the same node. However, these are meant to be temporally ordered. So one with a newer timestamp invalidates the older one.

In order to make sure that the node connected through that underlay does indeed operate the overlay address, its authentication must be obtained through the peer connection that was initiated by dialing the underlay. The This protects against malicious impersonation of a trusted overlay potentially.

7.3 CHUNKS, ENCRYPTION AND ADDRESSING

7.3.1 *Content addressed chunks*

First let us define some basic types, such as *payload, span, segment*. These fixed length byte slices enables verbose expression of fundamental units like `segment size` or `payload size`.

Definition 19 segment, payload, span, branches**.**

```
// /chunk                                                             1
                                                                      2
define type segment as [32]byte    // unit for type definitions      3
define type payload as [:4096]byte // variable length max 4Kilobyte  4
define type span as uint64         // little endian binary marshalled 5
                                                                      6
define function branches                                              7
```

```
    as payload size / segment size                                    8
```

Now let's turn to the definition of *address*, *key* and *reference*:

Definition 20 Chunk reference**.**

```
// /chunk                                                             1
                                                                      2
define type address as [segment size]byte                            3
                                                                      4
define type reference                                                 5
    address                    // result of bmt hash                 6
    key if context.encryption // decryption key optional (context    7
       dependent)
```

Now, define chunk as a object with span and payload.

Definition 21 Content addressed chunk**.**

```
// /chunk                                                             1
                                                                      2
define type chunk                                                     3
    span      // length of data span subsumed under node             4
    payload   // max 4096 bytes                                       5
                                                                      6
define function address of chunk                                      7
as                                                                    8
    @chunk payload bmt/hash with @chunk span                          9
                                                                      10
define function create from payload                                   11
    ?over span                                                        12
as                                                                    13
    @span = @payload length if no @span                               14
    @chunk = chunk{ @span, @payload }                                 15
    return @chunk if no context encryption                            16
    @key = encryption.key for @chunk                                  17
    @chunk encrypt with @key                                          18
```

Where length is the content of the length field and reference size is the sum of size of the referencing hash value and that of the decryption key, which is currently 64, as we use 256-bit hashes and 256-bit keys.

In order to remove the padding after decryption before returning the plaintext chunk.

Definition 22 Span to payload length.

```
// /chunk

define function payload.length of span
as
    while @span >= 4096
        @span = @span + 4095
           / 4096
           * reference size
    return @span
```

Finally, we can define the public API of chunks for retrieval and storage.

Definition 23 Chunk API retrieval.

```
// /chunk

define function retrieve reference
has api GET on "chunk/<reference>"
as
    retrieve @reference address as chunk
        (decrypt with @reference key if @reference key)
```

Definition 24 Chunk API: storage.

```
// /chunk


define function store payload
    ?over span
has api POST on "chunk/(?span=<span>)"
    from payload as body
as
    @chunk = create from @payload over @span
    reference{ @chunk address, @chunk key }
```

7.3.2 *Single owner chunk*

Single owner chunks are the second type of chunk in swarm (see 2.2.3). They constitute the basis of swarm feeds.

Definition 25 Single owner chunks.

```
// /soc

// data structure for single owner chunk
define type soc
    id         [segment size]byte // id 'within' owner namespace
    signature  crypto/signature   // owner attests to <content, id>
    chunk                         // content: embeds a content chunk

// constructor for single owner chunks
define function create from chunk
    by @owner crypto/keypair
    on @id [segment size]byte
 as
    @sig = crypto/sign @id and @chunk address by @owner
    soc{ @id, @sig, @chunk }

define function address soc
as
    hash @soc id and @soc signature signer address
```

Definition 26 Single owner chunk API: retrieval.

```
// /soc

define function retrieve @id [segment size]byte
    by @owner ethereum/address
    ?with key
has api GET on "soc/<owner>/<id>(?key=<key>)"
as
    retrieve hash @id and @owner
        as soc
            chunk (decrypt with @key if @key)
```

Definition 27 Single owner chunk API: storage.

```
// /soc

define function store payload
    on @id [segment size]byte
    by @owner ethereum/address
    ?over span
has api POST on "soc/<owner>/<id>?span=<span>&encrypt=<encrypt>"
```

```
    from <payload> as body                                          8
as                                                                  9
    @span = @payload length if no @span                             10
    @chunk = chunk{ @span, @payload }                               11
    if context encryption then                                      12
        @key = encryption.key for @chunk                            13
        @chunk encrypt= with @key                                   14
    @soc = create from @chunk on @id by private key of @owner       15
    reference{ @soc store address, @key }                           16
```

7.3.3 *Binary Merkle Tree Hash*

The hashing method used to obtain the address of the default content addressed chunk is called the binary Merkle tree hash, or BMT hash for short.

Calculating the BMT hash

The base segments of the binary tree are subsequences of the chunk content data. The size of segments is 32 bytes, which is the digest size of the *base hash* used to construct the tree. Given the Swarm hash tree used to represent files (see 7.4.1) assumes that intermediate chunks package references to other chunks.

Obtaining the BMT hash of a sequence involves the following steps:

1. *padding* - If the content is shorter than the maximum chunk Size (4096 bytes, 2.2.2), it is padded with zeros up to chunk size. Note that this zero padding is only for hashing and does not impact chunk data sizes.

2. *chunk data layer* - Calculate the base hash of *pairs of segments* in the padded chunk, i.e., segment size $(2 * 32)$ units of data and concatenate the results.

3. *building the tree* - Repeat previous step on the result until the result is just one section.

4. *calculate span* - Calculate the span of the data, i.e., the size of the data that is subsumed under the chunk represented by the unpadded data as a 64-bit little-endian integer value (see 7.4.1).

5. *integrity protection* - Prepend the span to the root hash of the binary tree and calculate the *base hash* of the data.

Definition 28 BMT hash**.**

```
// /bmt

define function hash payload
    with span
as
    @padded = @payload as [:chunk size]byte    // use zero padding
    // for BMT hashing only
    hash @span and root of @padded over chunk size

define function root of @section []byte
    over @len uint
as
    return hash @section        // data level
        if @len == 2 * segment size
    @len /= 2                  // recursive call
    @children = @section each @len go self over @len
    wait for @children
        join hash
```

Inclusion proofs

Having the segments align with the hashes packaged in these chunks one can extend the notion of inclusion proofs to files. The BMT hash enables compact 3rd party verifiable segment inclusion proofs.

7.3.4 *Encryption*

Symmetric encryption in Swarm is using a slightly modified version blockcipher in counter mode.

The encryption seed for the chunk is derived from the master seed if given, otherwise just generated randomly.

The reference to a single chunk (and the whole content) is the concatenation of the hash of encrypted data and the decryption key (see 20). This means the encrypted Swarm reference (64 bytes) will be longer than the unencrypted one (32 bytes). When a node syncs encrypted chunks, it does not share the full references (or the decryption keys) with the other nodes in any way. Therefore, other nodes will be unable to access the original data, or in fact, even to detect whether a chunk is encrypted.

Definition 29 Chunk encryption/decryption API.

```
// generate key for a chunk
define encryption.key for chunk
```

```
    ?with @seed [segment.size]byte
as
    return crypto/random key if no @seed // generate new
    hash @seed and @chunk address

define function encrypt chunk
    with key
as
    @segments = @chunk data pad to chunk size
        each segment size
            go crypt at @i++ with @key
    @span = chunk span crypt at branches with @seed
    @payload = wait for @segments
        join
    chunk{ @span, @payload }


define function decrypt chunk
    with key
as
    @span  = @chunk span
        crypt at branches with @key
    @segments = chunk data to @span payload.length
        each segment size
            go crypt at @i++ with @key
    @payload = wait for @segments
        join
    chunk{ @span, @payload }
```

Encrypted Swarm chunks are not different from plaintext chunks and therefore no change is needed on the P2P protocol level to accommodate them. The proposed encryption scheme is end-to-end, meaning that encryption and decryption is done on endpoints, i.e., where the http proxy layer runs. This has an important consequence that public gateways cannot be used for encrypted content. On the other hand, the apiary modular design allows for client side encryption on top of external APIs while proxying all other calls via the gateway.

7.4.1 *Files and the Swarm hash*

This table gives an overview of data sizes a chunk span represents, depending on the level of recursion.

	span					
	unencrypted			encrypted		
level	chunks	log_2 of bytes	standard	chunks	log_2 of bytes	standard
0	1	12	4KB	1	12	4KB
1	128	19	512KB	64	18	256KB
2	16,384	26	67MB	4,096	24	16MB
3	2,097,152	33	8.5GB	262,144	30	1.07GB
4	268.44M	40	1.1TB	16.78M	36	68.7GB
5	34,359.74M	47	140TB	1,073.74M	42	4.4TB

Table 3: Size of chunk spans

Calculating the Swarm Hash

Client-side custom redundancy is achieved by CRS erasure coding (see 5.1.2 and 6.1.4); using it neccessitates some CRS parameters.

Definition 30 File API: upload/storage; swarm hash**.**

```
// /file

define function encode @levels []chunk stream
    for @level uint
as
    @chunks = @levels at @level     // read chunk stream
    @crs = context crs              //
    @m = branches (- @crs parities if @crs)

    @parent = read @m from @chunks  // read up to m chunks from stream
       blocking
        (append crs/extend with @crs parities if @crs)
            each chunk/store        // package children reference
                join as chunk

    if @levels length == @level+1 then
        @levels append= stream{}
```

```
        go self @levels for @level+1
    write @parent to @levels at @level+1
    if no @chunks then
        close @levels at @level + 1
    else
        self @chunks for @level

define function split @data byte stream
as
    @level = chunk stream{}
    go @data each chunk size as chunk
        write to @chunks
    @level

define function upload byte stream as @data
has api POST  on "file/" from @data as body
as
    @levels append= @data split     //
    go encode @levels for 0
    @top = @levels each wait for  // wait for all levels to close
    return @top at 0              // return root hash as address
```

Definition 31 File API: download/retrieval.

```
// /file

define function copy to @reader stream of byte{}
    from @chunks stream of []chunk
    using @buffers stream of [@buffer.size]stream of [branches]chunk
as
    @chunks = read @buffers if no @chunks
    @chunk = read @chunks
    if no @chunk
        write @chunks to @buffer
        self @reader using @buffers
    write @chunk data to @reader

define function download reference
    ?using @buffer.size uint64
has api GET  on "file/<reference>"
as
```

```
    @reader = stream of byte{}
    @buffers = stream of [@buffer.size]stream of [branches]chunk
    chunk/retrieve @reference            // root chunk retrieval
        go decode into @buffer down to 1 // traverse
    copy into @reader from @buffers

define function decode chunk
    into @response chunk stream
    ?down to @limit uint8
as

    @crs = context crs
    @all = @m = branches
    if @crs then
        @m -= @crs parities
        if @crs strategy is not "race" then
            @all = @m

    @chunks = @chunk segments up to @all
        each go as reference retrieve
    wait for @m in @chunks

    if @crs then
        cancel @chunks
        @chunks = @chunks crs/repair with @crs parities

    if @chunk span < chunk size exp @limit + 1 then
        @chunks each  into @reader


    @chunks each go self down to @limit
```

Definition 32 File info.

```
// /file

define type info
    mode        int64
    size        int64
    modified    time
```

7.4.2 *Manifests*

Manifests represent a mapping of strings to references (see 4.1.2). The primary purpose is to implement document collections (websites) on swarm and enable URL-based addressing of content. This section defines the data structures relevant for manifests as well as the algorithms for lookup and update which implement the manifest API (see 6.2.2 and 10.2.4).

A manifest entry can be conceived of as metadata about a file pointed to and retrievable by its reference (see 7.4.1). The metadata is quite diverse, ranging from information needed for access control, file information similar to one given on file systems, information needed for erasure coding (see 5.1.2, 6.1.1 and 7.1.1), information for browser, i.e., response headers such as content type (MIME info) and most importantly the reference to the file. Using manifests as simple key-value store is exemplified by access control (see 4.2, 6.2.3 for discussion as well as 7.5 and 10.2.9 for the specification).

Definition 33 Manifest entry**.**

```
// /manifest

// manifest entry encodes attributes
define type entry
    file/info             // FS file/dir info
    access/params         // access control params
    crs/params            // erasure coding - CRS params
    reference             // reference
    headers               // http response headers

define type headers
    content.type  [segment size]byte
```

Definition 34 Manifest data structure**.**

```
// /manifest

define type node
    entry  *entry         // reference to chunk serialised as entry
    forks  [<<256]fork    // sparse array of max 256 fork

// fork encodes a branch
define type fork
    prefix segment   // compaction
    node   *node     // reference to chunk serialised as node
```

Definition 35 Manifest API: path lookup.

```
// /manifest

define function lookup @path []byte
    in *node
has api GET on "/manifest/<@node:reference>/<@path>"
as
    context access = @node entry access/params
    // manifest is  a compacted trie
    @fork = @node forks at head @path
    // if @path empty, the  paths matched return the entry
    if no @path then
        return @node entry

    if @fork prefix is prefix of @path then // including ==
      return self @path from @fork prefix length
          in @fork node
   fail with "not found"
```

Definition 36 Manifest API: update.

```
// /manifest

define function add *entry
    to *node
    on @path []byte
has api PUT on "/manifest/<@node>"

as
    // if called on nil call on zero value
    @node = node{} if no @node

    // if empty path then change entry field of node
    if no @path then
        @node entry = @entry
        return store @node

    // lookup the fork based on the first byte of path
    @fork = @node forks at head @path
    // if no fork yet, add the singleton node
    if no @fork then
```

```
        @node forks at head @path =
            fork{@path, store node{@entry}}
        return store @node

    @common = prefix of @path and @fork prefix // common cannot be empty
    @rest = @fork prefix from @common length
    @newnode = node{}
    @newnode forks at head @rest = fork{@rest, @fork node}
    @midnode = self @entry to @newnode on @path from @common length
    @node forks at head @path = fork{ @common, @midnode }
    @node store
```

Definition 37 Manifest API: remove.

```
// /manifest

define function remove @path []byte
    from *node
has api DELETE on "/manifest/<@node>/<@path>"
as
    // if called on nil call on zero value
    return node{} if no @node

    // if empty path then change entry field of node
    if no @path then
        return nil if @node forks length == 0
        @node entry = nil //entry exists
        return store @node

    // lookup the fork based on the first byte of path
    @fork = @node forks at head @path
    // if no fork yet, add the singleton node
    return @node if no @fork

    @common = prefix of @path and @fork prefix // common cannot be empty
    return @node if @common and @fork prefix have different length
                // path not found

    @rest = @fork prefix from @common length
    @newnode =  self @rest from @fork node
    if no @newnode then           // deleted item was terminal node,
       delete fork
```

```
        @node forks at head @res = nil
    else if @newnode forks length == 1 then // compact non-forking nodes
        @singleton = @newnode forks first
        @newprefix = @common append @singleton prefix
        @node forks at head @path =
            fork{ @newprefix, @singleton node }
    else
        @node fork at head @path node = @newnode

    @node store
```

Definition 38 Manifest API: merge.

```
// /manifest

define function merge @new *node
    to  @old *node
has api POST on "/manifest/<@old:reference>/<@new:reference>"
as
    // if called on nil call on zero value
    return @new if no @old
    return @old if no @new
    @node = node{ @new or @old }
    @new forks pos or @old forks pos
        each bit @pos go
            @fork = merge.fork @new forks at @pos
                to @old forks at @pos
            @node forks at @fork prefix head = @fork
    @node store

define function merge.fork @new fork
    to @old fork
as
    @common = prefix of @new prefix and @old prefix
    @restnew = @new prefix from @common length
    @restold = @old prefix from @common length
    if no @restnew and no @restold then
        return fork{@common, merge.node @new reference to @old reference
           }
    @node = add @new reference to nil on @restnew
        add to @old reference @restold
    fork{ @common, @node }
```

7.4.3 *Resolver*

Definition 39 Resolver.

```
// /resolver

define type resolver
    api url
    address ethereum/address
    tlds []string

define function resolve @host []byte through @resolver
as
    @tld = @host split on '.'' last    @resolver = @resolvers any tlds
       any ==  @tld

    ethereum/call "getContentHash" of @host using @resolver api at
       @resolver address
        as address
```

7.4.4 *Pinning*

Definition 40 Pinning.

```
// /pin

define type pin
    reference
    chunks uint64

define function list
    return []pin
has api GET "/pin/"

define function view reference
    return uint64
has api GET "/pin/<reference>"

define function pin reference
    return uint64
```

```
has api PUT "/pin/<reference>"

define function pin reference
    return uint64
has api DELETE "/pin/<reference>"
```

7.4.5 *Tags*

Definition 41 Tags.

```
// /tag

define type tag
    id     [segment size]byte
    reference     // the current root
    complete bool // if local upload finished
    total  uint64 // number of chunks expected
    split  uint64 // number of chunks split
    stored uint64 // number of chunks stored locally
    seen   uint64 // number of chunks already in db
    sent   uint64 // number of chunks sent with push-sync
    synced uint64 // number of chunks synced


define function list
    return []tag
has api GET "/tag/"

define function view reference
    return uint64
has api GET "/tag/<reference>"

define function add reference
    return uint64
has api POST "/tag/"

define function remove reference
    return uint64
has api DELETE "/tag/<reference>"
```

7.4.6 *Storage*

Definition 42 Public storage API.

```
// /bzz

define function upload @data stream of byte
has api POST on "bzz:/<host>/<path>" from @data as body
as
    @root = resolver/resolve @host
    @manifest = access/unlock @root
    @entry = file/upload @data
    @reference =  chunk/store @entry as []byte
    manifest/add @reference to @manifest on @path


define function download @path []byte     from @host []byte
has api POST on "bzz:/<host>/<path>" from @data as body
as
    @root = resolver/resolve @host
    @manifest = access/unlock @root
    @entry = manifest/lookup @path in @manifest
    file/download @entry reference
```

7.5 ACCESS CONTROL

Definition 43 Access control: auth, hint, parameters, root access manifest.

```
// /access

define type auth as "pass"|"pk"
define type hint as [segment size]byte

// access control parameters
define type params
  auth                        // serialises uint8
  publisher crypto/pubkey     // 65 byte
  salt                        // salt for scrypt/dh
  hint                        // hint to link identity
  act       *node             // reference to act manifest root
  kdf                         // params for scrypt
}
```

```
// root access manifest
define type root
    params
    reference
```

Definition 44 Session key and auth with credentials.

```
// /access

define function session.key.pass from hint
    with  salt
    using kdf
as
    crypto/scrypt from input/password using @hint
        with @salt using @kdf

define function session.key.pk from hint
    with crypto/pubkey
    using salt
as
    crypto/shared.secret between
        input/select key by @hint
            and @pubkey
        hash with @salt

define function session.key
    using params
as
    if @params auth == "pass" then
        return session.key.pass from  @params hint
            with @params salt using @params kdf

    session.key.pk from @params hint
        with @params publisher using @params salt
```

Definition 45 Access key.

```
// /access
define function access.key
    using params
as
```

```
    @key = session.key using @params
    return @key if no @params act
    act.lookup @key in @params act

define function act.lookup key
    in @act *node
as
        manifest/lookup hash @key and 0
            in @act
                xor hash @key and 1
```

Definition 46 Access control API: lock/unlock.

```
// /access

// control
define function lock reference
    using params
has api POST on "/access/<address>"
    with @params as body
as
    @key = hash @reference address and @key
    @encrypted = @reference as bytes
        crypto/crypt with @key
    root{ @params, @encrypted }
        store


define function unlock address
has api GET on "access/<address>"
as
    @root = retrieve @address
        try as root otherwise return @address
    @key = access.key using @root params
    @root encrypted crypto/crypt with @key
        as *node
```

Definition 47 ACT manipulation API: add/remove.

```
// /access

define type act as manifest/node
```

```
define function add @keys []crypto/pubkey
    to *root
has api PUT on "/access/<root>/"
    with @keys as body
as
    // get params from the root access structure
    @params = retrieve @root as root params
    @access.key = access.key using @params
    @keys each @key
        @session.key = session.key using @params
        manifest/add @access.key xor hash @session.key with 1
            to @act on hash @session.key with 0


define function remove @keys []crypto/pubkey
    from *root
has api DELETE on "/access/<root>"
    with @keys as body
as
    // get params from the root access structure
    @params = retrieve @root as root params
    @keys each @key
        @session.key = session.key using @params
        manifest/remove hash @session.key with 0
            from @params act
```

7.6 PSS

7.6.1 *PSS message*

7.6.2 *Direct pss message with trojan chunk*

Pss has two fundamental types, a message and a trojan chunk structure which wraps the encrypted serialised message and contains a nonce that is mined to make the resulting chunk's content address (BMT hash) to match the targets.

Definition 48 Basic types: topic, targets, recipient, message and trojan.

```
// /pss
```

```
define type topic        as [segment size]byte       // obfuscated topic matcher
define type targets      as [][]byte       // overlay prefixes
define type recipient    as crypto/pubkey

// pss message
define type message
    seal    segment
    payload [!:4030]byte    // varlength padded to 4030B

// trojan chunk
define type trojan
    nonce   segment           // the nonce to mine
    message [4064]byte        // encrypted msg
```

The message is encoded in a way that allows integrity checking and at the same time obfuscates the topic. The operation to package the payload with a topic is called *sealing*

Definition 49 Sealing/unsealing the message.

```
// /pss

define function seal @payload []byte
    with topic
as
    @seal = hash @payload and @topic // obfuscate topic
        xor @topic
    return message{ @seal, @payload }

define function unseal message
    with topic
as
    @seal = hash @message payload and @topic
    if @topic == @seal xor @message seal then // check
        return @payload
    return nil
```

Functions `wrap/unwrap` transform between message and trojan chunk. `wrap` takes an optional recipient public key to asymmetrically encrypt the message. The targets are a list of overlay address prefixes derived from overlay addresses of recipients, with

length specified to guarantee that a chunk matching it will end up with the recipient solely as a result if push-syncing.

Definition 50 Wrapping/unwrapping.

```
// /pss

define function wrap message
    for recipient
    to  targets
as
    @msg = @message
        (crypto/encrypt for @recipient if @recipient)

    @nonce = crypto/mine @n such that
        @targets any is prefix of
            trojan{@n, @msg} as chunk address
    trojan{@nonce, @msg} as chunk

define function unwrap chunk
    for recipient
as
    @chunk bytes
        (crypto/decrypt  for @recipient if @recipient)
            as message
```

When a chunk arrives at the node, `pss/deliver` is called as a hook by the storage component. First the message is unwrapped using the recipient private key and unsealed with all the topics API clients subscribed to. If the unsealing is successful, message integrity as well as topic matching is proven so the payload is written into the stream registered for the topic in question.

Definition 51 Incoming message handling.

```
// /pss

// mailbox is a handler type, expects payload
// sent sealed with the topic to be delivered via the stream
define type mailbox
    topic
    deliveries stream of []byte

define context mailboxes as []mailbox
```

```
define function deliver chunk
    @msg = @chunk unwrap for context recipient
    mailboxes each @mailbox
        @payload = unseal @msg  with @mailbox topic
        if @payload then
            write @msg payload
                to @mailbox deliveries
```

Definition 52 pss API: send.

```
// /pss

define function send @payload []byte
    about topic
    for recipient
    ?to    targets
has api POST on "/pss/<recipient>/<topic>(?targets=<targets>)"
    with @payload as body
as
    targets = lookup.targets for @recipient if no @targets
    context tag = tag/tag{}
    seal @payload with @topic        // seal with topic
        wrap for @recipient          // encrypt if given recipient
            to @targets              // mine nonce and returns trojan
               chunk
                store                // to be sent by push-sync
    return tag                       // tag to monitor status
```

Definition 53 pss API: receive.

```
// /pss

define function receive about topic
    on uint64 @channel
has api POST on "/pss/subscribe/<topic>(?on=<channel>)"
as
    @stream = open @channel
    context mailboxes append= mailbox{ @topic, @stream }

define function cancel topic
    on @channel uint64
has api DELETE on "/pss/subscribe/<topic>(?on=<channel>)"
```

```
as                                                          13
    context mailboxes any ch                                14
```

7.6.3 *Envelopes*

Definition 54 Envelope.

```
// /pss                                                     1
                                                            2
define type envelope                                        3
    id  [segment size]byte                                  4
    sig crypto/signature                                    5
    ps  postage/stamp                                       6
```

7.6.4 *Update notifications*

7.6.5 *Chunk recovery*

Definition 55 Targeted delivery request.

```
// /targeted.delivery                                       1
                                                            2
define type envelope                                        3
    id        segment                                       4
    signature crypto/signature                              5
                                                            6
                                                            7
define function wrap address                                8
    ?by @key crypto                                         9
    to @targets []target                                    10
as                                                          11
    @key = crypto/random  if no @key                        12
    @n = crypto/mine @n such that                           13
        @targets any is prefix of                           14
            hash @n and @key account                        15
    @sig = crypto/sign hash @n and @address                 16
        by @key                                             17
    envelope{ @n, @sig  }                                   18
```

Definition 56 Prod missing chunk notification with recovery request. c

```
// /recovery

define type request
    address
    envelope

define function request address
    with @response bool
    to @targets
as
    @request = request{ @address }
    if @response then
        @request envelope =
            targeted.delivery/wrap @address by @key to @targets
    pss/send @request bytes about "RECOVERY" to @targets
```

Definition 57 Recovery response**.**

7.7 POSTAGE STAMPS

7.7.1 *Stamps for upload*

Witness type

There can be different implementations of postage stamps that differ in the structure and semantics of the *proof of payment*. To allow for new cryptographic mechanisms to be used as they are developed, the `witnessType` argument indicates the type of the witness used.

Witness type 0 stands for ECDSA witness, which is an ECDSA signature on the byte slice resulting from the concatenation of 1) preamble constant 2) chunk hash 3) batch reference 4) valid until date.[2] This is the bare minimum that postage stamp contracts and clients must implement.[3]

2 The binary encoding of the ECDSA signature is 65 bytes resulting from the concatenation of the r (32 bytes), s (32 bytes) and v (1 byte) parameters of the signature, in this order. The signature is calculated on the secp256k1 elliptic curve, just like the signatures of Ethereum transactions.

3 The ECDSA witness is the simplest and cheapest solution both in terms of gas consumed by the stamp verification contract and in terms of computational resources used off chain. Also, it does not rely on cryptographic assumptions in addition to those on which Ethereum critically relies, therefore as long as Ethereum is considered cryptographically secure, no advance in cryptorgraphy can render this witness type insecure. This is the justification for this witness type to be the only mandatory witness type to be implemented.

Witness type 1 refers to the RSA witness, which is an RSA signature on the same 128 bytes as above. The binary encoding of the RSA signature is of variable length, and is an Solidity ABI encoded array of the RSA signature s.[4]

The RSA witness is specified so that blind stamping services can be implemented in a simple fashion, in order to mitigate the privacy issues arising from the ability to link chunks signed with the same private key. Even though blind ECDSA signatures also exist, their protocol requires more rounds of communication, making the implementation of such a service more complex, more error-prone and less performant.

The inclusion of the entire public key in each RSA witness rather than storing the public key in contract state and just referencing it from the witness is justified by reducing the gas costs of interactions with the contract as well as future-proofing the design in case contract state rent is introduced in Ethereum. These considerations are more important than the brevity of postage stamps, marginally reducing the bandwith costs of uploading and forwarding stamped content.

Note that cryptographic advances can render RSA witnesses insecure without rendering Ethereum insecure, therefore RSA witnesses can be phased out in future versions of the protocol, if the security of RSA signatures gets compromised. Note, furthermore, that such blind signing services are not entirely trustless, through the damage they can incur is bounded. Trustless blind stamping services based on ZK proofs are not feasible at this stage, as the current algorithms are not sufficiently performant for the purpose, but given the rapid advances in the field, the development of suitable algorithms can be expected in the future, in which case a corresponding witness type will have to be specified in a separate SWIP.

Contract Upgrades

In order to facilitate the upgrade of the contract either in case of a discovered vulnerability or some feature extension (such as adding new witness types), it is recommended that the part holding the funds with the database of payments and the part that verifies witnesses are in separate contracts so that a backwards-compatible upgrade can be performed with minimal disruption.

In order to avoid centralized control, it is also recommended that it is the witness-verifying contract that is referenced in client configuration so that client operators can independently decide for themselves when and whether to switch to a new contract, as they become available.

Nodes participating in the same postage system are configured to reference the same contract on the same blockchain. This contract must conform to the following interface:

Definition 58 Postage contract**.**

4 as defined in *PKCS #1*, https://tools.ietf.org/html/rfc8017 and the RSA public key parameters n (RSA modulus) and e (public exponent).

This accessor method returns `true` if the proof embodied by `witness` checks out for all other arguments within the claimed validity period, i.e. when `block.timestamp` (the output of `TIMESTAMP EVM` opcode) is between `beginValidity` (inclusive) and `endValidity` (exclusive). Outside of the validity period, the return value is `undefined`.

Definition 59 Postage stamp basic types: batchID, address, witness, stamp, validity**.**

```
// /postage

define type batchid as [segment size]byte
define type address as bzz/address
define type witness as crypo/signature

// postage stamp
define type stamp
    batchid
    address
    witness

define function valid stamp
as
    // check validity on blockchain
    ethereum call "valid" using context contracts "postage"
        with @stamp
```

7.7.2 *Postage subscriptions*

7.7.3 *Postage lottery race*

8

PROTOCOLS

8.1 INTRODUCTION

8.1.1 *Underlay network*

The `libp2p` networking stack provides all required properties for the swarm underlay network laid out in 2.1.1.

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

8.1.2 *Protocols and streams*

Communication between peers is organised in protocols as logical units under a unique name that may define one or more *streams*. `libp2p` provides streams as

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

Every stream defines:

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

Streams are identified by `libp2p` case-sensitive protocol IDs. The following convention is used construct stream identifiers:

```
/swarm/ProtocolName/ProtocolVersion/StreamName
```

1

- All stream IDs are prefixed with `/swarm`.
- `ProtocolName` is an arbitrary string that identifies the protocol.
- `ProtocolVersion` is a string in semver form that is used to specify compatibility between protocol implementations over time.
- `StreamName` is an arbitrary string that identifies a stream defined as part of the protocol.

8.1.3 *Data exchange sequences*

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

- *single message sending* - not waiting for the response by the peer if it is not needed before closing the stream.
- *multiple message sending* - a series of data that is sent to a peer without reading from it before closing the stream.
- *request/response* - requires a single response for a single request before closing the stream.
- *multiple requests/response cycles* - require a synchronous response after every request before closing the stream.
- *exact message sequence* - requires multiple message types over a single stream in an exact order (see the handshake protocol in 8.3).

Streams have predefined sequences that are kept as simple as possible for a single purpose. For complex message exchanges, multiple streams should be used.

Streams may be short lived for immediate data exchange or communication, or long lived for notifications if needed.

8.1.4 *Stream headers*

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

Definition 60 Header message**.**

```
syntax = "proto3";

package pb;

message Headers {
    repeated Header headers = 1;
}

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

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

Standard header key names are defined here:

1. tracing-span-context

8.1.5 *Encapsulation of context for tracing*

P2P Stream scoped tracing span context is exchanged by using stream headers. Header key "tracing-span-context" is reserved for binary encoded tracing span context data. This context should be used in tracing messages. The stream initiator node should

provide tracing span context to the responding node. This context is optional and all nodes must function the same as span context is provided by other nodes or not regardless if the node has tracing configured or not.

8.2 BZZ HANDSHAKE PROTOCOL

The bzz handshake protocol is the protocol that is always run after two peers are connected and before any other protocols are established. It communicates information about the peer's address, network ID and light node capability.

The handshake protocol defines only one stream and three messages:

Definition 61 Bzz handshake protocol messages**.**

```
// ID: /swarm/handshake/1.0.0/handshake

syntax = "proto3";

package handshake;

message Syn {
    bytes ObservedUnderlay = 1;
}

message Ack {
    BzzAddress Address = 1;
    uint64 NetworkID = 2;
    bool Light = 3;
}

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

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

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

Upon connection, requesting peer constructs a new handshake stream and sends a `Syn` message with the remotely observed Underlay address of the peer it is performing the handshake with. After that it waits for `SynAck` response message from the responding peer. In the `SynAck` message responder sends it own `Syn` message as well, together with the acknowledgement message which should include it's correctly signed BzzAddress. The received observed address can be used and compared with locally known addresses in order to send the better advertisable address (Underlay) in the acknowledgement. After the requesting peer receives the `SynAck` message from the responding peer and validates that the received `Ack` information in it is correct, it sends an `Ack` message itself as a confirmation to the responding peer. The stream is closed by the responding peer after it receives the `Ack` message.

The connection must be terminated if network IDs are do not match or if the exact order of messages is not followed.

The bzz address is verified and overlay, underlay and signature are extracted. light is a boolean field indicating whether the node is operating as a light (as opposed to full) node.

After the handshake, each peer should remember the following data about the other:

- the overlay address - used in forwarding (see 9.2),
- the underlay address - used for dialing, passed to the underlay network protocol when the connectivity driver needs to connect to the peer (see 9.1),
- the bzz address signature - needed by the hive protocol to pass information about the node to other peers (see 8.3),
- whether the peer is a light node.

8.3 HIVE DISCOVERY

The hive protocol enables nodes to exchange information about other peers that are relevant to them in order to bootstrap their connectivity (see 2.1.4) . The information communicated are both overlay and underlay addresses of the known remote peers (see 7.2). The overlay address serves to select peers to achieve the connectivity pattern needed for the desired network topology. Underlay address is needed to establish the peer connections by dialing selected peers.

8.3.1 *Streams and messages*

The protocol specifies one stream with two messages:

Definition 62 Hive protocol messages**.**

```
// /swarm/hive/1.0.0/peers                                    1
syntax = "proto3";                                            2
```

```
package hive;

message Peers {
    repeated BzzAddress peers = 1;
}

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

During the lifetime of connection, nodes can broadcast newly received peers to their peers. This is done by sending the `Peers` message over the `/swarm/hive/1.0.0/peers` stream.

Upon receiving a peers message, nodes are meant to store the peer information in their address book, i.e., a data structure containing info about peers known to the node. The address book is meant to be used to suggest peers to a connectivity manager according to a connection strategy (9.1) in order to bootstrap kademlia topology (111). The address book is meant to be persisted across sessions.

Sending side

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

Receiving side

When the stream is created, receiving node should wait for a `Peers` message. After receiving the message, node should close its side of the stream to let the sender node know that the message was received, and move on with processing. If the new node was not known, it should also be forwarded to all connected peers closer to peer address then the node themselves.

8.4 RETRIEVAL

Definition 63 Retrieval protocol messages**.**

```
// /swarm/retrieval/1.0.0/
syntax = "proto3";
```

```
message Request {
    bytes Addr = 1;
}

message Delivery {
    bytes Data = 1;
}
```

8.5 PUSH-SYNCING

Definition 64 Push-sync protocol messages.

```
// /swarm/push-sync/1.0.0/peers
syntax = "proto3";

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

message Receipt {
  bytes Address = 1;
}
```

8.6 PULL-SYNCING

8.7 SWAP SETTLEMENT PROTOCOL

9

STRATEGIES

The strategies clients follow will have a fundamental effect on the behaviour of the network and if they end up going against the preconceived design, the project may very well fail to suit user expectations. Such scenarios can easily turn fatal to the project, which is why it is instructive to err on the side of caution when change (or initially suggest) strategies for node behaviour. The scope of strategies are defined as those aspects of the intended 'protocol' which cannot be directly observed or easily verified. As an example contrast the very act of forwarding an incoming retrieve request (strategy) with the act of using correctly formatted and serialised messages when doing so (protocol constraint). The latter can be immediately detected and be responded to by disconnecting and blacklisting offenders. In contrast, whether a node does forwarding in a way conducive to the desired network outcome is subtle to detect.

Since we choose to work with the narrowest possible assumption of profit-maximising node operators, the choice of strategies ought to be course grained enough to evaluate the consequences of the options. In particular we must not allow for scenarios when even vaguely rational deviations from the recommended strategy have catastrophic effects.

In general incentives should be in place to guarantee that behaviour that is detrimental to the service incurs a risk of subsequent loss, deterrent upfront cost or opens up reciprocal vulnerability.

The constraints put on strategic choice are crucial in terms of rendering the game theory feasible to simulations and experiments or express them with simple enough analytical models to aid reasoning.

9.1 CONNECTION

All new (previously not known) peers found in the `Peers` message received from either of the two streams as well as newly connected peers that dialled in should be automatically broadcast to all connected peers that are closer to them than the node's base overlay address. Formally, node s (sender) notifies an existing peer r (recipient) about peer p if $PO(s,r) = PO(s,p)$.

9.1.1 *Connectivity and its constraints*

Without resource constraints the optimal connectivity for a proper storer node is full connectivity, i.e. when a node has all other nodes in the network as their peer. The chequebook contract also prefers a small number of peer connections to be maintained and other network overhead costs for keepalive connections or network socket shortage also implicates that above a certain network size full connectivity is prohibitive.

The relative gain is maximised if throughput on all keepalive connections are maximised. If network contention prevents a node from forwarding to a peer instantaneously, another peer must to be chosen. Assuming uniformity in the network throughput maximisation in the context of limited number of peer connections can be achieved with a connectivity pattern where each kademlia bin has a constant cardinality of 2^b and peers in the bin are balanced, i.e., match each distinct bit prefix of length b. Therefore, connectivity strategy can be formulated as follows:

9.1.2 *Light node connection strategy*

Light nodes in the context of connection topology are nodes that do not have kademlia connectivity. In the extreme case, a light node should be able to get away with a single connection. The lack of full connectivity is indicated to the node's peers so that no retrieve requests or push sync requests are sent to them. Note that if a node falsely indicates its status, that should cause minimal disruption.

A lightnode wants to identify its neighboorhood so that it connects to at least one of the R storer nodes closest to it (where R is the redundancy parameter determining the minimum neighbourhood size).

9.2 FORWARDING

Forwarding strategy refers to the process a node follows pertaining to relayed messages and responses in Forwarding Kademlia as relevant for chunk retrieval and upload.

Note that for retrieval as well as push-syncing, the protocol does not specify just allows forwarding of retrieve requests and chunk deliveries, respectively. Whether a node forwards an incoming message and where it forwards them, should be guided by the incentives in order to attain stability.

As described in 3.1.2, nodes formulate and advertise their prices for each PO bin separately. The optimal pricing strategy will reflect the PO bins in that the prices for closer bins will monotonically decrease.

With sufficiently

Retries

Both push-sync and retrieval messages are using backwarding (i.e., pass-back responses), and the compensation for forwarding only gets accounted for once the response is sent. This incentivises nodes to watch on peer connections where the downstream message was sent. In particular, if downstream peer disconnects before the response in received, the forwarding should be repeated and tried on another peer.

9.3 PRICING

In the following we describe 3 strategies: each is more complex than the previous.

passive: weak cartell - hard wired fix price table minimal responsiveness: if margin is not guaranteed, the peer is not forwarded to.

reactive - respond to downstream price increase with exploring alternative peers. If there is no alternative it raises the price. If downstream peers drop the price, it follows suit.

proactive: respond to

9.3.1 *Passive strategy*

The following strategy is a basic way on how nodes *can* use the protocol described above. The strategy intends to be as easy as possible to implement, while balbalbala

Margin

A nodes price for a 'ChunkDeliveryRequest' is always at least his desired 'margin' + the price of the cheapest upstream peer.

Filling the priceInformationRegistry

Initially, nodes only know the price for serving content from the most proximate bin (price == 'margin'). When this node gets his first request for serving content that is one proximity order away from the border of his most proximate bin, he will first of all check that the price for this request is more than his margin. If this is the case, he will forward the 'RetrieveRequest' with a price of 0 (since the state is not initialized yet). If the upstream peer in the appropriate bin did not change his price, this peer will get back a 'NewPrice' message, where the price equals the 'margin' of the downstream peer. If the original price is at least 2x the 'margin', the node can forward the request with a price of 'margin' (and pocket the other marin himself). From this point onwards, the 'state' is initialized for one peer and one proximity. Subsequent 'RetrieveRequests' to this peer will be send with a 'price' of 'margin'. Chunks which are further away will have a higher price. The peer will quickly learn about this via this iterative process. If all peers apply the same 'margin', the price of a 'ChunkDelivery' equals the proximity

order between the requestor and the chunk times the 'margin'. If a peer get's a request for a chunk with a price that is less than his 'margin' + the price of the cheapest upstream peer in that bin, the request is not forwarded, but immediately answered by a 'NewPrice' message with a price equal to the 'margin' + the price of the cheapest upstream peer in that bin.

Reacting to price differences

In a network where no peer ever charges a 'margin' different than the default margin, 'NewPrice' messages will be only send to peers who have not fully initialized their 'state' and, given equal proximity between the requested chunk and the peer in a particular bin, a node does not have a preferred supplier of a chunk based on the price. If any peer in the network would change his price, however, a preference based on price will emerge:

For any chunk request: - Select the bin to forward the request to - Select the peers who are closest to the chunk in terms of PO - Select the peer with the lowest price - Do a round-robin load balancing if there are multiple peers with the same price

From this, we can conclude that if a peer lowers his price, he is expected to receive more 'RetrieveRequests' and if a peer increases his price, he is expected to receive fewer 'RetrieveRequests'. A node who is connected to an upstream peer with a lower price may decide to lower his price for this distance as well in order to receive more requests.

Notifying about price changes

It is in the interest of the network, that when there is a price change this gets propagated as soon as possible to the relevant parties. A price change can be initiated by a change of 'margin', or by a change of the price from upstream peers. In both cases, a peer might decide not to notify other peers about the decrease in price–the only thing which changes is that it will accept 'RetrieveRequests' with a lower price than before. Without notifying the downstream peers, however, a node might not get the expected amount of additional clients directly, as the downstream peers might never propose a lower price than before. For this reason, we propose that nodes pro-actively send 'NewPrice' messages to his peers, essentially requesting them to update their 'priceInformationRegistry'. In the case of pro-active 'NewPrice' messages, the 'chunkReference' may be synthetic, meaning it does not have to respond to an actual chunk; all that is important is that peers update their 'priceInformationRegistry'.

Overpriced bins Since the price of a node depends on the price of his upstream peers, it might happen that by change, all his upstream peer-connections in a certain bin are populated by expensive peers—effectively making the node itself to be too expensive as well for his downstream peers. To solve this issue, we propose to do statistical analysis: if a node has sensible prices across his bins, it is expected that he receives an equal amount of requests for all his bins. If a node receives statistically

significantly less requests for one bin compared to his other bins, he marks the supposedly overpriced peers as non-functional and the hive protocol should kick in to suggest new peer connections.

Reacting to demand increases/decreases

A node that decreases its price such that he is the cheapest, is expected to get much more traffic. Without an adequate response to an increase in traffic, a node might be forced to go offline as D

Configuration options

The following configuration options will be added.

```
| name | unit | default |
| -------- | -------- | -------- |
| ‘margin‘     | base currency of chequebook     | 0     |
| ‘period‘      | duration in seconds    | 300     |
| ‘maximum_upstream_bandwidth‘      | bandwidth usage in bytes per ‘period‘     | TODO
|‘high_water_mark‘    | percentage: (upstream bandwidth used / ‘period‘) / (‘maximum_u
| ‘low_water_mark‘    | percentage: (upstream bandwidth used / ‘period‘) / (‘maximum_u
| ‘margin_change‘    | percentage    | 1%     |
```

See below for how these variables are used.

9.3.2 *Behavior of a node*

1) Given a particular ‘chunkReference‘, a node chooses to send the ‘*ChunkRequestMessage*‘, usually, to the peer with the lowest price of all peers in that bin. 2) If a ‘newPrice‘ message is returned from a ‘chunkRequestMessage‘, the node updates the ‘priceInformationRegistry‘ and sends the ‘chunkRequestMessage‘ again, choosing a peer on the basis of the updated ‘priceInformationRegistry‘. 4) When a peer requests a certain ‘ChunkRequestMessage‘ and does not have the ‘chunkContent‘ in its local storage, it forwards the ‘chunkRequestMessage‘ (see 1) with a price equal to the ‘chunkRequestMessage‘ it received minus his ‘margin‘. 5) A node keeps track of his ‘bandWidth_usage‘ ((upstream bandwidth used / ‘period‘)/ (‘maximum_upstream_bandwidth‘)) 6) Whenever ‘bandWidth_usage‘ > ‘high_water_mark‘, the ‘margin‘ will be (1+‘margin_change‘) * ‘margin‘ 7) Whenever ‘bandWidth_usage‘ < ‘low_water_mark‘, the ‘margin‘ will be (1-‘margin_change‘) * ‘margin‘. ‘margin_change‘ indicates the 8) Whenever a node can get a ‘chunk‘ for a cheaper price than requested by a ‘ChunkRequestMessage‘ (upstream peers changed prices, amount of hops are less than the average request for the same distance), he answers with a ‘ChunkDeliveryMessage‘, which will be priced based on the ‘ChunkRequestMessage‘. 9) Whenever a node expects to be able to deliver long-term lower prices than currently known by his peers, he sends a ‘NewPrice‘ message to his peers.

9.4 ACCOUNTING AND SETTLEMENT

payment threshold
disconnect threshold
receiving cheque
cashing strategy
chequebook balance
topup strategy

9.5 PUSH-SYNCING

9.6 PULL-SYNCING

purpose of pull syncing
neighbourhood syncronisation
maximum resource utilisation strategy

9.7 GARBAGE COLLECTION

Garbage collection is the glorified name for the process of purging the local chunk store of the node. Here the assumption is that storage is s scarce resource and with time nodes will experience capacity shortage and face the problem of deciding which existing chunks to remove to make space for new arrivals.

The garbage collection strategy must reflect the economic potential of chunks, i.e., it should ideally maximise the profit the node makes on storage compensation. Deriving high-level behaviour of Swarm as a network from the assumption that nodes employ a particular strategy is only valid if said strategy align best with the incentive design.

If only Swap is operational only a very simple garbage collection strategy is sufficient. Every time a chunk is retrieved, the node serving it earns SWAP revenue. Ideally, then, those chunks that are retrieved more often should be preferred to those retrieved rarely.

The best predictor for number of accesses is number of accesses in the past. For chunks that exist and been retrieved, these models work well, but they are only optimal for making distinctions among popular, not among the rarely accessed content. A priori estimation for new chunks can instead rely on the postage value, which directly map onto guaranteed revenue.

available, value consistent storage
indexing by per-chunk balance
updating postage stamp value
combining swap profit

Let's index the epochs with negative integers with an ordering respecting recency, i.e., the current epoch is 0, the most recent -1, the previous one -2, etc. Let n be the

cutoff memory size for past epochs. Let $Hits(e)$ denote the number of hits a chunk is served during epoch e.

Let's build a popularity predictive model based on past observations using the assumption of exponential decay with time in predictive power.

$$Hits(0) \stackrel{\text{def}}{=} \frac{\sum_{e=0}^{n} \frac{Hits(e)}{c^e}}{c^{n-1}} \tag{3}$$

10

API-S

Definition 65 HTTP status codes used in swarm**.**

103	Checkpoint	returns temporary root hash for resumable uploads
200	ok	
201	Created	returned by POST requests upon successful creation of file/manifest/tag/stamp/
400	Bad request	returned if request or its parameters are not well formed or missing
401	Unauthorized	returned by access control if auth fails
402*	Payment Required	returned if no swap balance or missing/invalid postage stamp
403	Forbidden	returned if retrieved chunk is encrypted but the reference has no decryption key
404	Not Found	returned if a local prerequisite is not found or manifest path does not exist.
405*	Method Not Allowed	HTTP verb not allowed for this endpoint
406*	Not Acceptable	No format acceptable by the ACCEPT header explicit in the request.
408	Request Timeout	Retrieve requests fallback error after TTL passed
413	Payload Too Large	Payload size exceeds maximum chunk size or span given (chunk API)
414	URI Too Long	manifest path > 32
416	Range Not Satisfiable	offset in range query out of range
420	Enhance your calm	returned when recovery was initiated but retrieval timed out
422	Unprocessable Entity	returned by the blockchain external API if eth api returns an error or by single owner chunk post API if owner/id pair already exists

*Generic errors detectable before endpoint API call so not documented.

10.1.1 *Signer*

Definition 66 API endpoint GET `/sign/{id}/{document}`.

GET **`/sign/{id}/{document}`**
ECDSA signature

Route Parameters	
`id`	hex string - eth address
`document`	(hex) string - document to sign (is prefixed and hashed before signing)

Response		application/json
200	ok	
401	Unauthorised	failed authentication on existing identity
404	Not found	unknown identity

Definition 67 API endpoint GET `/dh/{id}/{pubkey}`.

GET **`/dh/{id}/{pubkey}`**
Diffie-Hellman shared secret

Route Parameters	
`id`	hex string - eth address
`pubkey`	hex string - represents the remote party in the shared secret arrangement

Response		application/json
200	ok	
401	Unauthorised	failed authentication on existing identity
404	Not found	unknown identity

10.1.2 *Blockchain*

Definition 68 API endpoint GET `/eth/{contract}/{function}/{args}`.

GET **`/eth/{contract}/{function}/{args}`**

ethereum API call

Route Parameters

`contract`	hex string, eth address of contract
`function`	endpoint within contract
`args`	arguments for the eth API call

Response application/json

200	ok	
400	Bad request	unknown contract or function endpoint given
404	Not Found	unknown contract or function endpoint
422	Unprocessable entity	incorrect ABI, error by eth API

Definition 69 API endpoint POST `/eth/{contract}/{function}/{args}`.

POST **`/eth/{contract}/{function}/{args}`**

ethereum API send transaction

Route Parameters

`contract`	hex string, eth address of contract
`function`	endpoint within contract
`args`	arguments for the eth API call

Response application/json

200	ok	
401	Unauthorised	failure signing transaction
400	Bad request	unknown contract or function endpoint given
404	Not Found	unknown contract or function endpoint
422	Unprocessable entity	incorrect ABI, error by eth API

10.1.3 *User input*

Definition 70 API endpoint GET `/input/{id}`.

GET **/input/{id}**

Route Parameters	
id	hex string - eth address of the persona the input is expected from
note	the question to be answered or instruction to select

Response	application/json
200	ok

10.2 STORAGE API

10.2.1 *Chunks*

Definition 71 API endpoint GET `/chunk/{reference}`.

GET **/chunk/{reference}**
Retrieve chunk as in 23

Route Parameters	
reference	hex string

Response		application/json
200	ok	Chunk data as response body
403	Forbidden	chunk encrypted but no decryption key in reference
408	Request Timeout	
420	Enhance your calm	Recovery initiated but request timed out

Definition 72 API endpoint POST `/chunk/(?span={span})`.

POST **`/chunk/(?span={span})`**
Create chunk as in 24

Query Parameters

`span`	integer

Header Parameters

`SWARM-TAG`	hex string
`SWARM-STAMP`	hex string
`SWARM-ENCRYPTION`	hex string
`SWARM-PIN`	bool
`SWARM-PARITIES`	integer

Response application/json

200	ok	Minimal manifest entry in response body
400	Bad request	Span parameter not well formed.
413	Payload Too Large	Payload size exceeds span value or 4096

Definition 73 API endpoint GET `/soc/{owner}/{id}`.

GET **`/soc/{owner}/{id}`**
Retrieve single owner chunk as in 26

Route Parameters

`owner`	eth address of single owner
`id`	identifier within owner namespace

Query Parameters

`key`	string

Response application/json

200	ok	single owner chunk payload in response body
400	Bad request	if owner, id or key is not well formed
403	Forbidden	Single owner chunk encrypted but no decryption key given
408	Request Timeout	
420	Enhance your calm	Recovery initiated but request timed out

Definition 74 API endpoint POST `/soc/{owner}/{id}?span={span}`.

POST `/soc/{owner}/{id}?span={span}`

Create new single owner chunk as in 27

Route Parameters

`owner`	eth address of single owner
`id`	identifier within owner namespace

Header Parameters

`SWARM-TAG`	hex string
`SWARM-STAMP`	hex string
`SWARM-ENCRYPTION`	hex string
`SWARM-PIN`	bool
`SWARM-PARITIES`	integer

Query Parameters

`span`	integer

Response application/json

201	Created	Reference in response body
400	Bad request	owner, id or span is not well formed
401	Unauthorized	signing fails
404	Not found	owner keypair is found
413	Payload Too Large	Payload size exceeds span value or 4096
422	Unprocessable entity	owner/id pair already exists

10.2.2 *File*

Definition 75 API endpoint POST `/file/`.

POST **`/file/`**

Path description as in 30

Header Parameters

`SWARM-TAG`	hex string
`SWARM-STAMP`	hex string
`SWARM-ENCRYPTION`	hex string
`SWARM-PIN`	bool
`SWARM-PARITIES`	integer

Response application/json

201	Created	Minimal manifest entry as response body

Definition 76 API endpoint PUT `/file/{reference} .`

PUT **`/file/{reference}`**

Append body to file, returns new reference. Note that any intermediate chunk of a file is a file.

Route Parameters

`reference`	hex string
`file`	binary as request body

Header Parameters

`SWARM-TAG`	hex string
`SWARM-STAMP`	hex string
`SWARM-ENCRYPTION`	hex string
`SWARM-PIN`	bool
`SWARM-PARITIES`	integer

Response application/json

201	Created	Minimal manifest entry as request body
400	Bad Request	Reference is not well formed
403	Forbidden	Encrypted content but no decryption key in. reference
408	Request Timeout	Retrieval of file to append to times out
420	Enhance your calm	Recovery initiated but request timed out
413	Payload Too Large	

Definition 77 API endpoint GET `/file/{reference}`.

GET **`/file/{reference}`**

Retrieve file by reference as in 31

Route Parameters

`reference`	string

Response application/json

200	ok	file contents streamed in response body
400	Bad Request	Reference is not well formed
403	Forbidden	Encrypted content but no decryption key in reference
408	Request Timeout	
416	Range Not Satisfiable	Offset in range query out of range
420	Enhance your calm	Recovery initiated but request timed out

10.2.3 *Manifest*

Definition 78 API endpoint GET `/manifest/{reference}/{path}`.

GET **`/manifest/{reference}/{path}`**

Lookup entry by path in manifest, see 35

Route Parameters

`reference`	hex string
`path`	string

Response application/json

200	ok	Manifest entry in response body
400	Bad Request	Reference or path is not well formed
403	Forbidden	Encrypted content but no decryption key in reference
404	Not found	Path does not exist
408	Request Timeout	Timeout retrieving referenced manifest
420	Enhance your calm	Recovery initiated but request timed out

Definition 79 API endpoint DELETE `/manifest/{reference}/{path} .`

DELETE **`/manifest/{reference}/{path}`**

Delete entry on path in referenced manifest, see 37

Route Parameters

`reference`	hex string
`path`	string

Response application/json

204	No content	Successful deletion
400	Bad Request	Reference or path is not well formed
403	Forbidden	Encrypted content but no decryption key in reference
408	Request Timeout	Timeout retrieving referenced manifest
420	Enhance your calm	Recovery initiated but request timed out

Definition 80 API endpoint PUT `/manifest/{reference}/{path}`.

PUT **`/manifest/{reference}/{path}`**

Update manifest as in 36

Route Parameters

`reference`	hex string
`path`	(hex) string

Response application/json

200	ok	
400	Bad Request	Reference is not well formed
403	Forbidden	Encrypted content but no decryption key in reference
404	Not found	Path does not exist
408	Request Timeout	Timeout retrieving referenced manifest
414	URI Too Long	Path exceeds 32 byte limit
420	Enhance your calm	Recovery initiated but request timed out

Definition 81 API endpoint POST `/manifest/{old}/{new}`.

POST **`/manifest/{old}/{new}`**

Merge manifests as in 38

Route Parameters

`old`	hex string
`new`	hex string

Response application/json

201	Created	Reference in response body
400	Bad Request	Reference is not well formed
403	Forbidden	Encrypted content but no decryption key in reference
404	Not found	Path does not exist
408	Request Timeout	Timeout retrieving referenced manifest
414	URI Too Long	Path exceeds 32 byte limit
420	Enhance your calm	Recovery initiated but request timed out

10.2.4 *High level storage API*

Definition 82 API endpoint GET `/bzz:/{host}/{path}`.

GET **`/bzz:/{host}/{path}`**

Download file

Route Parameters

`host`

`path`

Header Parameters

`SWARM-TAG`	hex string
`SWARM-STAMP`	hex string
`SWARM-ENCRYPTION`	hex string
`SWARM-PIN`	bool
`SWARM-PARITIES`	integer

Response application/json

200	ok	
400	Bad Request	Host/Reference or path is not well formed
401	Unauthorized	Access denied: AC unlock failed
403	Forbidden	Encrypted content but no decryption key in reference
404	Not found	Host cannot be resolved or Path does not exist
408	Request Timeout	Timeout retrieving referenced manifest
420	Enhance your calm	Recovery initiated but request timed out

Definition 83 API endpoint PUT `/bzz:/{host}/{path}`.

PUT **`/bzz:/{host}/{path}`**

Append upload to file referenced, add new entry to path, returns new manifest

Route Parameters

`file/collection`	as request body

Header Parameters

`SWARM-TAG`	hex string
`SWARM-STAMP`	hex string
`SWARM-ENCRYPTION`	hex string
`SWARM-PIN`	bool
`SWARM-PARITIES`	integer

Response application/json

201	Created	New manifest root reference in response body
400	Bad Request	Host/Reference is not well formed
401	Unauthorized	Accessdenied: AC unlock failed
403	Forbidden	Encrypted content but no decryption key in reference
404	Not found	Host cannot be resolved or Path does not exist
408	Request Timeout	Timeout retrieving referenced manifest
414	URI Too Long	Path exceeds 32 byte limit
420	Enhance your calm	Recovery initiated but request timed out

Definition 84 API endpoint POST `/bzz:/{host}/{path}`.

POST **`/bzz:/{host}/{path}`**

Upload file or collection, returns new manifest root reference

Route Parameters

`file/collection`	as request body

Header Parameters

`SWARM-TAG`	hex string
`SWARM-STAMP`	hex string
`SWARM-ENCRYPTION`	hex string
`SWARM-PIN`	bool
`SWARM-PARITIES`	integer

Response application/json

201	Created	New manifest root reference in response body
400	Bad Request	Host/Reference is not well formed
401	Unauthorized	Accessdenied: AC unlock failed
403	Forbidden	Encrypted content but no decryption key in reference
404	Not found	Host cannot be resolved or Path does not exist
408	Request Timeout	Timeout retrieving referenced manifest
414	URI Too Long	Path exceeds 32 byte limit
420	Enhance your calm	Recovery initiated but request timed out

10.2.5 *Tags*

Definition 85 API endpoint POST `/tags`.

POST **`/tags`**

Create new tag

Response application/json

201	Created	ID in response body

Definition 86 API endpoint GET `/tags?offset={offset}&length={length}`.

GET **`/tags?offset={offset}&length={length}`**
Get all tags

Query Parameters

`offset` integer
`length` integer

Response application/json

200 ok

Definition 87 API endpoint GET `/tags/{id}`.

GET **`/tags/{id}`**
View details tag with given ID

Route Parameters

`id` string

Response application/json

200 ok
400 Bad Request ID not well formed
404 Not found Tag with ID does not exists.

Definition 88 API endpoint DELETE `/tags/{id}`.

DELETE **`/tags/{id}`**
Path description

Route Parameters

`id` string

Response application/json

204 No content
400 Bad Request ID not well formed
404 Not found Tag with ID does not exists.

Definition 89 API endpoint GET `/pin/?offset={offset}&length={length}`.

GET **`/pin/?offset={offset}&length={length}`**
List Pinned Content and metadata

Query Parameters

`offset` integer
`length` integer

Response application/json

200 ok

Definition 90 API endpoint GET `/pin/{id}`.

GET **`/pin/{id}`**
View Pinned Content and metadata

Query Parameters

`id` hex string

Response application/json

200	ok	pin in response body
400	Bad Request	ID not well formed
404	Not found	Pin with ID does not exists.

Definition 91 API endpoint PUT `/pin/{id}`.

PUT **`/pin/{id}`**
Pin an already uploaded content

Route Parameters

`id` hex string

Response application/json

200	ok	
400	Bad Request	ID not well formed
404	Not found	Pin with ID does not exists.

Definition 92 API endpoint DELETE `/pin/{id}`.

DELETE **`/pin/{id}`**
Remove pinning from content

Route Parameters

`id`

Response application/json

204	No Content	
400	Bad Request	ID not well formed
404	Not found	Pin with ID does not exists.

10.2.7 *Swap and chequebook*

10.2.8 *Postage stamps*

Definition 93 API endpoint GET `/stamp?offset={offset}&length={length}`.

GET **`/stamp?offset={offset}&length={length}`**

View all postage stamps

Query Parameters

`offset`	integer
`length`	integer

Response application/json

200	ok

Definition 94 API endpoint GET `/stamp/{id}`.

GET **`/stamp/{id}`**

View postage stamp with id

Route Parameters

`id`

Response application/json

200	ok	
400	Bad Request	ID not well formed
404	Not found	Stamp with ID does not exists.

Definition 95 API endpoint PUT `/stamp/{id}`.

PUT **`/stamp/{id}`**
Top up postage stamp with id

Route Parameters

`id`

Query Parameters

`amount` integer

Response application/json

200	ok	
400	Bad Request	ID not well formed
404	Not found	Stamp with ID does not exists.

Definition 96 API endpoint DELETE `/stamp/{id}`.

DELETE **`/stamp/{id}`**
Drain and expire stamp with id

Route Parameters

`id`

Response application/json

200	ok	
400	Bad Request	ID not well formed
404	Not found	Stamp with ID does not exists.

Definition 97 API endpoint POST `/stamp/`.

POST **`/stamp/`**
Create a new postage stamp, return ID in reponse body

Route Parameters

Response application/json

201	created	ID in response body

Definition 98 API endpoint POST `/access/{address}` .

POST **`/access/{address}`**
Lock ACT for address as in 46

Route Parameters

`address` hex string

Response application/json

201	Created	root access manifest reference in response body
400	Bad Request	Encrypted content but no decryption key in reference
403	Forbidden	Encrypted content but no decryption key in reference
404	Not found	
408	Request Timeout	Timeout retrieving referenced manifest
420	Enhance your calm	Recovery initiated but request timed out

Definition 99 API endpoint GET `/access/{address}` .

GET **`/access/{address}`**
Unlock ACT for address as in 46

Route Parameters

`address` hex string

Response application/json

200	ok	
400	Bad Request	Address not well formed
401	Unauthorized	Access denied: AC unlock failed
403	Forbidden	Encrypted content but no decryption key in reference
408	Request Timeout	Timeout retrieving referenced manifest
420	Enhance your calm	Recovery initiated but request timed out

Definition 100 API endpoint PUT `/access/{root}/{pubkey}`.

PUT **`/access/{root}/{pubkey}`**

Add entry for pubkey to the ACT referred in the root access manifest 47

Route Parameters

`root`	hex string - reference to root access manifest
`pubkey`	hex string - public key of grantee

Response application/json

201	Created	Reference to new manifest root in response body
400	Bad Request	Address or public key not well formed
401	Unauthorized	Permission denied: creating session key failed
403	Forbidden	Encrypted content but no decryption key in reference
408	Request Timeout	Timeout retrieving referenced manifest
420	Enhance your calm	Recovery initiated but request timed out

Definition 101 API endpoint DELETE `/access/{root}/{pubkey}`.

DELETE **`/access/{root}/{pubkey}`**

Remove entry for pubkey from ACT referred in the root access manifest, see 47

Route Parameters

`root`	hex string - reference to root access manifest
`pubkey`	hex string - public key of grantee

Response application/json

201	Created	Reference to new manifest root in response body
400	Bad Request	Address or public key not well formed
401	Unauthorized	Permission denied: creating session key failed
403	Forbidden	Encrypted content but no decryption key in reference
408	Request Timeout	Timeout retrieving referenced manifest
420	Enhance your calm	Recovery initiated but request timed out

10.3 COMMUNICATIONS

10.3.1 *PSS*

Definition 102 API endpoint POST `/pss/send/{topic}(?targets={targets}&recipient={recipien`

POST **`/pss/send/{topic}(?targets={targets}&recipient={recipient})`**
Send private message with topic to targets, encrypted for recipient, see 52

Route Parameters

`topic` string

Query Parameters

`recipient` hex string - recipient public key for encryption
`targets` hex string - comma separated list of targets

Header Parameters

`SWARM-TAG` hex string
`SWARM-STAMP` hex string

Response application/json

209 sent Tag to monitor
400 Bad Request Topic, targets or recipient not well formed.

Definition 103 API endpoint POST `/pss/subscribe/{topic}/(?on={channel})`.

POST **`/pss/subscribe/{topic}/(?on={channel})`**
Subscribe to messages with topic to be delivered on given channel, see 53

Route Parameters

`topic` string

Query Parameters

`on` hex string - channel ID

Response application/json

201 Created
400 Bad Request Topic. or channel not well formed.

Definition 104 API endpoint DELETE `/pss/subscribe/{topic}/(?on={channel})`.

DELETE **`/pss/subscribe/{topic}/(?on={channel})`**
Unsubscribe for topic on channel, see 53

Route Parameters

`topic` string

Query Parameters

`on` hex string - channel ID

Header Parameters

`SWARM-TAG` hex string
`SWARM-STAMP` hex string

Response application/json

204 No content Successfully uninstall
400 Bad Request Topic. or channel not well formed.

10.3.2 *Feeds*

BIBLIOGRAPHY

[Alwen et al., 2019] Alwen, J., Coretti, S., and Dodis, Y. (2019). The double ratchet: Security notions, proofs, and modularization for the signal protocol. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 129–158. Springer.

[Aspnes and Shah, 2007] Aspnes, J. and Shah, G. (2007). Skip graphs. *Acm transactions on algorithms (talg)*, 3(4):37–es.

[Baumgart and Mies, 2007] Baumgart, I. and Mies, S. (2007). S/kademlia: A practicable approach towards secure key-based routing. In *Parallel and Distributed Systems, 2007 International Conference on*, volume 2, pages 1–8. IEEE.

[BitTorrent Foundation, 2019] BitTorrent Foundation (2019). Bittorrent white paper.

[Bloemer et al., 1995] Bloemer, J., Kalfane, M., Karp, R., Karpinski, M., Luby, M., and Zuckerman, D. (1995). An xor-based erasure-resilient coding scheme. Technical report, International Computer Science Institute. Technical Report TR-95-048.

[Carlson, 2010] Carlson, N. (2010). Well, these new zuckerberg ims won't help facebook's privacy problems.

[Cohen, 2003] Cohen, B. (2003). Incentives build robustness in bittorrent. In *Workshop on Economics of Peer-to-Peer systems*, volume 6, pages 68–72.

[Crosby and Wallach, 2007] Crosby, S. A. and Wallach, D. S. (2007). An analysis of bittorrent's two Kademlia-based dhts. Technical report, Citeseer.

[Economist, 2020a] Economist (2020a). A deluge of data is giving rise to a new economy. [Online; accessed 26. Feb. 2020].

[Economist, 2020b] Economist (2020b). Governments are erecting borders for data. [Online; accessed 27. Feb. 2020].

[Economist, 2020c] Economist (2020c). Who will benefit most from the data economy? [Online; accessed 27. Feb. 2020].

[European Commission, 2020a] European Commission (2020a). European data strategy. [Online; accessed 3. Mar. 2020].

[European Commission, 2020b] European Commission (2020b). On Artificial Intelligence - A European approach to excellence and trust. Technical report, European Commission.

[Filecoin, 2014] Filecoin (2014). Filecoin: a cryptocurrency operated file storage network.

[Ghosh et al., 2014] Ghosh, M., Richardson, M., Ford, B., and Jansen, R. (2014). A TorPath to TorCoin: Proof-of-bandwidth altcoins for compensating relays. Technical report, petsymposium.

[Harari, 2020] Harari, Y. (2020). Yuval Harari's blistering warning to Davos. [Online; accessed 2. Mar. 2020].

[Heep, 2010] Heep, B. (2010). R/kademlia: Recursive and topology-aware overlay routing. In *Telecommunication Networks and Applications Conference (ATNAC), 2010 Australasian*, pages 102–107. IEEE.

[Hughes, 1993] Hughes, E. (1993). A Cypherpunk's Manifesto. [Online; accessed 7. Aug. 2020].

[IPFS, 2014] IPFS (2014). Interplanetary file system.

[Jansen et al., 2014] Jansen, R., Miller, A., Syverson, P., and Ford, B. (2014). From onions to shallots: Rewarding tor relays with TEARS. Technical report, DTIC Document.

[Kwon et al., 2016] Kwon, A., Lazar, D., Devadas, S., and Ford, B. (2016). Riffle: An efficient communication system with strong anonymity. In *Proceedings on Privacy Enhancing Technologies 2016*, pages 1–20. de Gruyter.

[Lee, 2018] Lee, K.-F. (2018). *AI Superpowers: China, Silicon Valley, and the New World Order*. Houghton Mifflin Harcourt.

[Locher et al., 2006] Locher, T., Moore, P., Schmid, S., and Wattenhofer, R. (2006). Free riding in bittorrent is cheap.

[Lua et al., 2005] Lua, E. K., Crowcroft, J., Pias, M., Sharma, R., and Lim, S. (2005). A survey and comparison of peer-to-peer overlay network schemes. *IEEE Communications Surveys & Tutorials*, 7(2):72–93.

[Marlinspike and Perrin, 2016] Marlinspike, M. and Perrin, T. (2016). The x3dh key agreement protocol. *Open Whisper Systems*.

[Maymounkov and Mazieres, 2002] Maymounkov, P. and Mazieres, D. (2002). Kademlia: A peer-to-peer information system based on the xor metric. In *Peer-to-Peer Systems*, pages 53–65. Springer.

[Merkle, 1980] Merkle, R. C. (1980). Protocols for public key cryptosystems. In *Proc. 1980 Symposium on Security and Privacy, IEEE Computer Society*, page 122. IEEE.

[Miller et al., 2014] Miller, A., Juels, A., Shi, E., Parno, B., and Katz, J. (2014). Permacoin: Repurposing bitcoin work for data preservation. In *Security and Privacy (SP), 2014 IEEE Symposium on*, pages 475–490. IEEE.

[Percival, 2009] Percival, C. (2009). Stronger key derivation via sequential memory-hard functions.

[Perrin and Marlinspike, 2016] Perrin, T. and Marlinspike, M. (2016). The double ratchet algorithm. *GitHub wiki*.

[Piatek et al., 2007] Piatek, M., Isdal, T., Anderson, T., Krishnamurthy, A., and Venkataramani, A. (2007). Do incentives build robustness in bittorrent. In *Proceedings of NSDI; 4th USENIX Symposium on Networked Systems Design and Implementation*.

[Plank et al., 2009] Plank, J. S., Luo, J., Schuman, C. D., Xu, L., Wilcox-O'Hearn, Z., et al. (2009). A performance evaluation and examination of open-source erasure coding libraries for storage. In *FAST*, volume 9, pages 253–265.

[Plank and Xu, 2006] Plank, J. S. and Xu, L. (2006). Optimizing Cauchy Reed-Solomon codes for fault-tolerant network storage applications. In *Network Computing and Applications, 2006. NCA 2006. Fifth IEEE International Symposium on*, pages 173–180. IEEE.

[Pouwelse et al., 2005] Pouwelse, J., Garbacki, P., Epema, D., and Sips, H. (2005). The bittorrent p2p file-sharing system: Measurements and analysis. In Castro, M. and van Renesse, R., editors, *Peer-to-Peer Systems IV*, pages 205–216, Berlin, Heidelberg. Springer Berlin Heidelberg.

[Roos et al., 2013] Roos, S., Salah, H., and Strufe, T. (2013). Comprehending Kademlia routing: A theoretical framework for the hop count distribution. *arXiv preprint arXiv:1307.7000*.

[Roos et al., 2015] Roos, S., Salah, H., and Strufe, T. (2015). Determining the hop count in Kademlia-type systems. In *IEEE ICCCN*.

[Rowstron and Druschel, 2001] Rowstron, A. and Druschel, P. (2001). Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *IFIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing*, pages 329–350. Springer.

[Schneier, 2019] Schneier, B. (2019). Data Is a Toxic Asset - Schneier on Security. [Online; accessed 6. Aug. 2020].

[Tron et al., 2016] Tron, V., Fischer, A., A, D. N., Felföldi, Z., and Johnson, N. (2016). swap, swear and swindle: incentive system for swarm. Technical report, Ethersphere. Ethersphere Orange Papers 1.

[Tron et al., 2019a] Tron, V., Fischer, A., and Nagy, D. A. (2019a). Generalised swap swear and swindle games. Technical report, Ethersphere. draft.

[Tron et al., 2019b] Tron, V., Fischer, A., and Nagy, D. A. (2019b). Swarm: a decentralised peer-to-peer network for messaging andstorage. Technical report, Ethersphere. draft.

[Tron Foundation, 2019] Tron Foundation (2019). Tron: Advanced decentralised blockchain platform.

[Vorick and Champine, 2014] Vorick, D. and Champine, L. (2014). Sia: Simple decentralized storage. Technical report, Sia.

[ZeroNet community, 2019] ZeroNet community (2019). Zeronet documentation.

[Zhao et al., 2004] Zhao, B. Y., Huang, L., Stribling, J., Rhea, S. C., Joseph, A. D., and Kubiatowicz, J. D. (2004). Tapestry: A resilient global-scale overlay for service deployment. *IEEE Journal on selected areas in communications*, 22(1):41–53.

Part IV

APPENDIX

A

FORMALISATIONS AND PROOFS

A.1 LOGARITHMIC DISTANCE AND PROXIMITY ORDER

Consider the set of bit sequences with fixed length d as points in a space. We can define a distance metric χ such that the distance between two such sequences is defined as the bigendian numerical value of their bitwise XOR ($\veebar$).

Definition 105 XOR distance (χ)**.**

$$\chi(x,y) \stackrel{\text{def}}{=} \mathcal{U}int(x \veebar y) \tag{4}$$

Given the fixed length $d > 0$, there is a maximum distance in this space, and thus we can define the notion of *normalised distance* and its inverse, *proximity*:

Definition 106 normalised XOR distance ($\overline{\chi}$)**.**

$$\overline{\chi}(x,y) \stackrel{\text{def}}{=} \frac{\chi(x,y)}{2^d - 1} \tag{5}$$

Definition 107 proximity**.**

$$Proximity(x,y) \stackrel{\text{def}}{=} \frac{1}{\overline{\chi}(x,y)} \tag{6}$$

Proximity order (PO) is a discrete logarithmic scaling of proximity.

Definition 108 Proximity order (*PO*)**.**

$$\begin{aligned} PO &: \mathcal{K} \times \mathcal{K} \mapsto \overline{0,d} \\ PO(x,y) &\stackrel{\text{def}}{=} \begin{cases} d & \text{if } x = y \\ int(\log_2(Proximity(x,y))) & \text{otherwise} \end{cases} \end{aligned} \tag{7}$$

In practice, $PO(x, y)$ is the length of the longest common prefix in the big-endian binary representation of x and y. It is therefore most practical to calculate by counting the matching bits of the binary form of the addresses from the left. Hence, Kademlia falls into the prefix matching class of DHT-s [Rowstron and Druschel, 2001, Zhao et al., 2004]. The maximum possible value of proximity is the total number of bits d.

Taking the proximity order relative to a fixed point x_0 partitions all points in the space, consisting of all possible bit sequences of length d into equivalence classes. Points in each class are at most half as distant from x_0 as items in the previous class. Furthermore, any two points belonging to the same class are at most half as distant from each other as they are from x_0. We can generalise the important properties of the proximity order function as follows:

Definition 109 Proximity order definitional properties**.**

$$PO : \mathcal{K} \times \mathcal{K} \mapsto \overline{0, d} \tag{8}$$

$$\text{reflexivity: } \forall x, y \in \mathcal{K}, PO(x, y) = PO(y, x) \tag{9a}$$
$$\text{monotonicity: } \forall x, y, z \in \mathcal{K}, PO(x, y) = k \wedge PO(x, z) = k \Rightarrow PO(y, z) > k \tag{9b}$$
$$\text{transitivity: } \forall x, y, z \in \mathcal{K}, PO(x, y) < PO(y, z) \Rightarrow PO(x, z) = PO(x, y) \tag{9c}$$

Given a set of points uniformly distributed in the space (e.g. the results of a hash function), proximity orders 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.

A.2 PROXIMITY ORDERS IN GRAPH TOPOLOGY

This section outlines a graph theoretical approach to Kademlia topology [Aspnes and Shah, 2007]. Consider a set of points V in the space with a logarithmic distance and a binary relation R. Define a directed graph where every point in the set is a vertex and any two vertices x and y are connected with an edge if they are in relation R.

Points in relation R with a particular point x_0 can be indexed by their proximity order relative to x_0. This index is called Kademlia table. The Kademlia table can then serve as the basis for local decisions in graph traversal where the task is to find a path between two points.

Definition 110 Kademlia table**.**

$$\begin{aligned} &Kad_x : \overline{0, d} \mapsto \mathcal{P}(V) \\ &\forall y, \langle x, y \rangle \in Edges(G) \rightarrow y \in Kad_x(p) \iff PO(x, y) = p \end{aligned} \tag{10}$$

We say that a point has *Kademlia connectivity* (or the point's connectivity has the Kademlia property) 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.

Definition 111 Kademlia connectivity**.**

$$\exists d, 0 \leq d \leq l, \text{such that} (1) \forall p < d, |Kad_x(p)| > 0 (2) \forall y \in V, PO(x,y) = p \geq d \longrightarrow y \in Kad_x(p) \tag{12}$$

If each point of a connected subgraph has Kademlia connectivity, then we say the subgraph has a Kademlia topology. In a graph with Kademlia topology, (1) a path between any two arbitrary points must exist, (2) this path can be discovered using decisions based on only information local to each hop and (3) is guaranteed to terminate in no more steps than the depth of the destination from the start point plus one.

The procedure is as follows. Given point x and y, construct the path $x_0 = x, x_1, ..., x_k = y$ such that $x_{i+1} \in PO(x_i, x_{i+1}) = PO(x_i, y)$, i.e. starting from the source point, choose the next point from the available options in the PO bin that the destination address falls into. Because of the first assumption of Kademlia connectivity as defined above, such a point must exist. Due to another previous conjecture, the actual bin must be monotonically increasing at each step and is able to start with zero. Therefore for every $0 < i \leq l$, $PO(x_i, y) \geq i$. Hence, since there exists a $d \leq d_y$, such that $PO(x_d, y) \geq d_y$. This, together with the second Kademlia connectivity criterion above, entails that either $x_d = y$ and $k = d$ or y is connected to x_d, and so we can choose $x_{d+1} = y$, and $k = d + 1 \leq d_y + 1$.

Theorem 1. In graphs with Kademlia topology, any two points are connected and traversal path can be constructed based on local Kademlia tables where the path length upper bound is logarithmic in the number of nodes.

Proof.

A.3 CONSTRUCTING KADEMLIA TOPOLOGY

A.4 COMPLEXITY OF FILLING UP A POSTAGE STAMP BATCH

A postage batch is utilised optimally when each of its collision slots are filled. Although hashes are uniform, due to variance one cannot guarantee that 2^d independent hashes will always map to the collision slots of a batch of depth d. We have presented various ways of mining chunks: by (1) choosing the nonce of a trojan chunk (see 4.4.1), (2)

choosing the encryption key (see 3.3.1), or (3) choosing the index of a single owner chunk (see 4.4.3). Whichever way we are mining the chunks, we may calculate in the same manner the chance of finding an address that takes the i-th collision slot of the stamp, i.e.

$$PO(H(Enc(k,c)),CS(i)) \geq D, \text{ where } CS(i) \stackrel{\text{def}}{=} i * 2^{256-D} \text{ for } 0 \leq i \leq 2^D. \tag{13}$$

If a user wishes to always fill their postage stamp batch before using another one, for each new chunk there is one less choice to pick a free collision slot. Let the probabilistic variable X_i^n denote the number of trials one needed to find a slot when i out of n slots are free. The probability that we will require at least 1 trial is $\frac{n-i}{n}$, and in general:

$$P(X_i^n \geq j+1 | X_i^n \geq j) = \frac{n-i}{n} \tag{14}$$

Therefore, the expected number of trials for the i-th chunk is:

$$E(X_i^n) = \sum_{j=1}^{\infty} jP(X_i^n = j) \tag{15a}$$
$$= P(X_i^n \geq 1) - P(X_i^n \geq 2) + 2(P(X_i^n \geq 2) - P(X_i^n \geq 3)) + \dots \tag{15b}$$
$$= P(X_i^n \geq 1) + P(X_i^n \geq 2) + \dots \tag{15c}$$

Multiplying both sides with $\frac{n-i}{n}$, then using the equivalence defined in 14 we get:

$$E(X_i^n)\frac{n-i}{n} = P(X_i^n \geq 1)P(X_i^n \geq 2 | X_i^n \geq 1) + P(X_i^n \geq 2)P(X_i^n \geq 3 | X_i^n \geq 2) + \dots \tag{16a}$$
$$= P(X_i^n \geq 2) + P(X_i^n \geq 3) + \dots \tag{16b}$$
$$= E(X_i^n) - P(X_i^n \geq 1) \tag{16c}$$
$$= E(X_i^n) - \frac{n-i}{n} \tag{16d}$$

From this, we solve to $E(X_i^n)$:

$$E(X_i^n)(1 - \frac{n-i}{n}) = 1 \tag{17a}$$
$$E(X_i^n) = \frac{n}{i} \tag{17b}$$

Averaging for the whole batch ($n = 2^D$) gives[1]

$$\sum_{i=1}^{2^D} \frac{2^D}{i} \frac{1}{2^D} = \sum_{i=1}^{2^D} \frac{1}{i} \tag{18a}$$

$$= ln(2^D) + 1 \tag{18b}$$

$$= \frac{log_2(2^D)}{log_2(e)} + 1 \tag{18c}$$

$$= D * 0.6931 + 1 \tag{18d}$$

A.5 AVERAGE HOP COUNT FOR CHUNK RETRIEVAL

This section gives a formal estimation of the expected hop count for a random non-cached chunk. Hop-count in traditional kademlia has already been analysed [Roos et al., 2013, Roos et al., 2015].

A.6 DISTRIBUTION OF REQUESTS

In this section we derive the distribution of requests per PO bin that an average node experiences.

A.7 EPOCH-BASED FEEDS

Update example

Let's say that we want to update our resource 5 minutes later. The Unix Time is now 1534093015. We calculate $epochBaseTime(1534093015, 25) = 1509949440$. This results in the same epoch as before $\langle 1534093015, 25 \rangle$. Therefore, we decrease the level and calculate again: $epochBaseTime(1534093015, 24) = 1526726656$. Thus, the next update will be located at $\langle 1526726656, 24 \rangle$

If the publisher keeps updating the resource exactly every 5 minutes, the epoch grid will look like this:

1 see harmonic series https://en.wikipedia.org/wiki/Harmonic_series_(mathematics)#Rate_of_divergence.

update	timestamp	epoch representation	update	timestamp	epoch representation
1	1534092715	$\langle 1509949440, 25\rangle$	16	1534097215	$\langle 1534096384, 10\rangle$
2	1534093015	$\langle 1526726656, 24\rangle$	17	1534097515	$\langle 1534096896, 9\rangle$
3	1534093315	$\langle 1526726656, 23\rangle$	18	1534097815	$\langle 1534097408, 11\rangle$
4	1534093615	$\langle 1530920960, 22\rangle$	19	1534098115	$\langle 1534097408, 10\rangle$
5	1534093915	$\langle 1533018112, 21\rangle$	20	1534098415	$\langle 1534097920, 9\rangle$
6	1534094215	$\langle 1534066688, 20\rangle$	21	1534098715	$\langle 1534098176, 8\rangle$
7	1534094515	$\langle 1534066688, 19\rangle$	22	1534099015	$\langle 1534098432, 10\rangle$
8	1534094815	$\langle 1534066688, 18\rangle$	23	1534099315	$\langle 1534098944, 9\rangle$
9	1534095115	$\langle 1534066688, 17\rangle$	24	1534099615	$\langle 1534099200, 8\rangle$
10	1534095415	$\langle 1534066688, 16\rangle$	25	1534099915	$\langle 1534099456, 15\rangle$
11	1534095715	$\langle 1534066688, 15\rangle$	26	1534100215	$\langle 1534099456, 14\rangle$
12	1534096015	$\langle 1534083072, 14\rangle$	27	1534100515	$\langle 1534099456, 13\rangle$
13	1534096315	$\langle 1534091264, 13\rangle$	28	1534100815	$\langle 1534099456, 12\rangle$
14	1534096615	$\langle 1534095360, 12\rangle$	29	1534101115	$\langle 1534099456, 11\rangle$
15	1534096915	$\langle 1534095360, 11\rangle$	30	1534101415	$\langle 1534100480, 10\rangle$

If the publisher keeps updating every 5 minutes ($300s$), we can expect the updates to stay around level 8-9 ($2^8 = 256s$, $2^9 = 512s$). The publisher can however, at any time vary this update frequency or just update randomly. This does not affect the algorithm.

Parallel algorithm to look up the latest update

This section gives a walkthrough about the steps of the algorithm that finds the latest update of a epoch-based feed (see 4.3.4).

The lookahead area indicates the area that will continue to be explored if R1 succeeds, while the lookback area highlights what will be continued to be explored if R1 fails to find an update at their location. After a short interval (configurable parameter head start) waiting for R1 to resolve, the algorithm sets out to explore both lookahead and lookback areas, headed by $\langle 12, 2\rangle$ and $\langle 4, 2\rangle$ respectively. These two simultaneous lookups are labeled R2 in the figure below, and are active together with R1. Also, recursively, 4 more lookup areas (labeled LA2 and LB2) are defined that depend on the result of each instance of R2: Once R1 resolves we can prune one area or the other. This is implemented by recursively cancelling a context, which aborts all chunk retrievals associated with it. The found update U9 can then be used as a hint for future lookups

In our example, R1 returns update U6, therefore our status is as below. Note that U6 is now marked as "known" in yellow, thus we're certain that the area in dark red, while it could contain other updates, **does not contain the latest one**, which is the one we care about:

Again, after a short head start, the algorithm proceeds to look up the lookback and lookahead headers $\langle 8, 2\rangle$ and $\langle 14, 1\rangle$, marked as active (purple), with the label R3

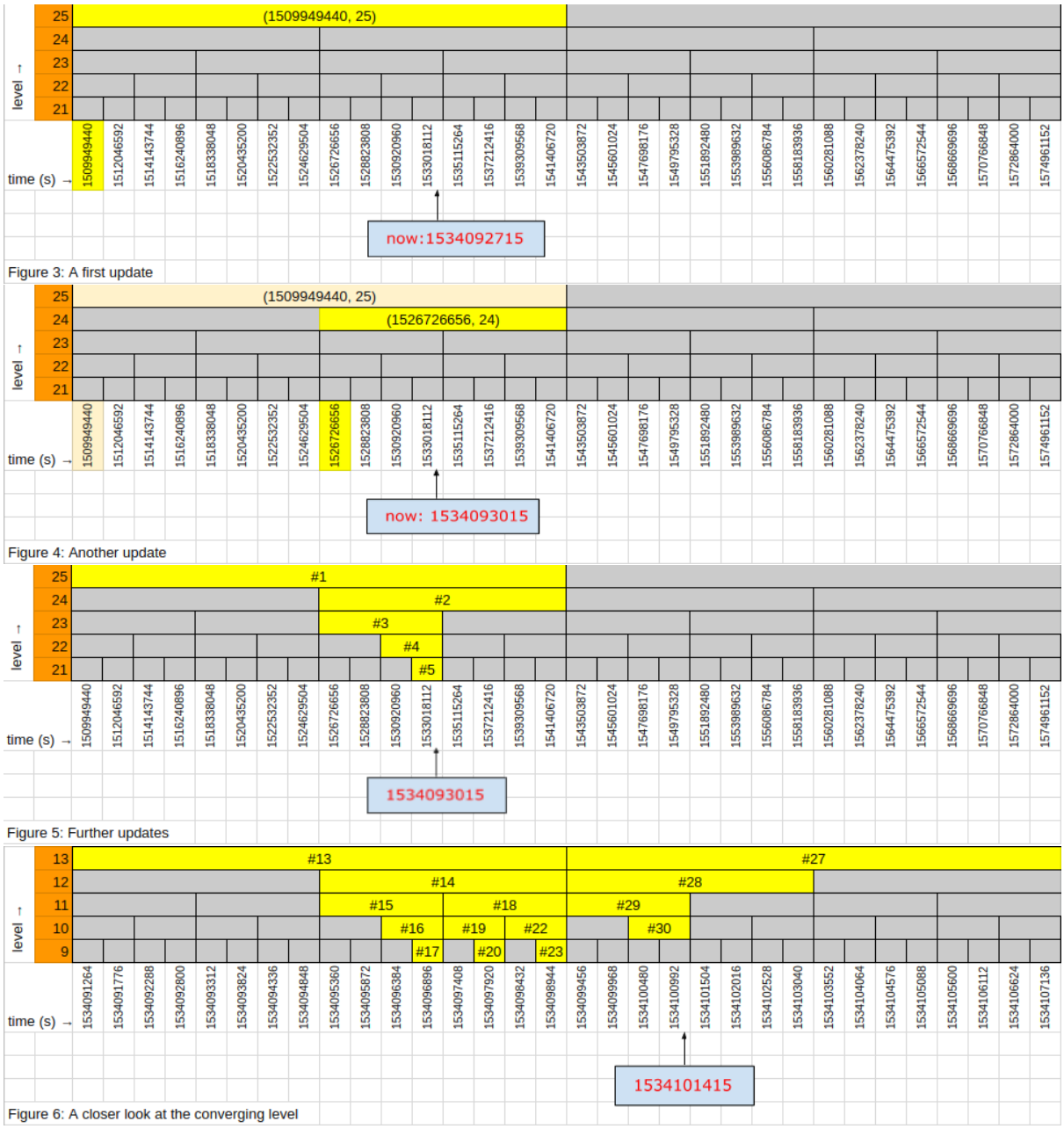


Figure 72: Updates of epoch-based feed in the epoch grid. Epochs occupied are marked in yellow.

Figure 73: Latest update lookup algorithm for time based feeds: steps. Known updates (U1) are marked in yellow. The hint is light orange (U2). Updates marked in grey (U3-U9) are unknown. $t = 14$ indicates the current time, Step 1: lookahead (LA1) area (blue) and lookback (LB1) area (pink);

below. These come, recursively, with their own lookahead and lookback areas, labeled LA3 and LB3:

In our example, R2 finally resolves, finding update U8. This means we can cancel lookback area LB2 as we are now sure the last update can't be there:

To shorten up the example, if R3 resolved immediately with no update found ($\langle 14, 1\rangle$ did not contain an update), that would cancel LA3 family of lookups, while LB3 continues (now marked as R4):

Although not drawn above, it is easy to see the pattern: LA4 and LB5 would then be scanned as R5 and R6 respectively, and in this case, fail, thus leaving $\langle 12, 1\rangle$ (U9), as the found update:

.

This example is simplified – with the current algorithm parameters, there can potentially be around 30 lookups taking place concurrently, exploring the search space for updates and pruning entire branches recursively once a better path is found. This can be configured by adjusting the lookup timeout and headstart times.

If the algorithm failed to find an update (i.e. if U3-U9 did not actually exist), then the hint in $\langle 0, 3\rangle$ (U2) would be challenged for validity: In this situation if the hint actually contains an update, that is then returned the last update and we're done. If the hint, however, is proven false then the algorithm restarts without a hint at that point.

Part V

INDEXES

GLOSSARY

access control The selective restriction of access to a read a document or collection in Swarm.

access control trie A tree-like data structure containing access keys and other access information.

access key Symmetric key used for encryption of reference to encrypted data.

access key decryption key The key the publisher grants to a party in a multi-party selective access scenario, used to decrypt the global access key.

accessible chunk A chunk is accessible if a message is routable between the requester and the node that is closest to the chunk.

address book Kademlia table of a peer's known addresses.

addressed envelope A construct where the address of the single owner chunk is created before the chunk content is associated with it.

aligned incentives Rewarding / penalisation in such a way, that the actors are inclined towards the desired behavior.

anonymous retrieval Not disclosing the identity of the requestor node while retrieving a chunk.

anonymous uploads Uploading leveraging the forwarding Kademlia routing while keeping the identity of the uploader hidden.

area of responsibility The area of the overlay address space in the node's neighbourhood. A storer node is responsible for chunks belonging to this area.

authoritative version history A secure audit trail of the revisions of a mutable resource.

backwarding A method of delivering a response to a forwarded request, where the response simply follows the request route back to the originator.

balanced binary tree A binary tree in which subtrees of every node differ in height by at most 1.

batch A group of chunks referenced under an intermediate node.

batch proof of custody A canonically serialised ordered list of binary Merkle tree proofs.

bin ID A sequential counter per PO bin acting as an index of the chunks stored locally on a node.

binary Merkle tree A binary tree in which every leaf node is labelled with the cryptographic hash of a data block, and every non-leaf node is labelled with hash of the labels of its child nodes.

binary Merkle tree chunk The canonical content addressed chunk in Swarm.

binary Merkle tree hash The method used for calculating the address of the binary Merkle tree chunks.

BitTorrent A communication protocol for peer-to-peer file sharing used to distribute data and electronic files over the Internet.

blockchain An immutable list of blocks, where each next block contains a cryptographic hash of the previous block.

blockhash The hash of a block which itself forms part of a blockchain.

bzz account An account in Swarm, also see Swarm base account.

bzz network ID The ID of the Swarm network.

Cauchy-Reed-Solomon erasure code A systemic erasure code which when applied to data consisting of n chunks produces k extra 'parity' chunks in such a way that any n chunks out of the total $n + k$ are enough to reconstruct the original blob.

challenge User can submit a challenge when they attempt to retrieve insured content and fail to find a chunk.

cheque An off-chain method of payment where issuer signs a cheque specifying a beneficiary, a date and an amount, gives it to the recipient as a token of promise to pay at a later date.

chequebook contract Smart contract that allows the beneficiary to choose when payments are to be processed.

chunk A chunk is fixed sized data blob, the basic unit of storage in Swarm's DISC keyed by its address. There are both content addressed chunks and single owner chunks.

chunk span The length of data subsumed under an intermediate chunk.

chunk synchronisation The process in which a peer stores locally the chunks from an upstream peer.

chunk value The chunk value is determined by the price of the postage batch it is stamped with. It serves to determine the order of chunks when a node prioritises for garbage collection.

chunking Splitting of data into chunks, to be stored in Swarm.

collect-and-run attack Situation where a party would collect the funds for some promised work, but not actually do the work.

collective information Data produced by collective effort, such as data on public forums, reviews, votes, polls, wikis.

collision slot The collection of maximum length prefixes that any two chunks stamped with a postage batch are allowed to share. A stamped chunk occupies a collision slot.

content addressed chunk A chunk is content addressed if the chunk content determines the chunk address. The address usually represents a fingerprint or digest of the data using some hash function. The default content addressed chunk in Swarm uses the Binary Merkle Tree hash algorithm with Keccak256 base hash to determine its address.

data silo A collection of information in an organization that is isolated from and not accessible by other parts of the organization. In more general terms, the large datasets organisations often keep to themselves.

data slavery A situation where an individual is trapped in a situation where they do not control or receive sufficient remuneration for their personal data which is used for commercial purposes by companies.

decentralised network A network designed with no central nodes upon which the other nodes would need to depend on.

deep bin A bin that is relatively close to a particular node and hence contains a smaller part of the address space.

denial of service (DoS) Denying of access to services by flooding those services with illegitimate requests.

destination target A bit sequence that represents a neighbourhood in the address space. In the context of chunk mining, it refers to the prefix that the mined address should match.

devp2p A set of network protocols which form the Ethereum peer-to-peer network. Implemented as a set of programming libraries with the same name.

direct delivery Chunk delivery happens in one step via some lower level network protocol.

direct notification from publisher The process where a recipient is notified of a feed update by the publisher or other parties known to have it.

disconnect threshold The threshold for debt between peers that determines the value of the debt at which a peer in debt will be disconnected.

distributed hash table A distributed system that provides a lookup service with which any participating node can efficiently retrieve the value associated with a given key.

distributed immutable store for chunks Swarm's version of a distributed hash table for storing files. Swarm does not keep a list of where files are to be found, instead it actually stores pieces of the file directly on the node.

distributed storage A network of storage where information is stored on more than one node, possibly in replicated fashion.

distributed web application A client side web application that leverages the Web 3.0 technologies (e.g. Ethereum network) and does not rely on any central server.

double ratchet An industry standard key management solution providing forward secrecy, backward secrecy, immediate decryption and message loss resilience.

duplicate chunk We define a chunk as a duplicate (or seen) if and only if it is already found in the local store.

elliptic curve Diffie-Hellman A key agreement protocol that allows two parties, each in possession of an elliptic-curve public–private key pair, to establish a shared secret over an insecure channel.

encrypted reference Symmetric encryption of a swarm reference to access controlled content.

enode URL scheme A URL scheme under which an Ethereum node can be described.
entanglement code An error correction code optimized for bandwidth of repair.
epoch A concrete time period starting at a specific point in time with specific length.
epoch base time The specific point in time when an epoch starts.
epoch grid The arrangement of epochs where rows (referred to as levels) represent alternative partitioning of time into various disjoint epochs with the same length.
epoch reference A pair of epoch base time and level that identify a specific epoch.
epoch-based feeds Special kind of feeds which provide feeds with sporadic updates a way to be searched.
epoch-based indexing Indexing based on the epoch in which an action took place.
erasure code An error correction coding scheme which optimally inflates data of n chunks with k parities to allow any n out of the $n + k$ chunks to recover the original data.
Ethereum Name Service The system that, analogously to the DNS of the old web, translates human-readable names into system-specific identifiers, i.e. a reference in the case of Swarm.
Ethereum Virtual Machine (EVM) A Turing complete byte code interpreter that is responsible to calculate state changes by executing the instructions of smart contracts.
eventual consistency The guarantee that all chunks are redundantly retrievable after allowing time for the neighbourhood peers to sync their content.
extended triple Diffie–Hellmann key exchange The customary method of establishing the initial parameters of a double ratchet key-chain.

FAANG Facebook, Apple, Amazon, Netflix and Google.
fair data economy Economy of processing data characterised by fair compensation of all parties involved in its creation or enrichment.
feed aggregation The process of combining a set of sporadic feeds into a periodic one.
feed index Part of the identifier of the feed chunk.
feed topic Part of the identifier of the feed chunk.
feeds A data structure based on single owner chunks, suitable for representing a variety of sequential data, such as versioning updates of a mutable resource or indexing messages for real-time data exchange offering a system of persisted pull messaging.
- **partitions** Special kind of feeds, updates of which are meant to be accumulated or added to earlier ones, e.g. parts of a video stream.
- **periodic feeds** Feeds with updates published at regularly recurring intervals.
- **real-time feeds** Feeds where the update frequencies may not be regular but show variance within the temporal range of real-time human interaction.
- **series** Special kind of feeds representing a series of content connected by a common thread, theme or author such as status updates on social media, a person's blog posts or blocks of a blockchain.

sporadic feeds Feeds with irregular asynchronicities, i.e. updates can have unpredictable gaps.

fingerpointing Litigation scheme allowing upstream peers to shift blame to a downstream peer in the case of deferred responsibility.

forwarding Kademlia Recursive flavour of Kademlia routing involving message relay.

forwarding lag The time in which healthy nodes could forward messages.

freeriding The uncompensated depletion of limited resources.

future secrecy A feature of specific key agreement protocols that gives assurances that all other session keys will not be compromised even one or more session keys is obtained by an attacker.

garbage collection The process that selectively purges chunks from a node's local storage.

garbage collection strategy The process that dictates which chunks are chosen for garbage collection.

global balance The amount deposited in the chequebook to serve as collateral for the cheques.

granted access A type of selective access to encrypted content requiring root access as well as access credentials comprising either an authorized private key or passphrase.

guaranteed delivery Guaranteed in the sense that delivery failures due to network problems result in direct error response.

head start A configurable time interval that defines the waiting time before an exploration of an area for updates of epoch-based feeds.

hive protocol The protocol nodes joining the network use to discover their peers.

immutable chunk store No replace/update operation is available on chunks.

incentive strategy A strategy of rewarding and penalising behavior to encourage desired behaviour.

inclusion proofs A proof that a string is a substring of another string, for instance that a string is included in a chunk.

indexing scheme Defines the way the addresses of subsequent updates of a feed are calculated. The choice of indexing scheme depends on the type and usage characteristics (update frequency) of the feed.

insider A peer inside the Swarm that already has some funds.

InterPlanetary File System A protocol and peer-to-peer network for storing and sharing data in a distributed file system.

Kademlia A network connectivity or routing scheme based on bit prefix length used in distributed hash tables.

Kademlia connectivity Connectivity pattern of a node x in forwarding Kademlia where (1) there is at least one peer in each PO bins $0 <\leq i < d$, and (2) no peer y in the network such that $PO(x, y) \geq d$ and y is not connected to x.

Kademlia table Indexing of peers based on proximity order of peer address relative to local overlay address.

thin Kademlia table A Kademlia table in which a single peer is present for each bin (up to a certain bin).

Kademlia topology A scale free network topology that has guaranteed path between any two nodes in $O(log(n))$ hops.

key derivation function A function which deterministically produces keys from an initial seed, often used concurrently by parties separately to generate secure messaging key schemes.

libp2p A framework and suite of protocols for building peer-to-peer network applications.

light node Concept of light node refer to a special mode of operation necessitated by poor bandwidth environments, e.g., mobile devices on low throughput networks or devices allowing only transient or low-volume storage. Such nodes do not accept incoming connections.

litigation An on-chain process where nodes violating the rules of Swarm stand to lose their deposit.

load balancing The process of distributing a set of tasks over a set of nodes to make the process more efficient.

lookahead area An area that can be part of search of the algorithm that finds the latest update of a epoch-based feed.

lookback area An area that can be part of search of the algorithm that finds the latest update of a epoch-based feed.

lookup key One of the keys involved in the process of allowing selective access to content to multiple parties.

lookup strategy A strategy used for following updates to feeds.

manifest entry Contains a reference to the Swarm root chunk of the representation of a file and also specifies the media mime type of file.

mining chunks An example of chunk mining is to generate an encrypted variant of chunk content so that the resulting chunk address satisfies certain constraints, e.g. is closer to or farther away from a particular address.

missing chunk notification protocol A protocol upon which downloader not finding a chunk, can fall back to a recovery process and request the chunk from a pinner of that chunk.

mutable resource updates Feeds that represent revisions of the same semantic entity.

nearest neighbours Generally, peers that are closest to the node. In particular, those residing within the neighbourhood depth of each other.

neighbourhood An area of a certain distance around an address.

neighbourhood depth Distance from the node within which its peers are considered to be nearest neighbours.

neighbourhood notification Notification of a feed update which works without the issuer of the notification needing to know the identity of prospective posters.

neighbourhood size The number of nearest neighbours of a node.

net provider A node that is providing more resources to the Swarm network than it is using.

net user A node that is using more resources of the Swarm network than it is providing.

network churn The cycle of accumulation and attrition of nodes by a network.

newcomer A party entering the Swarm system with zero liquid funds.

node

forwarding node Nodes that engage in forwarding messages.

stable node Node that is stably online.

storer node A node storing the requested chunk.

on-chain payment Payment via a blockchain network.

opportunistic caching When a forwarding node receives a chunk, then the chunk is saved in case it will be requested again.

outbox feed A feed representing the outgoing messages of a persona.

outbox index key chains Additional key chains added to the double-ratchet key management (beside the ones for encryption) that make the feed update locations resilient to compromise.

overlay address The address with which each node running Swarm is identified. It is the basis for communication in the sense that it remains stable across sessions even if underlay address changes.

overlay address space Address space of the overlay Swarm network consisting of 256-bit integers.

overlay network Connectivity pattern of Swarm secondary conceptual network, a second network scheme overlayed over the base underlay network.

overlay topology The connectivity graph realising a particular topology over the underlay network.

payment threshold The value of debt at which a cheque is issued.

peer Nodes that are in relation to a particular node x are called peers of x.

downstream peer A peer that succeeds some other peer in the chain of forwarding.

peer-to-peer A network where tasks or workloads are partitioned between peers who are all equally privileged participants.

pinner A node keeping a persistent copy of a chunk.

pinning The mechanism that makes content sticky and prevents it from being garbage collected.

plausible deniability The ability to deny knowledge of any damnable actions committed by others.

postage batch A postage batch is an id associated with a verifiable payment on chain which can be attached to one or more chunks as a postage stamp.

postage lottery A scheme to provide compensation to storing nodes through redistribution of the revenue resulting from postage stamps among the registered storers in a fair way.

postage stamp Proof of payment for pre-paid delivery and storage.

pre-key bundle Consists of all information the initiator needs to know about the responder to initiate a cryptographic handshake.

prefix collision In the context of postage stamps, when the owner of a batch attaches a stamp to two chunks with a shared prefix longer than the depth of the batch.

prompt recovery of data The protocol of the missing chunk notification and recovery.

proximity order A measure of relatedness of two addresses on a discrete scale.

proximity order bin An equivalence class of peers in regard to their proximity order.

pub/sub systems A publish/subscribe system is a form of asynchronous communication where any message published is immediately received by subscribers.

pull syncing Network protocol responsible for eventual consistency and maximum resource utilisation by pulling chunks by a certain node.

push syncing Network protocol responsible for delivering a chunk to its proper storer after it is uploaded to an arbitrary node.

radius of responsibility The proximity order designating the area of responsibility.

raffle draw Instance of the postage lottery draw repeated every N blocks conducted by the postage lottery contract on the blockchain.

raffle–apply–claim–earn (race) The postage lottery protocol for incentivising non-promissory storage by redistributing postage stamp revenue to simple non-staked storers applying with proof of custody.

range queries Range queries will trigger the retrieval of all but only those chunks of the file that cover the desired range.

real-time integrity check For any deterministically indexed feed. Integrity translates to a non-forking or unique chain commitment.

recover security A property that ensures that once an adversary manages to forge a message from A to B, then no future message from A to B will be accepted by B.

recovery A process of requesting a missing chunk from specific recovery hosts.

recovery feed A publisher's feed advertising recovery targets to their consumers.

recovery host Pinning nodes that are willing to provide their pinned chunks in the context of recovery.

recovery request A request to a recovery host to initiate the reupload of a missing chunk we know it has pinned in its local store.

recovery response envelope An addressed envelope which provides a way for recovery hosts to respond directly to the originator of the recovery request promptly and without cost or computational burden.

recovery targets Volunteering nodes that are advertised by the publisher as keeping pinned its globally pinned publication.

redundancy In the context of the distributed chunk store, redundancy is provided by surplus replicas or so called parities that contribute to resilience of chunk storage in the face of churn and garbage collection.

redundant Kademlia connectivity A Kademlia connectivity where some peers might churn, yet Kademlia connectivity would still exist.

redundant retrievability A chunk is said to be redundantly retrievable with degree r if it is retrievable and would remain so after any r nodes responsible for it leave the network.

reference count A property of a chunk used to prevent it from being garbage collected. It is increased and decreased when the chunk is pinned and unpinned, respectively.

relaying node A node relaying messages in the context of forwarding Kademlia.

requestor node A node requesting some information from the network.

retrieve request Peer-to-peer protocol message asking for the delivery of a chunk based on the chunk address.

root access Non-privileged access to encrypted content based on meta-information encoded in the root manifest entry for a document.

root access manifest A special unencrypted manifest used as an entry point for access control.

routability The ability for a chunk to be routed to a destination.

routed delivery Hypothetical way of implementing chunk delivery using Kademlia routing independently of the initial request.

routing Relaying messages via a chain of peers ever closer to the destination.

saturated Kademlia table Nodes with a saturated Kademlia table realise Kademlia connectivity.

saturation depth The neighbourhood depth in the context of saturation (minimum cardinality) constraints on proximity bins outside the nearest neighbourhood.

second-layer payment Payments processed by an additional system superimposed on a blockchain network.

security deposit The stake a node must put up when registering to be able to sell promissory storage receipts.

seeder User hosting the content in the BitTorrent peer-to-peer file exchange protocol.

sender anonymity As requests are relayed from peer-to-peer, those further down on the request cascade can never know who the originator of the request is.

session key One of the keys involved in the process of allowing selective access to content to multiple parties.

shallow bin A bin that is relatively far away from a particular node and hence contains a larger part of the address space.

single owner chunk A special type of chunk in Swarm, the integrity of which is given by the association of its payload to an identifier attested by the signature of its owner. The identifier and the owner's account determines the chunk address.

identifier A 32-byte key used in single owner chunks: the payload is signed against it by the owner and hashed together with the owner's account results in the address.

owner Account of the owner of single owner chunk.

payload Part of single owner chunk with size of maximum 4096 bytes of regular chunk data.

singleton manifest A manifest that contains a single entry to a file.

span value An 8 byte encoding of the length of the data span subsumed under an intermediate chunk.

spurious hop Relaying traffic to a node without increasing proximity to the target address.

stamped addressed envelope Addressed envelope with an attached stamp.

statement of custody receipt A receipt from the storer node to the uploader after successful push syncing of a chunk.

stream provider To provide of a stream of chunks to another node on request.

swap swap is a Swarm accounting protocol with a tit-for-tat accounting scheme that scales microtransactions. Includes a network protocol also called Swap.

Swarm base account The Ethereum account the Swarm node is associated with. See also bzz account.

Swarm manifest A structure that defines a mapping between arbitrary paths and files to handle collections.

swear Incentive scheme where nodes registered on the Swarm network are accountable and stand to lose their deposit if they are found to violate the rules of the Swarm in an on-chain litigation process.

swindle Incentive scheme where nodes monitor other nodes to check if they comply with their promise by submitting challenges according to a process of litigation.

tar stream In computing, tar is a computer software utility for collecting many files into one archive file, often referred to as a tarball.

targeted chunk delivery Mechanism to request a chunk from an arbitrary neighbourhood where it is known to be stored to an arbitrary neighbourhood where it is known to be needed.

time to live The lifespan or lifetime of a request or other message in a computer or network.

tragedy of commons A situation where, in a shared-resource system, individual users act in their own interest contrary to common good and deplete or spoil the shared resource through collective action.

tragedy of the commons Disappearing content will have no negative consequence to any one storer node if no negative incentives are used.

Trojan chunk A chunk containing a disguised message while appearing no different to other chunks.

trustless property of a system of economic interaction where service provision is either realtime verifiable and/or providers are accountable, reward and penalties are

automatically enforced, and where – as a result – transaction security is no longer contingent upon reputation or trust and is therefore scalable.

underlay address The address of a Swarm node on the underlay network. It might not remain stable between sessions.

underlay network The lowest level base network by which nodes connect using a peer-to-peer network protocol as their transport layer.

uniformity requirement A constraint on postage batches that chunks signed with same batch identifier must not have a common prefix longer than the depth.

upload and disappear A way to deploy your interactive dynamic content to be stored in the cloud so it may be retrieved even if the uploader goes offline.

upload tag An object representing an upload and tracking the progress by counting how many chunks reached a particular state.

uploader An entity uploading content to the Swarm network.

upstream peer The peer preceding some other peer in the chain of forwarding.

value-consistent garbage collection strategy Strategy of garbage collection where a minimum postage stamp value accepted on a chunk coincides with the garbage collection cutoff value.

witness Digital signature issued by the entity that the payer of a postage stamp designates.

witness batch Serves as a spot check for the applicant's claim that they store all chunks they are responsible for. The witness batch is random choice of the valid postage batches; the applicant must have actually stored all chunks in their neighbourhood.

world computer Global infrastructure that supports data storage, transfer and processing.

World Wide Web A part of the Internet where documents and other web resources are identified by Uniform Resource Locators and interlinked by hypertext.

- **Web 1.0** Websites where people were limited to viewing content in a passive manner.
- **Web 2.0** Describes websites that emphasise user-generated content, ease of use, participatory culture and complex user interfaces for end users.
- **Web 3.0** A decentralised, censorship-resistant way of sharing and even collectively creating interactive content, while retaining full control over it.

ZeroNet Decentralized web platform using Bitcoin cryptography and BitTorrent network.

INDEX

F

G

H

I

K

L

M

N

O

P

R

S

LIST OF ACRONYMS AND ABBREVIATIONS

AC access control.
ACT access control trie.
API application programming interface.

BMT chunk binary Merkle tree chunk.
BMT hash binary Merkle tree hash.
BMT proof binary Merkle tree proof.
BMT binary Merkle tree.

CRS Cauchy-Reed-Solomon erasure code.

dapp distributed web application.
DHT distributed hash table.
DISC distributed immutable store for chunks.

ECDH elliptic curve Diffie-Hellman.
ENS Ethereum Name Service.
EVM Ethereum Virtual Machine (EVM).

HTTP Hypertext Transfer Protocol.

IPFS InterPlanetary File System.
ISP internet service provider.

MAC message authentication code.

P2P peer-to-peer.
PO proximity order.
prod prompt recovery of data.

race raffle–apply–claim–earn (race).
RAID Redundant Array of Inexpensive Disks.

SWAP service wanted and provided ALSO settle with automated payments ALSO send waiver as payment ALSO start without a penny, see swap.
SWEAR Secure Ways of Ensuring ARchival or Swarm Enforcement And Registration, see swear.
SWINDLE Secured With INsurance Deposit Litigation and Escrow, see swindle.

TLD top level domain.
TTL time to live.

WWW World Wide Web.

X3DH extended triple Diffie–Hellmann key exchange.