THE BOOK OF SWARM

STORAGE AND COMMUNICATION INFRASTRUCTURE FOR SELF-SOVEREIGN DIGITAL SOCIETY

BACK-END STACK FOR THE DECENTRALISED WEB

Viktor Trón

v2.0 pre-release 1.0

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 present 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 (**??**), provides the formal specification of components. This is meant to serve as the reference handbook for Swarm client developers.

The index, glossary for technical terms and acronyms, and an appendix complete the compendium.

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 use the book as background reading for the specs whenever wider context is needed or one is interested in the justification for the choices.

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) as well as Attila Lendvai 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 Gazsó 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 Szabó, Vlad Gluhovsky, Rafael Matiasf 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 Gazsó 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 Inter-Planetary 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 back-door 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* and equipped with robust guarantees for long-term continuity.

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

- *stable* - The specifications and software implementations are stable and resilient to changes in participation, or politics (political pressure, censorship).

- *scalable* - The solution is able to accommodate many orders of magnitude more users and data as it scales, without adversely impacting performance or reliability during mass adoption.

- *secure* - The solution is resistant to deliberate attacks, remains impervious to social pressure and political influences, and demonstrates fault tolerance 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 datascene: 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 violated.

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 provides the necessary responsiveness for dapps running on users' devices, while also offering incentivised storage utilizing various storage infrastructures ranging from smartphones to high-availability clusters. Continuity will be guaranteed with well-designed incentives for bandwidth and storage.

Content creators will receive fair compensation for the content they offer, and content consumers will be paying for it. By eliminating the middlemen providers who 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 and every device leaves a trail of data, which is collected and stored in silos, whose potential is used up only in part and to the benefit of large players.

Swarm will serve as the go-to platform for digital mirror worlds, providing individuals, societies, and nations with a cloud storage solution that is independent of any one large provider.

Individuals will have full control over their own data. They will no longer need to be part of the data slavery, where personal data is exchanged for services. Moreover, they will be able to form data collectives or data co-operatives, sharing certain kinds of data as commons to achieve shared objectives.

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. These clouds will operate in a peer-to-peer manner, potentially within exclusive regions, and third parties will not be able to interfere in the flow of data and communication to monitor, censor, or manipulate it. However, authorized parties will have access to the data, aiming to level the playing field for AI and services based on it.

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

In the future society, Swarm will become ubiquitous, transparently and securely serving data from individuals and devices to data consumers within 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).

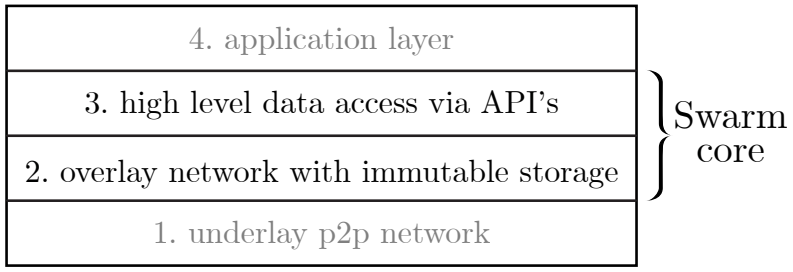

Figure 1: Swarm's layered design

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 elaborates 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 **??**.[1]

1 Swarm's initial 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 that 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 the Ethereum 1.x blockchain and for storing its state on Swarm, we may provide a thin devp2p node that proxies queries to the new libp2p-based Swarm client, or just uses its API. Otherwise we expect the devp2p networking support to be discontinued.

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 **??**). 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. It is important to derive the address from a public key as it allows the nodes to issue commitments associated with an overlay location using cryptographic signatures that 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 **??**. How nodes use this protocol to bootstrap the overlay topology is discussed in 2.1.4.

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 Proximity order is the discrete logarithmic scale of proximity, which, in turn is the inverse of normalised XOR distance.

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. 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).

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.

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

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

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, 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. Thus, routing using this approach, simply means 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 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 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 are 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 **??**) 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 managed to forward it.[4]

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 peers further down in the request cascade can never know who the originator of the request was, because requests are relayed from peer-to-peer.

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, i.e. clients that due to resource constraints do not keep a full Kademlia saturation but instead have just a local neighbourhood, then 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. This means that unless the number of required connections in a bin doubles as bins go farther from the node, shallower bins will always allow more choice of nodes for potential connection. In particular, nodes have a chance to increase the number of connections per bin in

4 Healthy nodes could commit to being able to forward within a (very short) constant time; let's call this 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.

such a way that peer addresses maximise density (i.e., in proximity order bin b, the subsequent bits of peer 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 physically closer to each other 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 **??** 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 that each node must follow to reach and maintain a saturated Kademlia connectivity. 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 an initial step that helps naive nodes to begin exchanging information about each other. This discovery process is called the hive protocol and is formally specified in **??**.

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, electing one particular node as an initial connection should be avoided, and the role of being a bootnode for the newly connecting naive nodes should ideally be distributed among participant nodes. This is achieved either with an invite system, or a centralised bootnode service running a public gateway that 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 kicks in and the naive node begins to learn about the bzz addresses of other nodes, and thus it 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 about 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 y connects to x, x notifies a subset of its connected peers. A peer p belongs to this subset if $PO(x,p) = PO(x,y)$ or $d_p \leq PO(y,p)$. The notification takes the form of a protocol message and includes the full overlay address and underlay address information (see **??**).[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 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.

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.

6 Note that the node does not need to know the total 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 to the power of 2.

In this section, in 2.2.1, 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, 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 first line of defense against node churn.

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 a 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*.

From DHT to DISC

Swarm's DISC shares many similarities with the wider known distributed hash tables. The most important difference is that Swarm does not keep track 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.

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.

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 onto 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.

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).

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 that serve as the basis of legal protection for node operators. They need to be able 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 **??**).

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, fair and equal load balancing requires that the addresses of chunks should also be uniformly distributed in the address space, and have their content limited and roughly uniform in size.

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, a 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.

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 **??**. 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.

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 3.4). Together with the Swarm file hash (see 4.1.1 and **??**), it allows for logarithmic inclusion proofs for files, i.e., proof that a string is found to be part of a file.

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.

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).

Chunk encryption (see figure 12) is formally specified in **??**. Chunks shorter than 4 kilobytes are padded with random bytes (generated from the chunk encryption 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 containing 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 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 **??**). 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). 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 3.3.4).

2.3 PUSH AND PULL: CHUNK RETRIEVAL AND SYNCING

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 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.

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.

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.

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 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 3.2.6).

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 **??**).

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 feedback of the successful propagation of their data across the network (see 6.1 and **??**).

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 3.2.6).

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 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 **??**). 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 **??**). 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 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.

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

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.

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 and 3.3.4 and a consolidated client strategy is specified in **??**.

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 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 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.

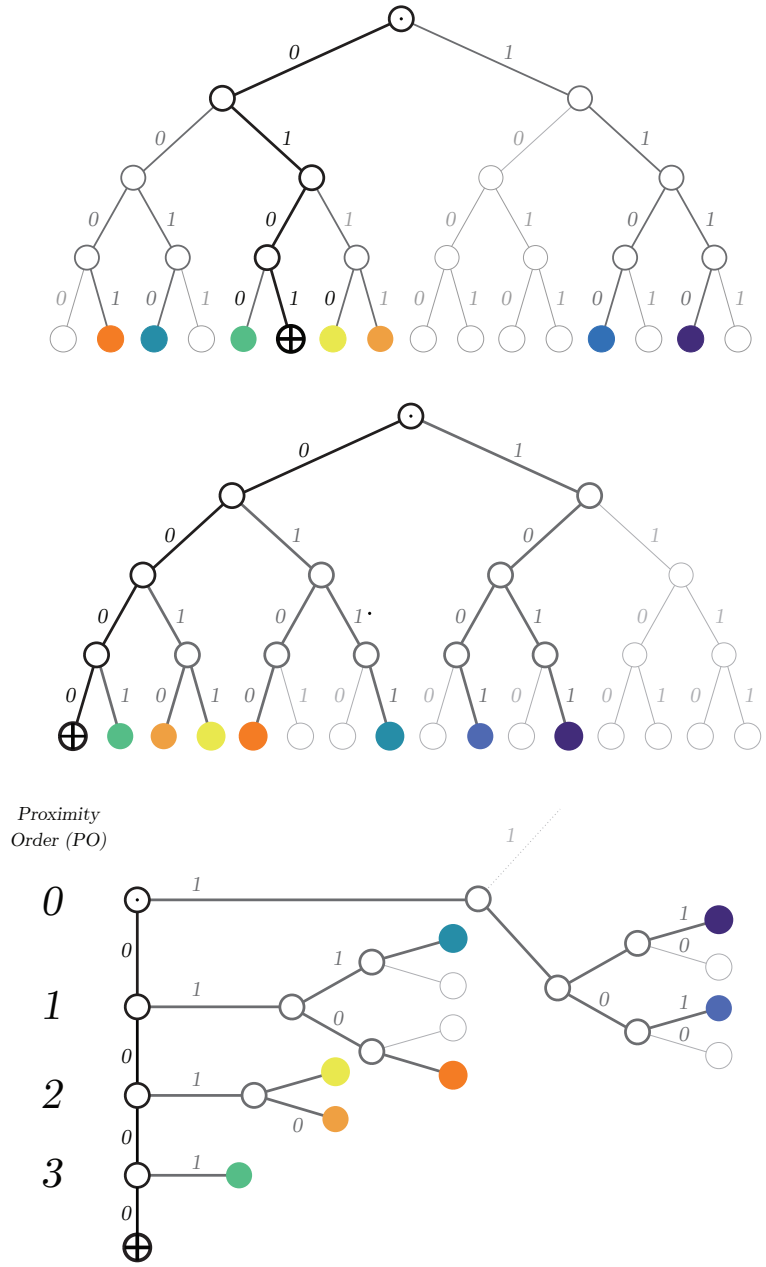

Figure 2: From overlay address space to Kademlia table. **Top**: the overlay address space is represented with a binary tree, colored leaves are actual nodes. The path of the pivot node (+) is shown with thicker lines. **Centre**: peers of the pivot nodes are shown keyed by the bits of their xor distance 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.

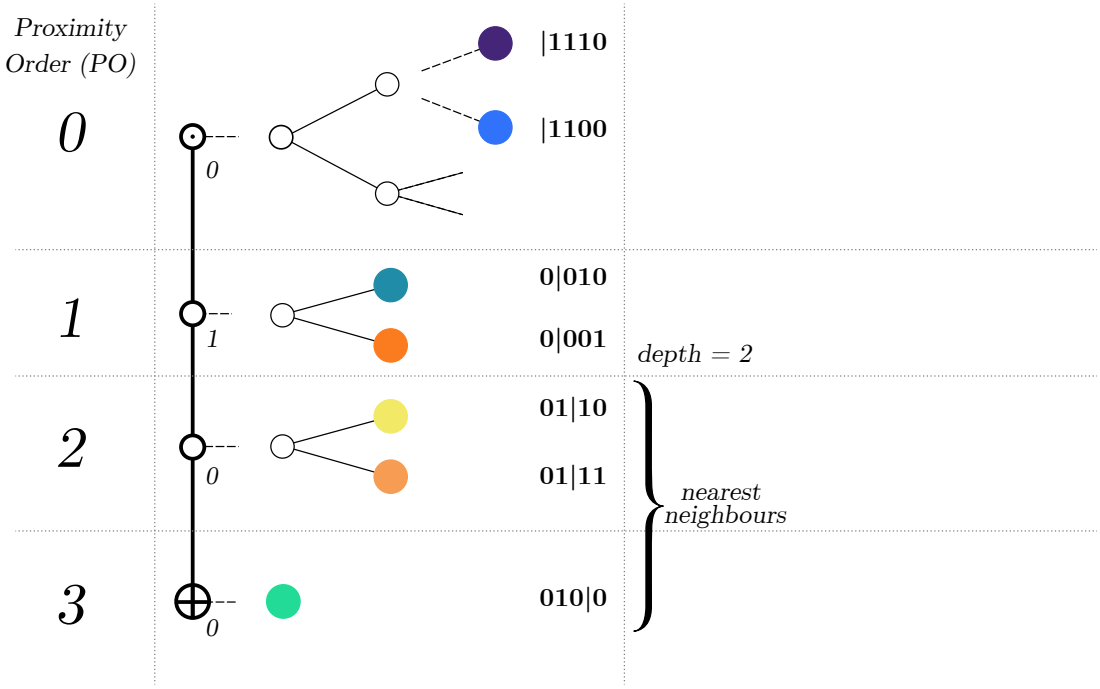

Figure 3: Nearest neighbours in a 4 bit network with $d = 2$

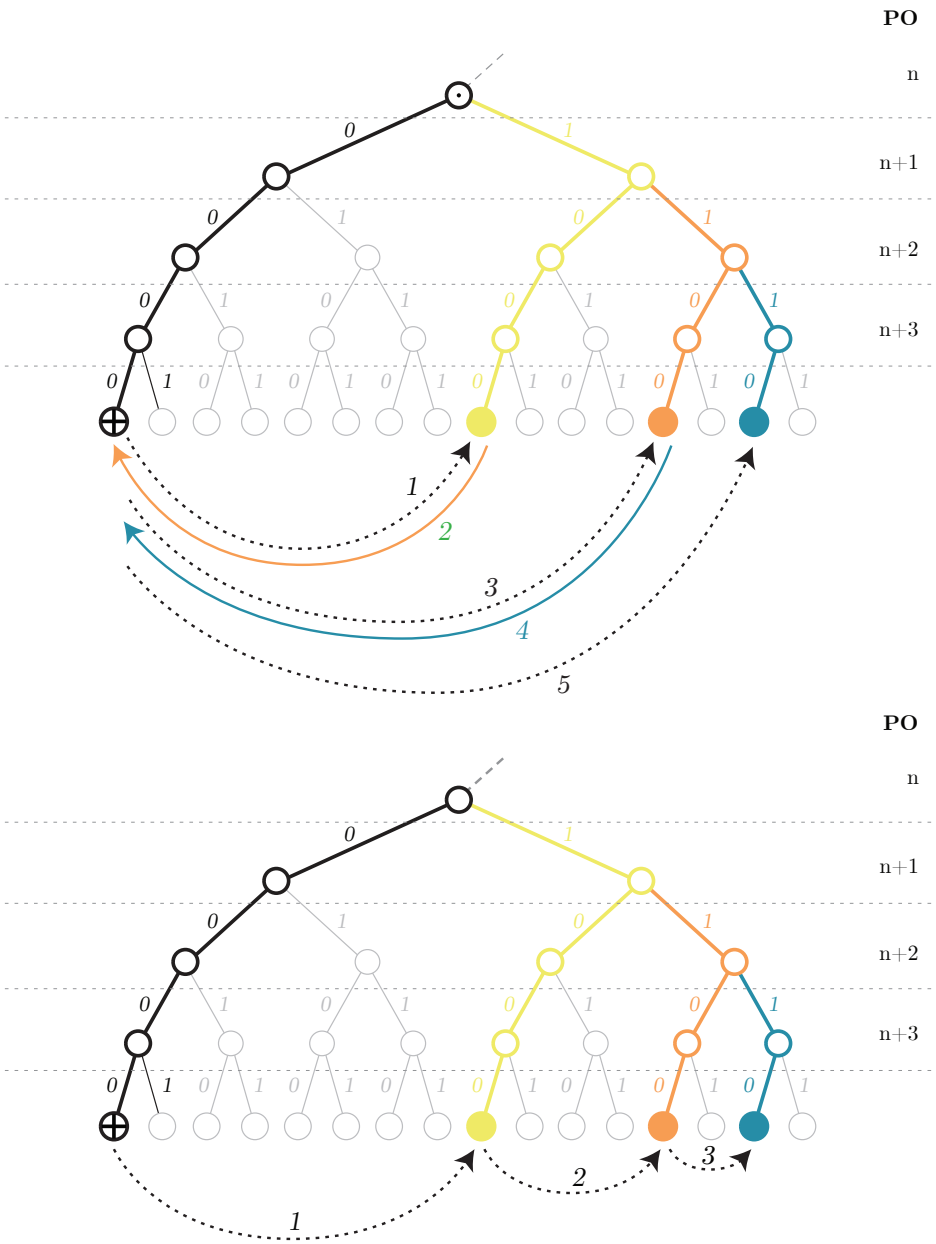

Figure 4: Iterative and Forwarding Kademlia routing: A requestor node shown with a cross in the circle at address ...0000... wants to route to a destination address ...1111... to which the closest peer online is the blue circle at ...1110... These initial ellipses represent the prefix shared by requestor 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.

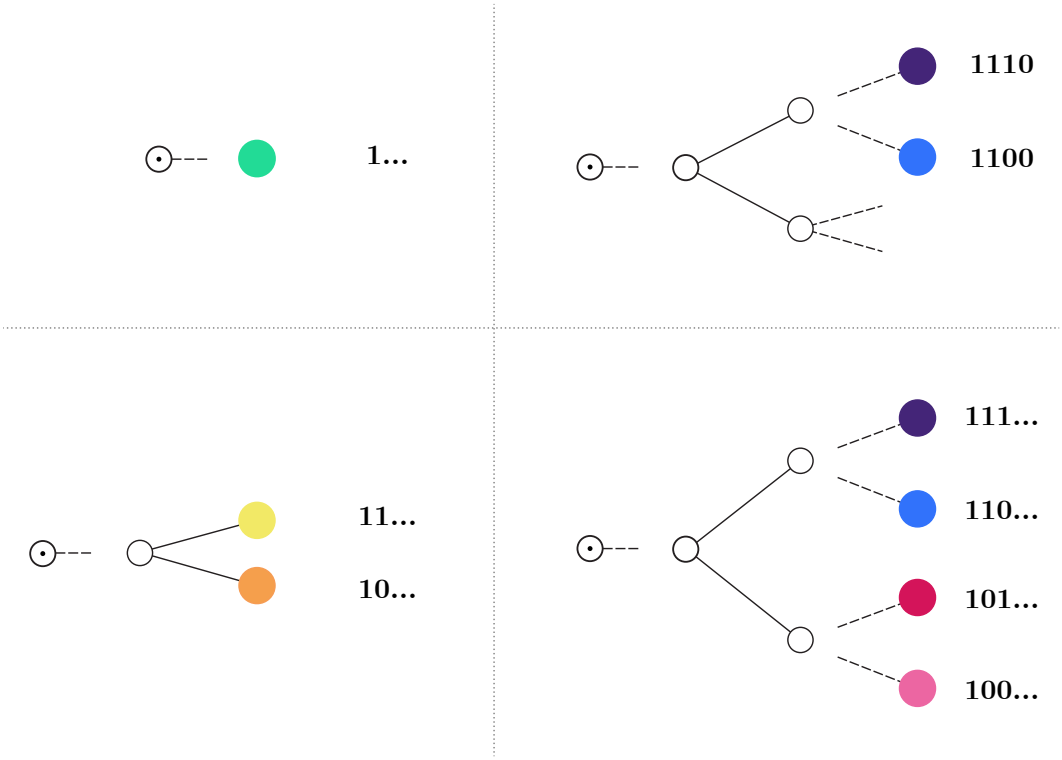

Figure 5: Bin density: types of saturation for PO bin 0 for a node with overlay address starting with bit 0. **Top left**: A "thin" bin with a single peer is not resilient to churn and only increases PO by 1 in one hop. **Top right:** At least two peers are needed to maintain Kademlia topology in case 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.

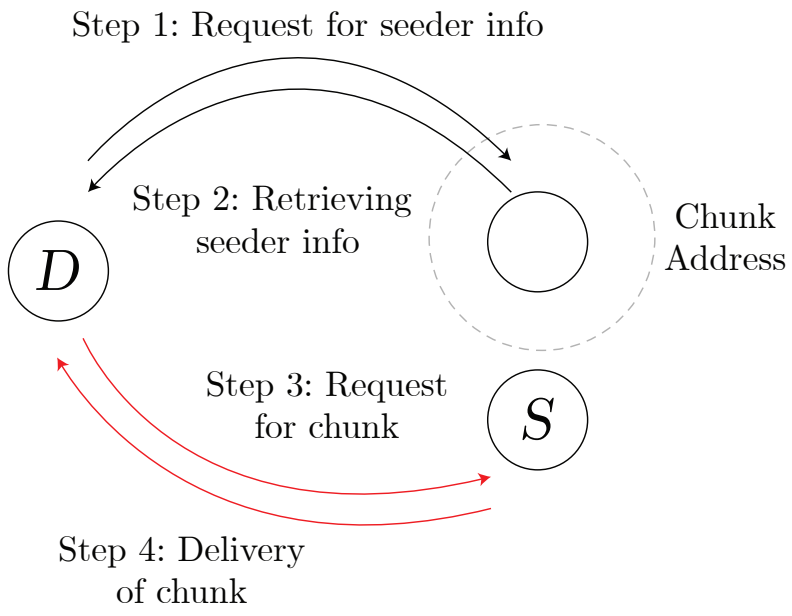

Figure 6: Distributed hash tables (DHTs) used for storage: node *D* (downloader) uses Kademlia routing in step 1 to query nodes in the neighbourhood of the chunk address to retrieve seeder info in step 2. The seeder info is used to contact node *S* (seeder) directly to request the chunk and deliver it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

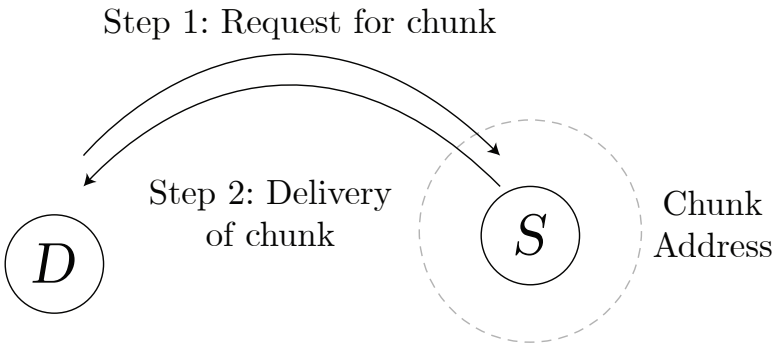

Figure 7: Swarm DISC: Distributed Immutable Store for Chunks. In step 1, downloader node *D* uses forwarding Kademlia routing to request the chunk from a storer node *S* in the neighbourhood of the chunk address. In step 2 the chunk is delivered along the same route using forwarding step just backwards.

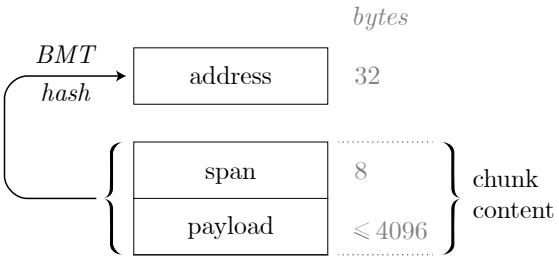

Figure 8: Content addressed chunk. An at most 4KB payload with a 64-bit little endian encoded span prepended to it constitutes the chunk content. The BMT hash of the payload concatenated with the span then yields the content address.

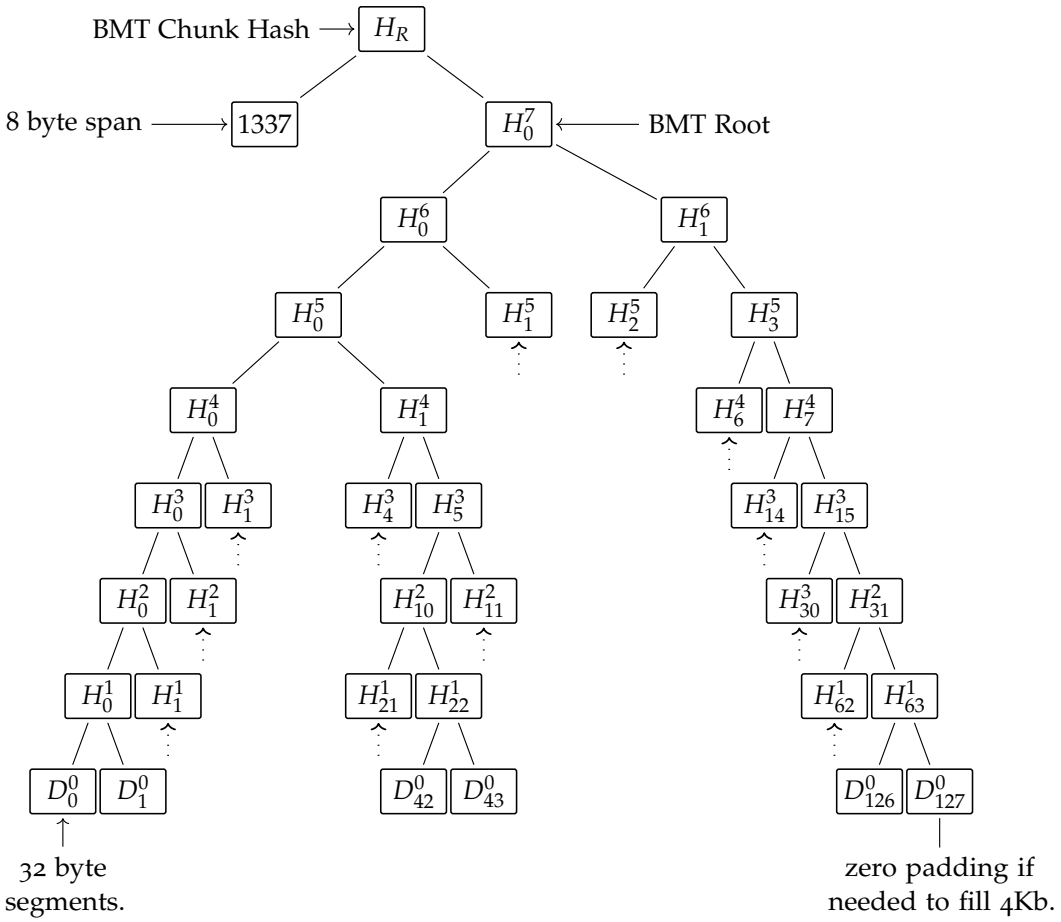

Figure 9: BMT (Binary Merkle Tree) chunk hash in Swarm: the 1337 bytes of chunk data is segmented into 32 byte segments. Zero padding is used to fill up the rest up to 4 kilobytes. Pairs of segments are hashed together using Keccak256 to build up the binary tree. On level 8, the binary Merkle root is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

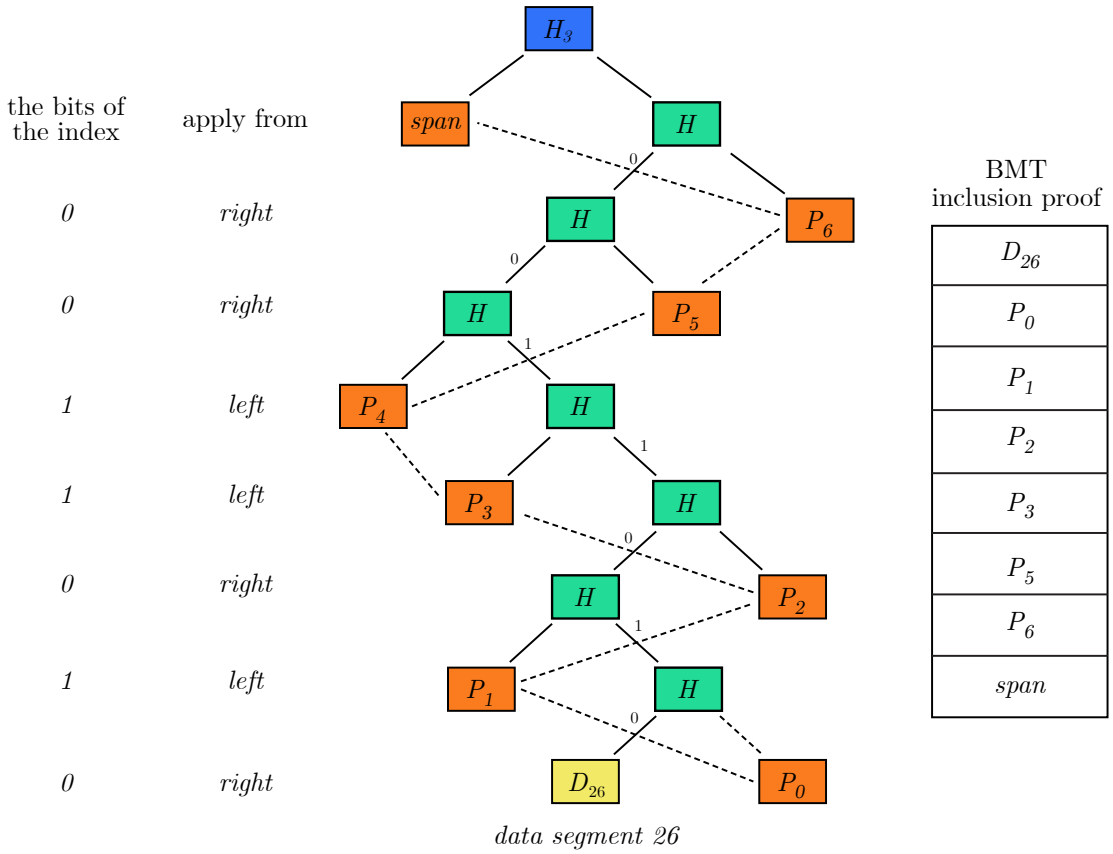

Figure 10: Compact segment inclusion proofs for chunks. Assume we need proof for segment 26 of a chunk (yellow). The orange hashes of the BMT are the sister nodes on the path from the data segment up to the root and constitute what needs to be part of a proof. When these are provided together with the 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.

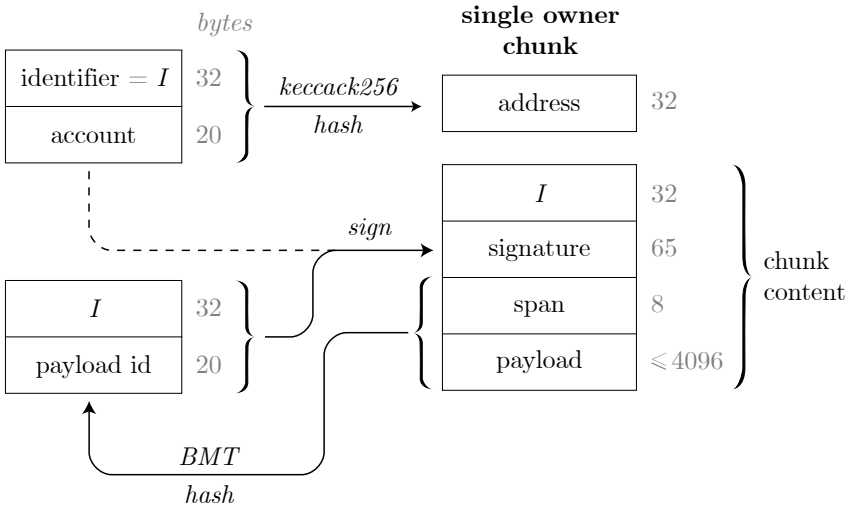

Figure 11: Single-owner chunk. The chunk content is composed of headers followed by an at most 4KB payload. The last header field is the 8 byte span prepended just like in content addressed chunks. The first two header fields provide single owner attestation of integrity: an identifier and a signature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

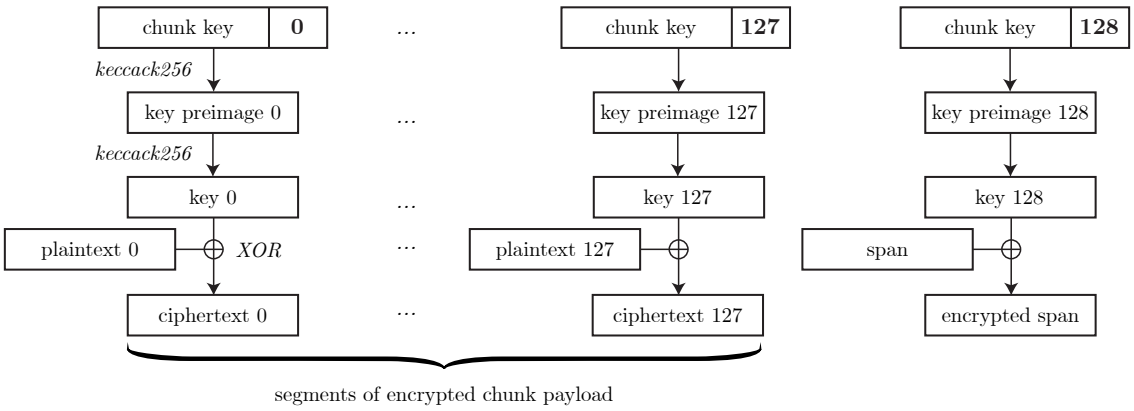

Figure 12: Chunk encryption in Swarm. Symmetric encryption with a modified counter-mode block cipher. The plaintext input is the content padded with random bytes to 4 kilobytes. The span bytes are also encrypted as if they were continuations of the payload.

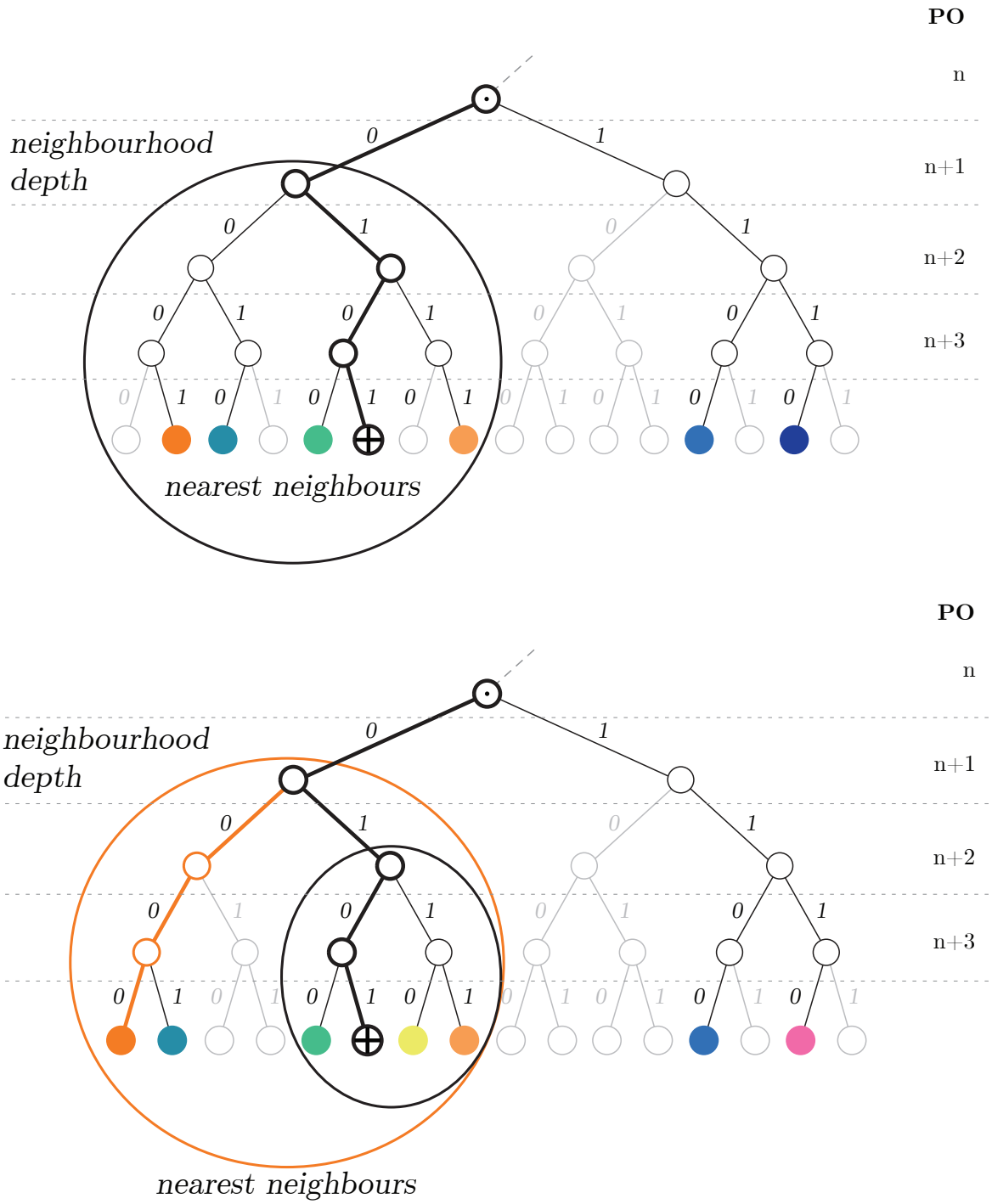

Figure 13: Nearest neighbours. **Top**: Each PO defines a neighbourhood, the neighbourhood depth of the node (black circle) is defined as the highest PO such that the neighbourhood has at least R=4 peers (redundancy parameter) and all shallower bins are non-empty. **Bottom**: An asymmetric neighbourhood.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

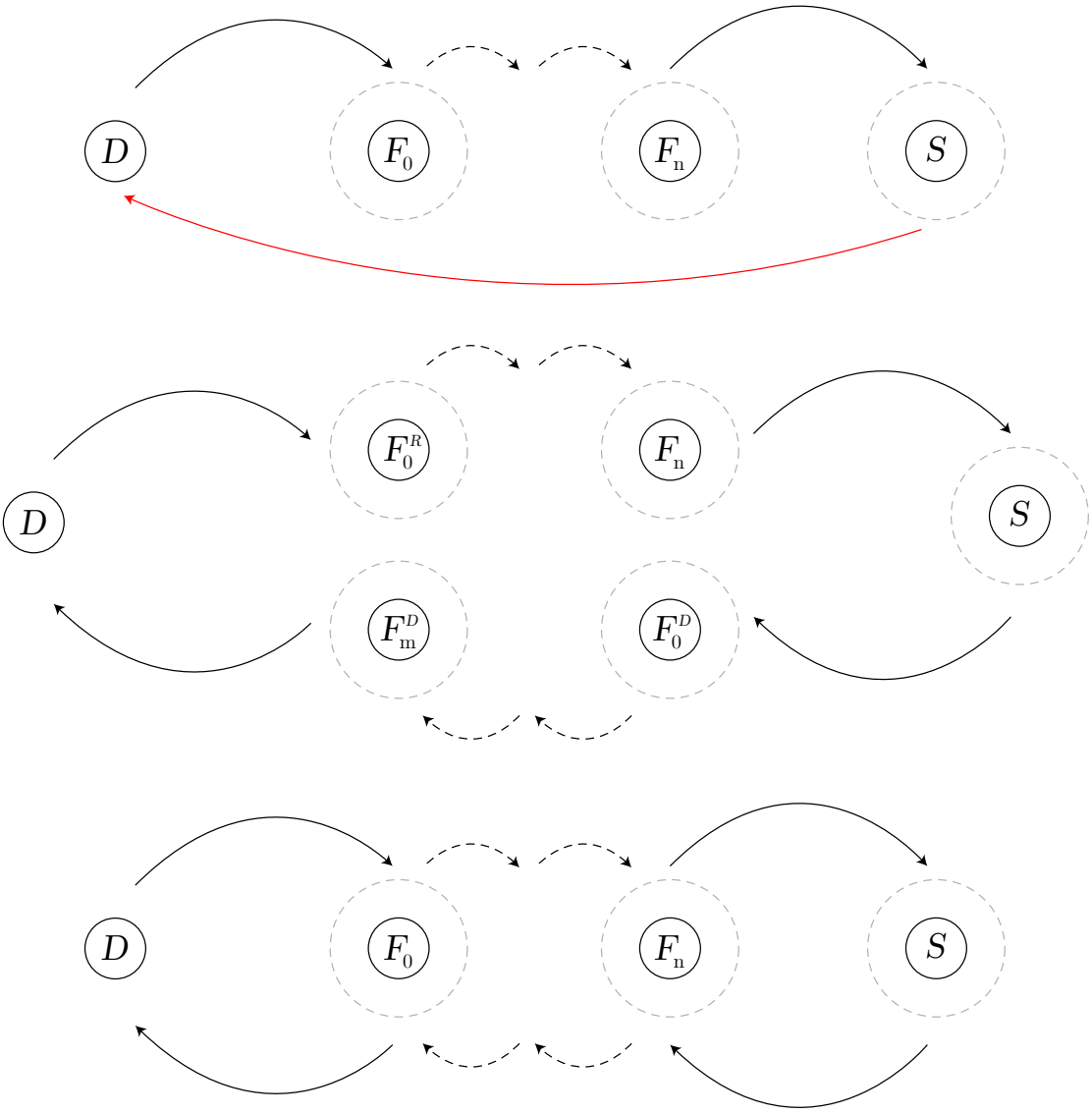

Figure 14: Alternative ways to deliver chunks. **Top:** *direct delivery*: via direct underlay connection. **Centre:** *routed delivery*: chunk is sent using Kademlia routing. **Bottom:** backwarding re-uses the exact peers on the path of the request route to relay the delivery response.

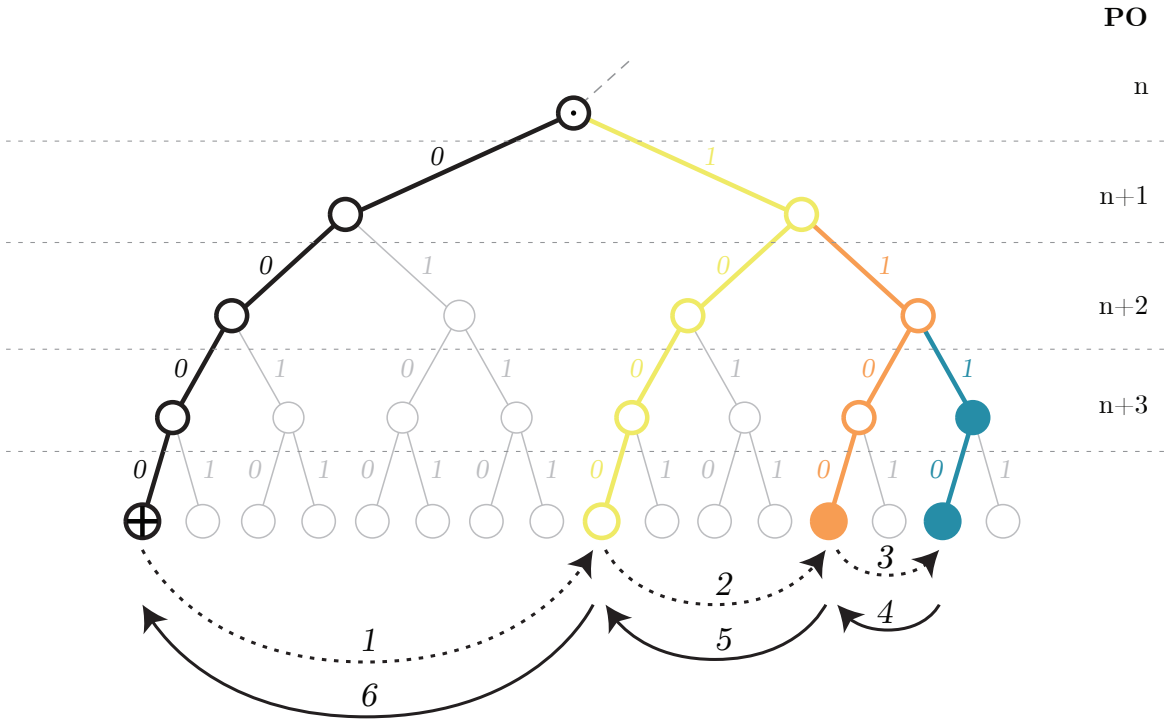

Figure 15: Backwarding: pattern for anonymous request–response round-trips in forwarding Kademlia. Here a node with overlay address ...0000... sending a request to target ....1111... to which the closest online node is ...1110... The leading ellipsis represents the prefix shared by the requestor and 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.

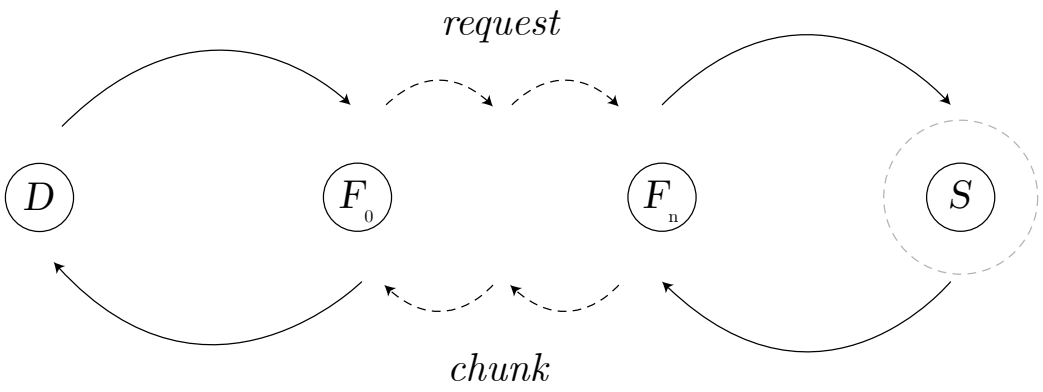

Figure 16: Retrieval. Node D (Downloader) sends a retrieve request to the chunk's address. Retrieval uses forwarding Kademlia, so the request is relayed via forwarding nodes F_0, ..., F_n all the way to node S, the storer node closest to the chunk address. The chunk is then delivered by being passed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

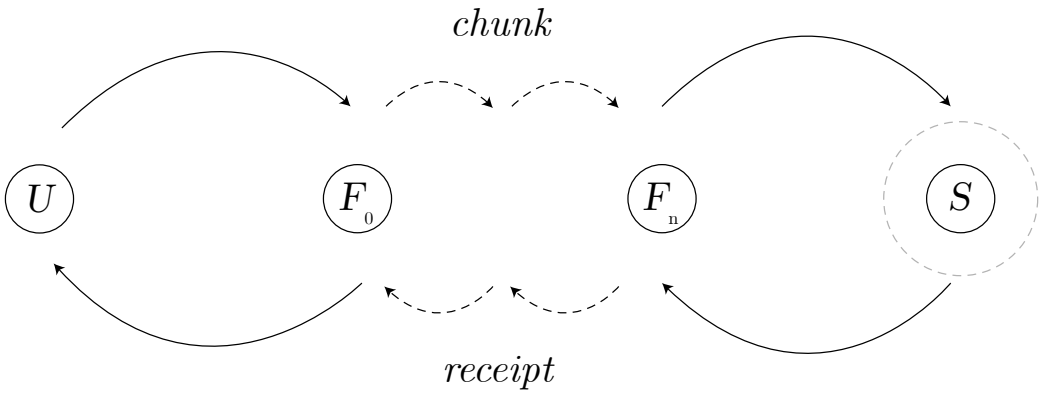

Figure 17: Push syncing. Node U (Uploader) push-syncs a chunk to the chunk's address. Push-sync uses forwarding, so the chunk is relayed via forwarding nodes F_0, ..., F_n all the way to node S, the storer node closest to the chunk address (the arrows represent transfer of the chunk via direct peer-to-peer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

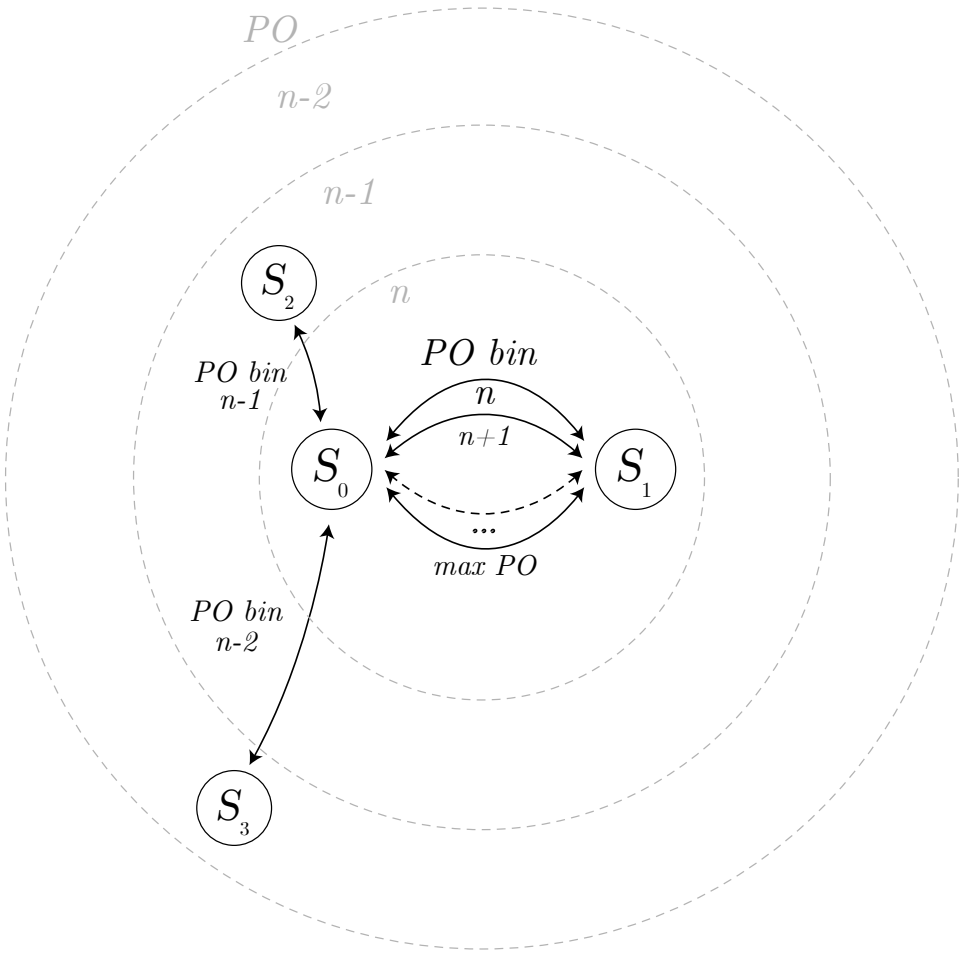

Figure 18: Pull syncing. Nodes continuously synchronise their nearest neighbourhood. If they have free capacity they also pull sync chunks belonging to shallower bins from peers falling outside the neighbourhood depth.

3

INCENTIVES

The Swarm network comprises many independent nodes, running software which implements the Swarm protocol (see chapter **??**). 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).

3.1 SHARING BANDWIDTH

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

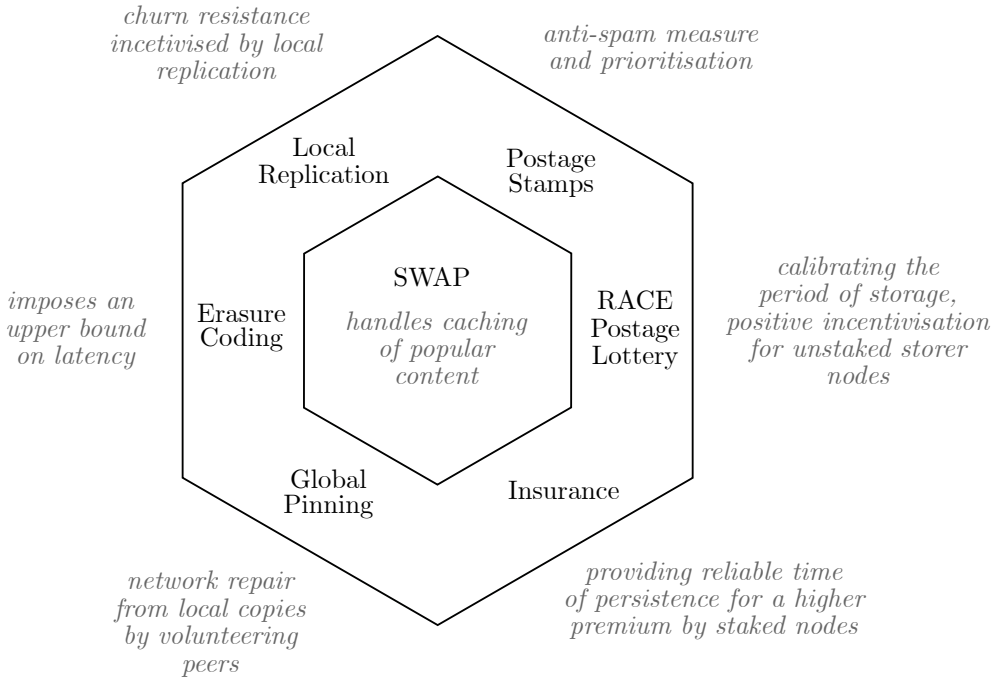

Figure 19: Incentive design

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 the set of connected peers is logarithmic in the 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.

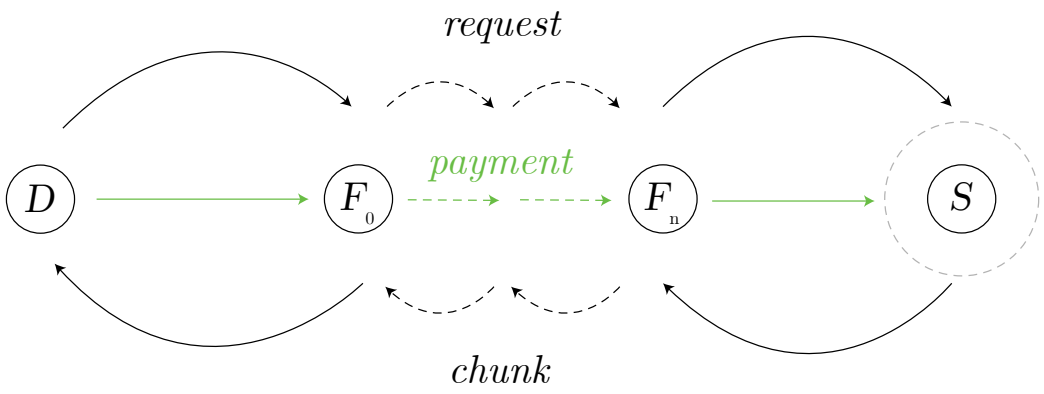

Figure 20: Incentivising retrieval. Node D (Downloader) sends a retrieve request to the chunk's address. Retrieval uses forwarding, so the request is relayed via forwarding nodes F_0, ..., F_n all the way to node S, the storer node closest to the chunk address. The chunk is delivered by being passed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

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 imposing sanctions on peers that send too many requests for chunks that do not exist (see 3.2.6).

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 **??**).

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 **??**). We can conceptualise this message as an alternative response to a request. Nodes maintain the prices associated with each peer for each proximity order, so when they issue a retrieve request they already know the price they commit to pay as long as 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 **??**).

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).

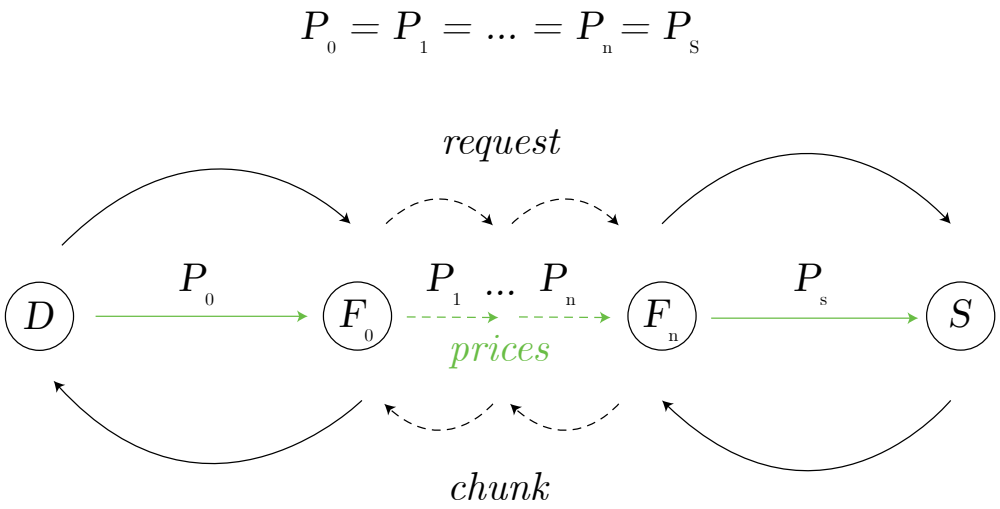

Figure 21: Uniform chunk price across proximities would allow a DoS attack. An attacker can create a flow of traffic between two nodes D and S by sending retrieve requests towards S which only S can serve. If prices are the same across proximities, such an attack would incur no cost for the attacker.

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.

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 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.

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 $n+3$.

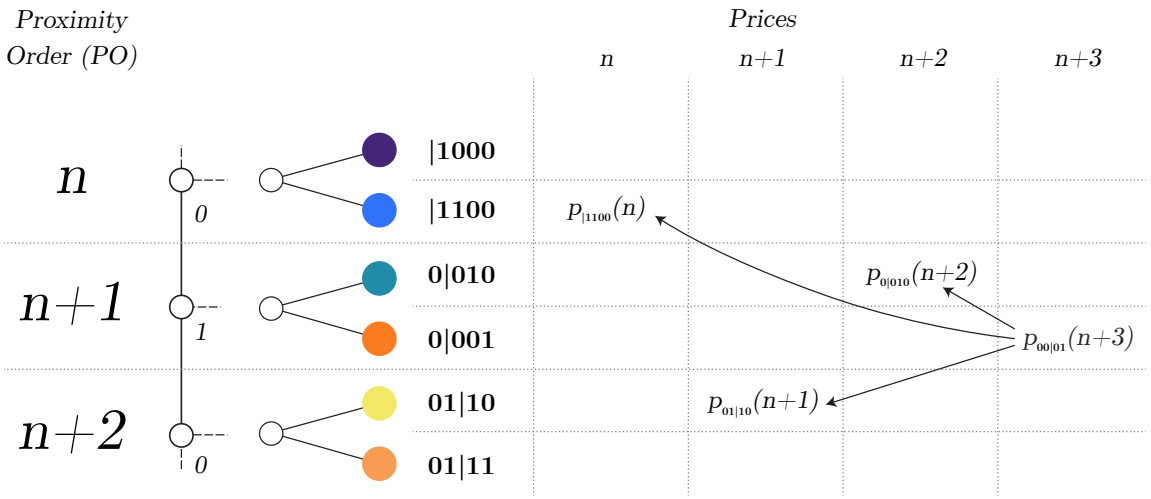

Figure 22: Price arbitrage. Nodes keep a pricetable for prices of every proximity order for each peer. The digaram shows node 0101 trying to forward a retrieve request for 0000. The arrows originate from the closest node, and point to cells where other peers although further from the chunk, offer cheaper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

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 and 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 3.4. 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*.

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 inher-

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

ently 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 3.4). 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.

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 and 3.4, respectively.

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, 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.

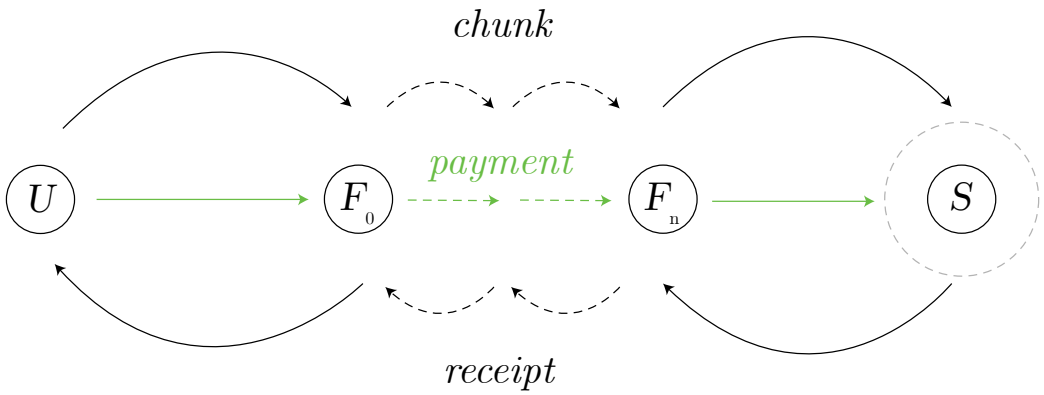

Figure 23: Incentives for push-sync protocol. Node U (uploader) sends the chunk towards its address, the closest node to which is node S (storer) via forwarding nodes $F_0, \ldots F_n$. The storer node responds with a statement of custody receipt which is passed back to the uploader via the same forwarding 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 3.2.6 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 **??**). 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.

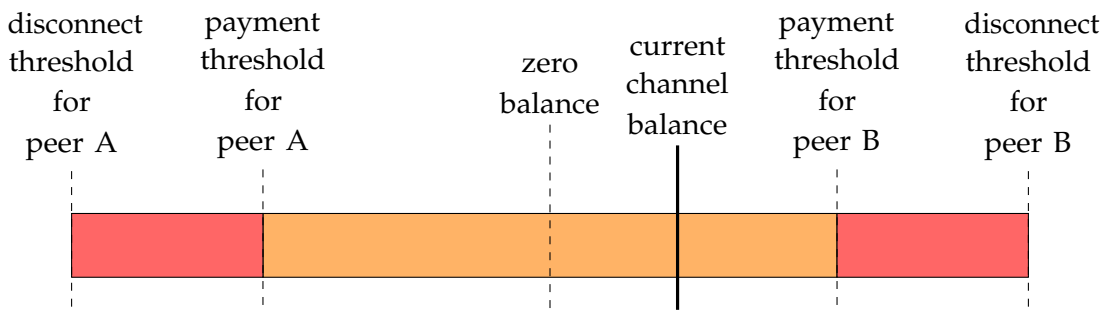

Figure 24: Swap balance and swap thresholds. Zero balance in the middle indicates consumption and provision are equal. The current channel balance represents the difference in uncompensated service provision: If to the right of zero, the balance tilts in favour of A with peer B being in debt, whereas 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 (**??**): 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 **??**).

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.

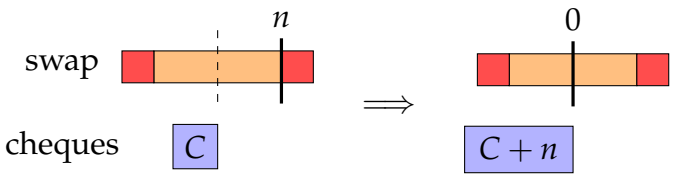

Figure 25: Peer B's swap balance (with respect to A) reaches the payment threshold (left), B sends a cheque to peer A. B keeps the cheque and restores the swap balance to zero.

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.

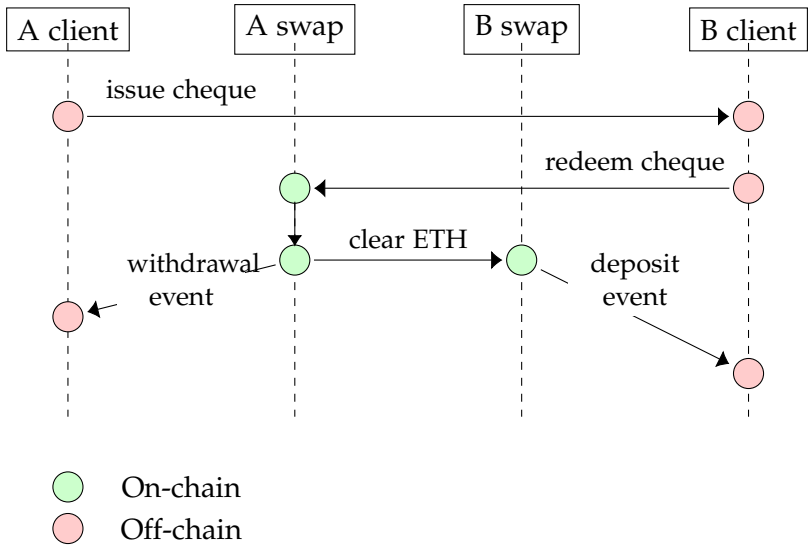

Figure 26: The basic interaction sequence for swap chequebooks

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 uncashed 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 [Poon and Dryja, 2015, Ferrante, 2017, McDonald, 2017, Tremback and Hess, 2015].

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.

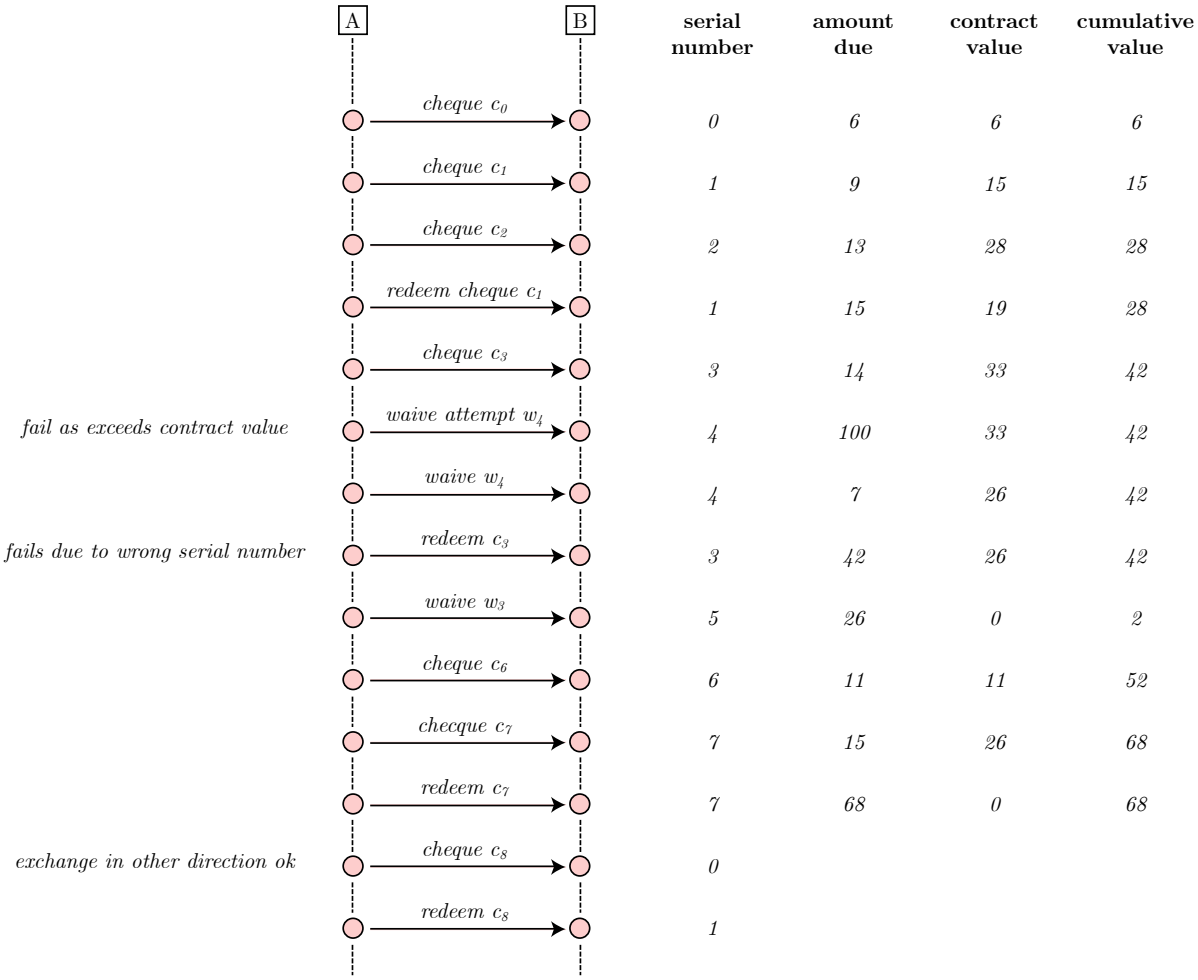

Figure 27: Example sequence of mixed cheques and waivers exchange

With these rules in place it is easy to see how cheque cancellation would work. Suppose user A has issued cheques $c_0 \ldots c_n$ with cumulative totals $t_0 \ldots t_n$ 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

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*.

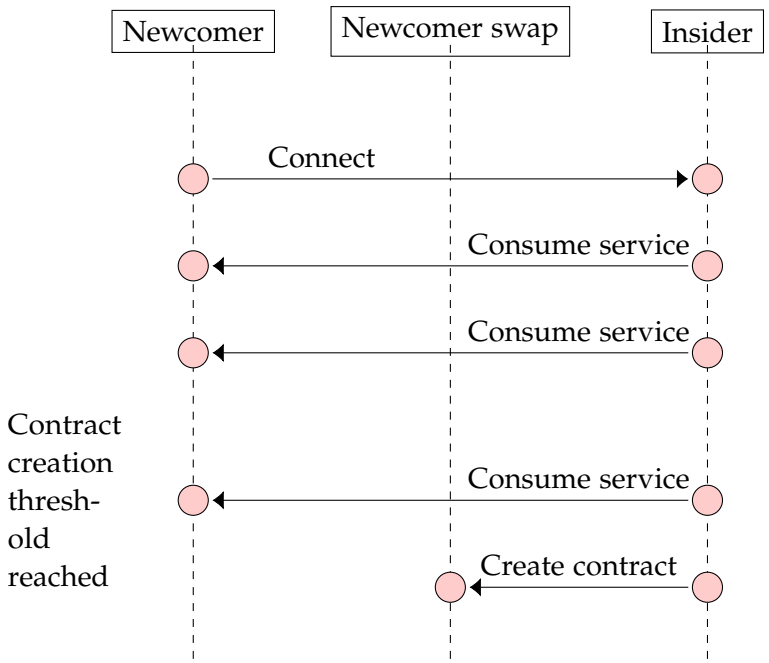

Figure 28: Bootstrapping or how to launch as a swap capable node consuming and providing a service and earn money.

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 *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).

In the case of pull-sync retrieval, 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 **??**) and connectivity (see **??**).

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 POSTAGE STAMPS

A postage stamp is a verifiable proof of payment associated with a chunk witnessed by the signature of its owner. On the one hand, postage stamps prevent frivolous uploads by imposing an advance cost. On the other hand, by ascribing a quantity of BZZ, they signal a chunk's relative importance which storer nodes can then use to rank chunks when selecting which ones to retain and serve, and which ones to garbage collect in the event of capacity shortage.

In this section we first introduce the concept of postage batch enabling the bulk purchase of stamps (3.3.1). In 3.3.2, we explain how limited issuance is represented and enforced. In 3.3.3, we introduce the notion of reserve and detail the rules governing how storer nodes keep it maximally utilised. We conclude in 3.3.4 with exploring the relationship between reserved capacity, effective demand and the number of nodes and their impact on the data availability.

3.3.1 *Purchasing upload capacity*

Uploaders purchase postage stamps in bulk in the form of a postage batch from the postage smart contract on the Ethereum blockchain. Postage batches are created by this contract when a transaction is sent to its batch creation endpoint, together with an amount of BZZ tokens and transaction data specifying some parameters. As the transaction executes, a new batch entry is registered in the postage contract with the following pieces of information:

- *batch identifier* – A random ID that is generated to provide reference for this batch.
- *batch depth* – Base 2 logarithm of the issuance volume, i.e., number of chunks that can be stamped using this batch.
- *owner address* – The Ethereum address of the owner entitled to issue stamps, as per the transaction data sent along with the creation or the transaction sender if not specified.
- *per-chunk balance* – The total amount sent along with the transaction divided by the issuance volume.
- *mutability* – A boolean flag indicating if the storage slots of the batch can be reassigned to another chunk with a stamp if its timestamp is older.
- *uniformity depth* – the base 2 logarithm of the number of equal-size buckets the storage slots are arranged in.

The postage contract provides endpoints to users to modify the per-chunk balance of batches, i.e., add funds to extend the validity period of the stamps issued by the batch (*top-up*) or add volume to decrease it (*dilute*). Anyone can then choose to top up the balance of a batch at a later date but only the owner can dilute it.[4]

Owners issue postage stamps in order to attach them to chunks. A batch has a number of storage slots effectively arranged over a number of equal sized buckets. Issuing a stamp means to assign a chunk to a storage slot. A stamp is a data structure comprising the following fields (see figure 29):

- *chunk address* – The address of the chunk the stamp is attached to.
- *batch identifier* – The ID referencing the issuing batch (generated at its creation).
- *storage slot* – An bucket index referencing one of the equal sized buckets of the batch and a within-bucket index referencing the storage slot the chunk is assigned to.
- *timestamp* – The time the chunk is stamped.

4 As a planned feature, the remaining balance of a batch can be reassigned to a new batch, resulting in the immediate expiry of the original.

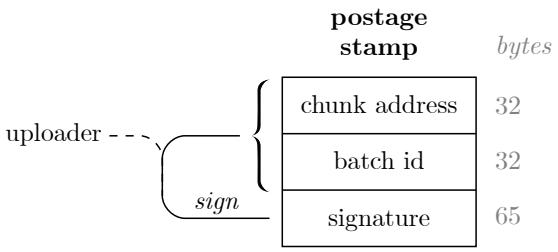

Figure 29: Postage stamp is a data structure comprised of the postage contract batch id, storage slot index, timestamp the chunk address and a witness signature attesting to the association of these four. Uploaders and forwarders must attach a valid postage stamp to every chunk uploaded.

– *witness* – The batch owner's signature attesting to link between the storage slot and the chunk.

A postage stamp's validity can be checked by verifying that it scores all true on the following five attributes:

– *authentic* – The batch identifier is registered in the postage contract's storage.
– *alive* – The referenced batch has not yet exhausted its balance.
– *authorised* – The postage stamp is signed by the address specified as the owner of the batch.
– *available* – The referenced storage slot is within range given the batch depth, and, in the case of an immutable batch, has no duplicates.
– *aligned* – The referenced storage slot has the bucket specified and it aligns with the chunk address stamped.

All this can be easily checked by nodes in the swarm only using information available on the public blockchain (read-only endpoints of the postage contract). When a chunk is uploaded, the validity of attached postage stamp is verified by forwarders along the push-syncing route (see figure 30).

The normalised per-chunk balance of a batch is calculated as the batch inpayment divided by the batch size in chunk storage slots. The chunk balance is interpreted as an amount pre-committed to be spent on storage. The balance decreases with time as if *storage rent* was paid for each block at the price dictated by the price oracle contract.

This system allows prepayment for storage without having to speculate on the future price of storage or fluctuations in the currency's exchange rate. At the cost of decreased certainty about the expiration date, one gains resilience against price volatility. On top of this, uploaders can enjoy the luxury of non-engagement by tying up more of

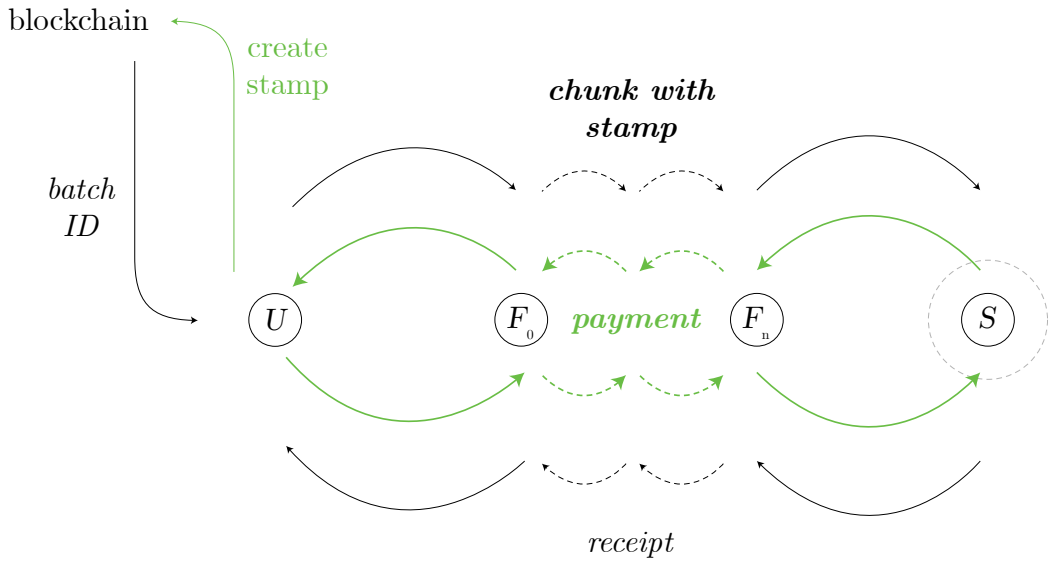

Figure 30: Postage stamps are purchased in bulk on the blockchain and attached to chunks at upload. They are passed along the push-syncing route together and their validity is checked by forwarders at each hop.

the batch balance; while it serves as collateral against price increase, if that does not happen the funds can still be used up (for storing).

3.3.2 *Limited issuance*

Purchasing a postage batch effectively entitles the owner to issue a fixed amount of postage stamps against the batch ID called the issuance volume or batch size. It is restricted to the powers of 2 and is specified using the base 2 logarithm of the amount which is called batch depth. Storage slots of a batch are arranged in a number of buckets and are indexed within the bucket. The number of buckets is restricted to the powers of 2 and is specified using its base 2 logarithm called uniformity depth. The size limitation of a batch with batch depth d and uniformity depth u is equivalent to the conditions that 1) the bucket index ranges from 0 to $2^u - 1$, 2) the within-bucket index ranges from 0 to $2^{d-u} - 1$ and 3) there are no duplicate indexes.

While 1) and 2) is easily verifiable by any third party, 3) is not. In order for index collisions to be detectable by individual storer nodes, uniformity depth must be large enough to fall within nodes' area of responsibility. As long as this is maintained, all chunks in the same bucket are guaranteed to land in the same neighbourhood, and, as a result, duplicate assignments can be locally detected by nodes (see figure 31).

In order to keep their stamps collision-free, uploaders need to maintain counters for how many stamps they have issued for each bucket of a batch and must not issue more than the allowed bucket size.

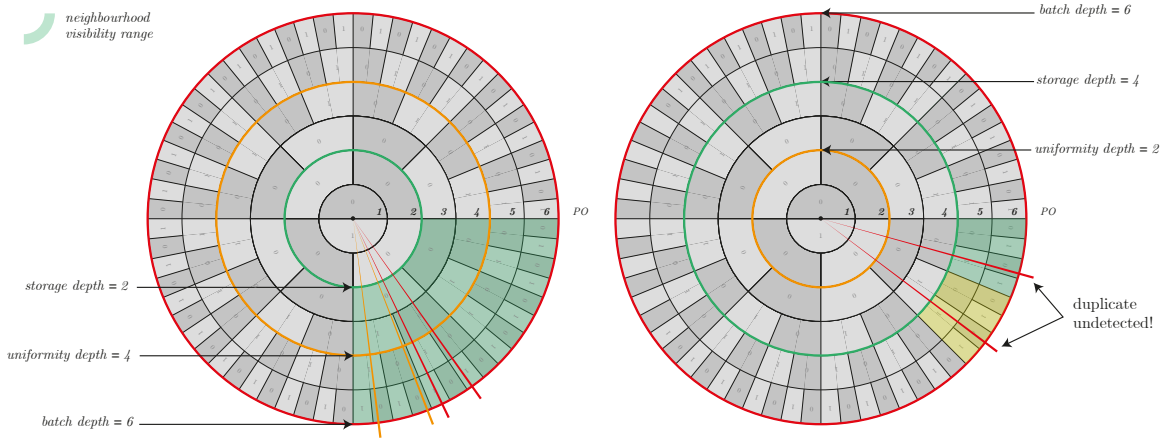

Figure 31: Batches come with 2^u equal-sized buckets (u is uniformity depth, orange circle) each containing an equal number of storage slots (2^{d-u}) adding up to batch capacity of 2^d chunks (d is batch depth, red circle). Storage slots are indexed and the index is associated with a chunk via the stamp signature. Postage stamp over-issuance is detected locally by storer nodes as long as the buckets are deeper than their storage depth (green circle), as in the diagram on the left. In this case they will receive all the chunks that are correctly assigned to the relevant bucket (orange radii) and correctly identify collisions (red radii) by forbidding indexes that are either out of range ($\geq 2^{d-u}$) or multiply assigned. In contrast, the diagram on the right shows it is not possible for a node with storage depth 4 to identify duplicates for a batch with $u = 2$.

In general, the most efficient utilisation of a batch is by filling each bucket fully. Continued non-uniformity (i.e., *targeted issuance*) leads to underutilised batches, and therefore a higher unit price for uploading and storing each chunk. This feature has the desired side effect that it imposes an upfront cost to non-uniform uploads: the more concentrated the distribution of chunks of an upload, the more storage slots of the postage batch remain unused. In this way, we ensure that targeted denial-of-service attacks against a neighbourhood (i.e., uploading a disproportionate number of chunks in a particular address range) is costly since the *inert cost* (due to the degree of under-utilisation of the batch) is exponential in the depth of the skew.

Beyond DoS protection, postage stamps can serve as a *fiduciary signal* indicating how much it is worth for a user to persist a chunk in Swarm. In particular, the per-chunk balance of batches can provide the differential a priori bias determining which chunks should be protected from garbage collection in the absence of evidence to predict their profitability from swap.

3.3.3 *Rules of the reserve*

The reserve is a fixed size of storage space dedicated to storing the chunks in the node's area of responsibility. Chunks in the reserve are the chunks that are protected against garbage collection with valid postage stamps. When batches expire, i.e., their balance is completely depleted, the chunks they stamped are no longer protected from eviction. Their eviction from the reserve frees up some space that can accommodate new or farther chunks belonging to valid batches.

From the point of view of incentives, chunks which are of the same proximity order and the same batch are equivalent. When it comes to eviction due to batch expiry, these equivalence classes, called batch bins are handled as one unit: the chunks in a batch bin are evicted from the reserve and inserted to the cache in one atomic operation.

Assuming a global oracle for the unit price of rent and a fixed reserve capacity prescribed for nodes, the content of the reserve is coordinated with a set of constraints on batch bins called the rules of the reserve:

- if a batch bin of a certain PO is reserved then the batch bins are reserved also for all closer bins (higher PO).
- if a batch bin is reserved at a certain proximity order (PO), then all the batch bins at the same PO belonging to batches with a greater per-chunk balance are also reserved.
- the reserve should not exceed capacity.
- the reserve is maximally utilised, i.e, cannot be extended and have 1-3 remain true.

The first rule means the reserve is closed upwards for PO, which encodes a global preference for chunks closer to the node's address. This is incentivised by routing: keeping the closest chunks, a node will maximise the number of receipts it can issue and the number of retrieve requests it can respond to and at the same time provides the widest coverage within the neighbourhood even after the neighbourhood is no longer supporting the desired redundancy.

The second rule expresses the constraint that the reserve for a PO is upward closed for per-chunk balance, which encodes a secondary preference among chunks of the same proximity for those stamped using a batch with higher per-chunk balance. This is incentivised by the differential absolute profit chunks promise: due to the constraint that balances are not revocable, chunks with higher balance expire later and therefore

contribute more to storers' absolute profit than those expiring earlier despite the same rent paid during their period of validity.[5]

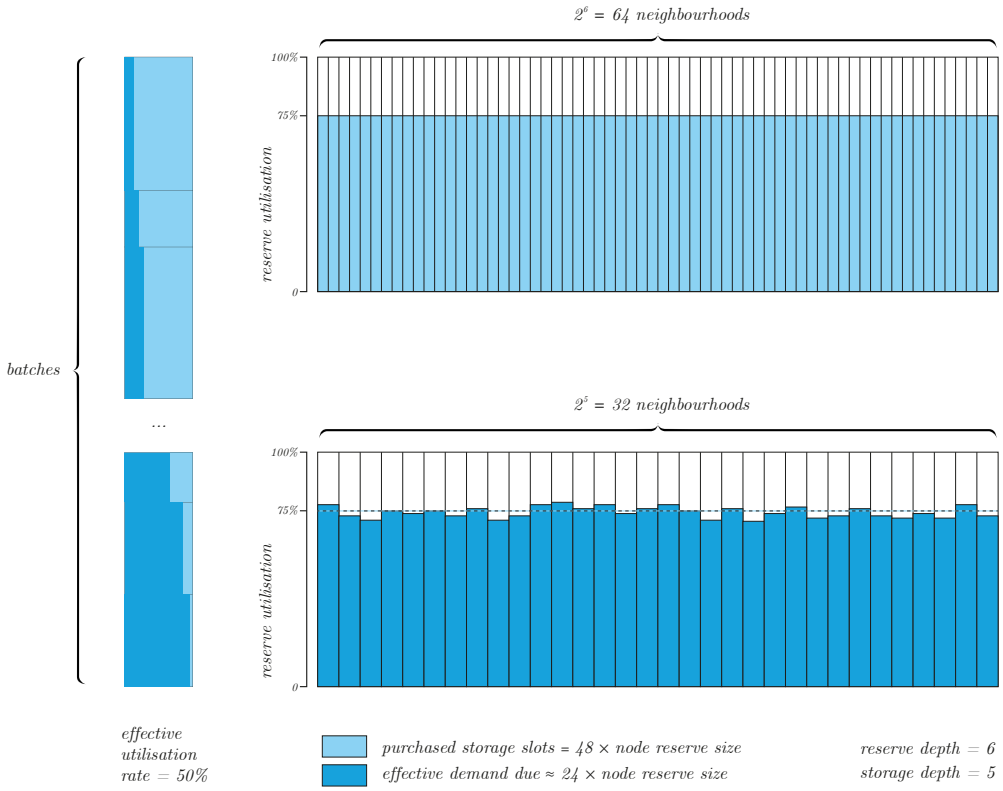

Figure 32: Potential demand for chunk storage is expressed by the total size of all batches with non-zero balance on the blockchain (left). The lower bound on neighbourhood depth to store this capacity is the reserve depth (top right). Storage depth marks the effective volume of chunks uploaded and stored in a neighbourhood's reserve (bottom right). The difference between them is a result of partial batch utilisation. The uniformity of the volume of chunks across neighbourhoods is incentivised by the efficient utilisation of postage batches.

When a new chunk arrives in swarm through pull-sync, push-sync or upload, the validity of the attached postage stamp is verified. If the PO of the chunk is lower than the batch depth, the node inserts the chunk into the garbage collection index, otherwise it is by definition in the reserve. If the reserve size is above capacity, a number of batch bins are identified so that their total size covers the excess so that after these batch bins are *evicted* from the reserve, the reserve size will be within capacity.

5 Note that even if there was no scheme for redistributing postage revenue and the inpayments are frozen/burnt, this strategy is still mildly incentivised in as much as it is aligned with token-holders interest: batches with higher balance exert more deflationary force on the token (per chunk, i.e, the unit of invested resource) by keeping their balance frozen which is expected to realise in a proportional price increase.

3.3.4 *Reserve depth, storage depth, neighbourhood depth*

Reserve depth

The potential demand for chunks to be stored in the DISC is quantified by the total storage slots of valid batches. This is calculated as the sum of the sizes of non-expired batches. Since the batches and their balances are recorded in the postage contract, the reserved DISC size is under consensus.[6]

The base 2 logarithm of the DISC reserve size rounded up to the nearest integer is called the reserve depth. The reserve depth is the shallowest PO such that disjoint neighbourhoods of this depth are collectively able to accommodate the volume of data corresponding to the total number of chunks paid for, assuming that nodes in the neighbourhood have a fixed prescribed storage capacity to store their share of the reserve.

The reserve depth is also the *safe lower bound* for pull-syncing, i.e, the farthest bin a neighbourhood needs to synchronise to guarantee storing the reserve. Conversely, if any neighbourhood marked by reserve depth has no nodes in it, the swarm is not working correctly, i.e., chunks with valid stamps are not protected from getting lost. See figure 33.

Storage depth

The effective demand for chunks to be stored in the DISC is the total number of chunks actually uploaded. While each chunk in the reserve has a valid postage batch and therefore is assigned to a storage slot, a postage batch can always have some of its storage slots unassigned. This entails that the number of chunks actually stored in the DISC can in fact be a fraction of the DISC reserve size.

The effective area of responsibility is marked by the proximity order of the farthest batch bin of the reserve assuming the node complies with the rules of the reserve.

6 The volume is best explicitly maintained by the contract by adding the size of newly created batches and deduct the sizes of newly expired batches. DISC reserve size is updated each time a batch is created or topped up and expired batches are removed during each redistribution round, executed as part of the process triggered by the claim transaction.

A node's storage depth is defined as the shallowest *complete* bin, i.e., the lowest PO that compliant reserves stores all batch bins at. Unless the farthest bin in the node's reserve is complete, the storage depth equals the reserve's edge PO plus one.

The storage depth is the *optimal lower bound* for pull-syncing, i.e, the farthest bin the node needs to synchronise with its neighbours to achieve maximum reserve utilisation.[7] Maximum reserve utilisation should be incentivised as part of the storage incentives.

The gap between actual storage depth and the reserve depth exists because of the bulk purchase of stamps. Since entire batches of stamps reserve storage slots that are assigned to chunks only at later times when they are actually uploaded, the batch *utilisation rate* can be substantially less than 1. Storage depth and reserve depth are the same only if all batches are fully utilised.

Neighbourhood depth

Swarm's requirements on local replication is that each neighbourhood designated by the storage depth contains at least four nodes. If neighbourhoods were made of one node, then the outage of that one node will make the chunks in the node's area of responsibility not retrievable. With two nodes in a neighbourhood, we significantly improve resilience against ad-hoc outages, but because of connectivity latencies a two-peer neighbourhood still displays unstable user experience. The ideal scenario is to have four nodes per full connectivity neighbourhood, which prompts the following definition: neighbourhood depth for a particular node is the highest PO d such that the address range designated by the d-bit-long prefix of the node's overlay contains at least 3 other peers.

Figure 33 details the potential relative orders of the three depths and their consequences on the health, efficiency and redundancy of the swarm.

7 The nodes will have full connectivity up to the shallowest bin that they are pull syncing. This choice is incentivised by the risk of having two disjoint connected sets of pull-syncing nodes resulting in non-consensual reserve. As a consequence, we can say that storage depth is an upper bound on the depth of full connectivity.

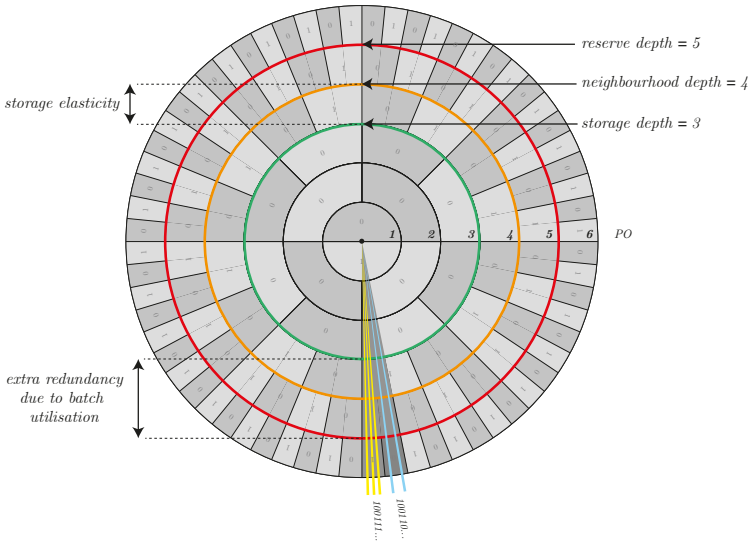

Figure 33: The 3 depths (reserve, storage and neighbourhood) express the order of magnitude of reserved capacity (potential demand, red circle), uploaded chunks (effective demand, green circle) and the number of nodes (effective supply, orange circle), respectively. Their possible orderings express different scenarios with characteristic impact on data availability. Storage depth cannot be greater than reserve depth. A gap between storage depth and reserve depth quantifies the average batch utilisation rate. The gap between storage depth and a deeper neighbourhood depth quantifies the elasticity of the storage: the difference expresses how many times the effective volume can double before redundancy goes below required. While such oversupply may be anticipatory of growth in demand, if neighbourhood depth remains deeper than storage depth long term, it may indicate excessive profits. The opposite order indicates undersupply (redundancy below the desired level).

3.4 FAIR REDISTRIBUTION

The system of positive[8] storage incentives in Swarm is concerned with redistributing to storage providers the BZZ tokens that uploaders deposited within the postage contract.[9] The overall balance on the contract covers the reward pot which represents the total storage rent cumulated over all postage batches for a particular period. The storage rent must be redistributed to storage providers in a way that guarantees that their earnings are proportional to their contribution weighing in storage space, quality of service and length of commitment.

8 The concept of *positive incentives* refers to a scheme whereby providers of a service are entitled to reward but there is no loss involved if they discontinue their service or are not online.

9 As explained earlier, uploaders pay in an unwithdrawable amount to the postage contract which serves as the balance to pay storage rent. In exchange they obtain the right to issue a fixed number of postage stamps which they attach to chunks they want the network to store.

The procedure for redistribution is best conceived of as a game orchestrated by a suite of smart contracts on the blockchain. Nodes earn the right to play through participation in storing and syncing chunks. The winners claim their reward by sending a transaction to the game contract.

In section 3.4.1, we formulate the idea of redistribution in terms of probabilistic outpayments to allow an easy proof of fairness. Next, in 3.4.2, we introduce the mechanics of the redistribution game. Then, in 3.4.3 and 3.4.4 we explain how we enforce maximum utilisation of dedicated storage for persisting relevant content redundantly. We conclude in 3.4.5 by discussing how to read certain aspects of the game as price signals that render the network self-regulating through automatic price discovery.

3.4.1 *Neighbourhoods, uniformity and probabilistic outpayments*

In this section we argue that the efficient use of postage batches incentivises a balanced chunk distribution which in turn gives rise to uniform storage depth across neighbourhoods. We then explain how this enables a fair system of redistribution using probabilistic outpayments.

Assuming an oracle that sets the unit price of storage, the storage rent due for a period of time for a batch can be calculated. The number of rent units for a batch is the result of multiplying the size of the batch with the number of blocks in the period. The price of rent is calculated from the number of rent units multiplied by the unit price.[10] The total storage rent cumulated over all batches for the period between two outpayments constitutes the *reward pot* for the round.

Instead of dividing the reward pot among neighbourhoods regularly, the entire reward pot can be transferred to (representative nodes in) one target neighbourhood in each round. This probabilistic outpayment scheme is fair on the level of neighbourhoods as long as we can make sure that over a large number of rounds the probability with which a neighbourhood is selected as the target corresponds to its relative contribution to the overall network storage. Given a constant prescribed reserve capacity and replication of the reserve content by nodes in a neighbourhood, each neighbourhood defined by storage depth contributes equally to the network.

In section 3.3, we wrote that uploaders are strongly incentivised to use their postage batch in a way that the chunks they stamp with it are uniformly distributed across

10 If this theoretical amount is less than the the current balance of the batch, then the batch is expired and the effective rent is only the remaining balance.

the address space. This being true of all batches creates a situation that chunks are uniformly distributed across the DISC. In particular, the sets of chunks sharing a prefix are expected to be roughly equal in size. Therefore we expect nodes to fill their prescribed reserve capacity with chunks at the same proximity order, irrespective of their location in the address space, i.e., the storage depth is uniform across nodes and therefore across neighbourhoods.[11] With neighbourhoods at equal depth, uniform sampling of neighbourhoods can be modelled by choosing the neighbourhood which contains an anchor (called the *the neighbourhood selection anchor*) randomly dropped in the address space (see figure 34).

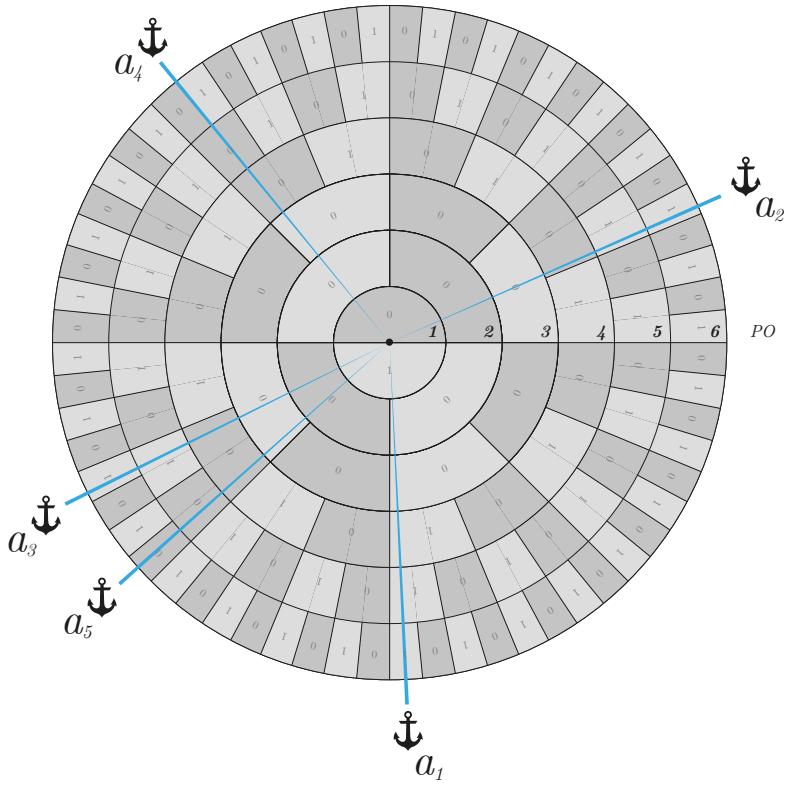

Figure 34: Neighbourhood selection and pot redistribution. The winning locality is selected by the neighbourhood selection anchor. Neighbourhoods that contain the anchor within their storage depth are invited to submit an application by committing to a consensual reserve sample.

11 Differences do occur due to variance but over many rounds, deviation from the mean is meant to be independent of the location.

3.4.2 *The mechanics of the redistribution game*

The cooperation of peers to redundantly store for the network's benefit is underpinned by a Schelling game aimed at proving that the peers agree on the chunks they need to store and they do, in fact, store them. The redistribution game itself is orchestrated by the game contract, one of the building blocks of the system of 4 smart contracts which collectively drive the swarm storage incentive system (see figure 35):

- Postage contract – serving as the batch store to sell postage batches to uploaders, keeping track of batch balances, batch expiry, storage rent and the reward pot itself.

- Game contract – orchestrates the redistribution rounds interacting with potential winners accepting commit, reveal and claim transactions from storage providers in selected neighbourhoods.

- Staking contract – operates a stake registry, maintaining committed stake and stake balance for nodes by their overlay; enables freezing and slashing of stake as well as withdrawal of surplus balance for stakers.

- Price oracle – maintains the unit price of storage rent, accepts signals from the game contract to dynamically adjust according to supply and demand and provides current price oracle service for the other three contracts.

The game is structured as a sequence of *rounds*. Each round lasts for a fixed number of blocks and recur periodically. A round consists of 3 phases: *commit*, *reveal*, and *claim*.[12] The phases are named after the type of transaction the smart contract expects during that phase, and that nodes from the selected neighbourhood need to submit.[13] See figure 36

Once the reveal phase is over, the neighbourhood selection anchor becomes known. Nodes that have the anchor within their neighbourhood[14] are allowed to participate in the following round (see figure 34).

The storer nodes in a neighbourhood are assumed to have consensus over the chunks that make up their reserve and provide evidence called proof of entitlement to the blockchain (discussed below in detail in 3.4.4). In such a game, the Nash-optimal

12 The commit and reveal phases are one quarter of the round length while the claim phase is one half.
13 Both commit and reveal are simple and cheap transactions. The only expensive transaction is claim but that only the winner needs to submit.
14 If storage depth is less than the anchor's proximity order relative to the overlay address.

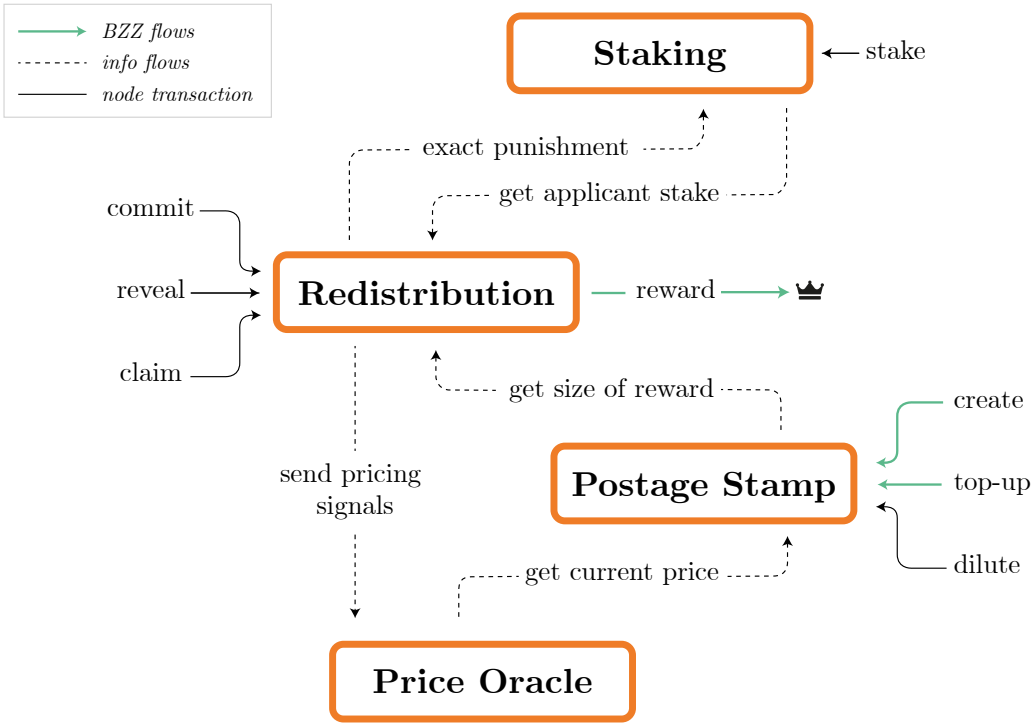

Figure 35: Interaction of smart contracts for swarm storage incentives. The figure shows with the dotted line the information flow between the four contracts comprising the storage incentive smart contract suite as well as the public transaction types they accept.

strategy for each node is to follow the protocols and coordinate to guarantee that from the same information, all neighbouring peers will arrive at the same proof of entitlement. Since the proof of entitlement needs to be consensual but unstealable,[15] a commit/reveal scheme must be used.

In the commit phase, nodes in a neighbourhood will apply by submitting the reserve commitment obfuscated with an arbitrary key that they later reveal. The smart contract receiving the commit transaction verifies that the node is staked, i.e., the registry of the staking contract contains an entry for the node's overlay with a stake that is higher than the minimum stake.

In the reveal phase, each node that previously committed to a reserve, now reveal their commitment by submitting a transaction containing their reserve commitments,

15 Any explicit communication between independent nodes about this reserve before the end of the commit phase constitutes risk in that it may leak the proof to a node not doing storage work. Therefore nodes are incentivised to keep the proof a secret. Making these proofs unstealable helps detect opportunistic peers that pose as storers but do not provide adequate storage.

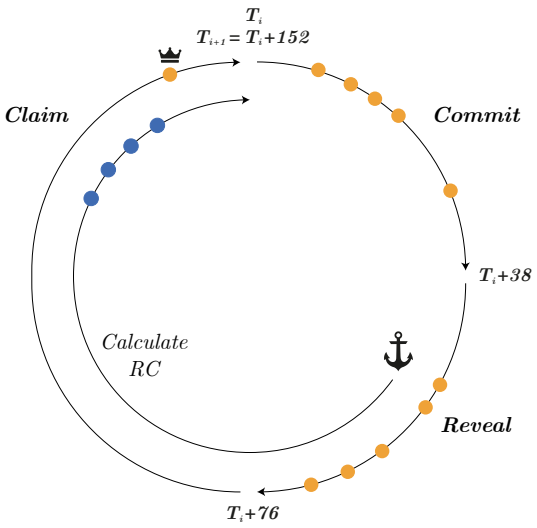

Figure 36: Phases of a round of the redistribution game. The figure displays the timeline of the repeating rounds of the redistribution game with its phases. In the context of smart contract interaction, logically starting with the commit phase, followed by reveal and claim. From the point of view of client node engagement starting with the end of the reveal phase with the neighbourhood selection anchor revealed, those in the selected neighbourhood start calculating their reserve sample only to submit it by the end of the next commit phase. If they are selected as an honest node and as a winner, they submit their proof of entitlement in a claim transaction.

their storage depth, their overlay address, and the key they used to obfuscate the commit. When receiving the reveal transaction, the contract verifies that the revealed data serialised does indeed hash to their commitment. It is also checked if the node belongs to the neighbourhood designated by the neighbourhood selection anchor, i.e., is within the storage depth provided in the reveal.

In the claim phase, the winner node must submit a claim transaction.[16] First, in order to decide the outcome of the Schelling game, one reveal is selected from among the reveals submitted during the reveal phase.[17] The selected reveal represents the truth; the set of applications agreeing with the selected one represent the honest peers of the neighbourhood, the ones disagreeing are the liars, while those committers that did not reveal or revealed invalid data are the saboteurs. Honesty is incentivised by the fact that liars and saboteurs get punished. In what follows we introduce staking that is needed for both the selection processes and the punitive measure.

16 Every node in the selected neighbourhood needs to perform the corresponding calculations to determine whether or not they are the winner.

17 This is relevant only if the depth and/or the commitment are non-uniform across applicants.

3.4.3 *Staking*

Neighbours with shared storage

In order to provide robust protection against accidental node churn, i.e., ensure retrievability of chunks from a neighbourhood in the face of some nodes being offline, the swarm require a number of independent storers in each neighbourhood physically replicating content. If payout was given to each node that shows proof of entitlement, then operators would be incentivised to create spurious nodes with the sole purpose of applying for the reward. Measures can be introduced to enforce that these spurious nodes must be operating on the network, but ultimately, operators may choose to actually run several nodes yet share their storage on a single hardware. The incentive system must ensure that storage providers do not adopt this strategy. To this end, we introduce staking.

Stakes are used as weights by the contract to determine the true reserve commitment (truth selection) as well as the winner among the honest nodes (winner selection). Since peers' relative stake determine their chance of winning, stake is additive, i.e., operators' profit only depends on their total stake within the neighbourhood. Given the cost of running a node, operators will have no motivation to divide their stake between multiple nodes sharing storage hardware.

Committed stake and stake balance

When registering in the staking contract, stakers commit to a stake denominated in rent units called committed stake. The committed stake must have a lower bound.[18] The amount sent with the transaction is recorded and serves as collateral called stake balance. Stakes can be created or topped up any time, but the update time is recorded together with the amount. Participation is restricted to peers whose stake has not changed recently, thereby excluding the possibility of changing stakes after knowing the selected neighbourhood. Every time the stake of a node is queried, the contract returns the absolute committed stake in BZZ calculated as (1) the committed stake in rent units multiplied by the unit price of rent or (2) the entire stake balance, whichever is smaller.

18 A large number of staked nodes could cause the claim transaction to fail due to gas cost needed for iterating over them. This presents a potential attack where the adversary registers stakes for many nodes and commits for all of them. Such an attack is made prohibitively costly by enforcing a minimum stake.

Stakes must be transferable between overlay addresses to facilitate neighbourhood hopping in case the distribution of stake per neighbourhood is unbalanced.

Withdrawability of surplus stake balance

The committed stake lets operators express their profit margin together with time preference for realising this profit. Since the profit is only transparent once the relative stakes within the neighbourhood are known, it is possible that nodes take a while to discover their optimal stake.

If the BZZ token price increases and the unit price of rent drops, the entry for the node in the stake registry will show excess balance. This surplus can always be withdrawn, and, as a consequence, stakers can realise their profit from BZZ appreciation.[19]

3.4.4 *Neighbourhood consensus over the reserve*

Peers applying for the round must agree on which chunks belong to their respective reserves. For this, at the very least, the applicants must consent on their area of responsibility, which can be derived from their storage depth and their overlay address. The consensus over the reserve content is tested with the identity of a reserve sample. The sample is the first k chunks in the reserve using an ordering based on a modified hash of the chunks. The modified chunk hash is obtained using the chunk contents and a *salt* specific to the round.[20] It is impossible for any node to construct this set unless they store all (or a substantial number of the) valid chunks together with their data at or after the time the salt is revealed.

Recency and sampling

The reserve sample must exclude too recent chunks because, otherwise, malicious uploaders could bombard nodes in the neighbourhood with a non-identical set of chunks that are going to be sampled thereby breaking the consensus about the

19 In case the token price goes up substantially, the stake balance ends up worth much more than what nodes can ever expect to earn. If the stake balance was not at all withdrawable, participation would be disincentivised due to fear of losing the potential gains in the event of BZZ token appreciation.

20 This modified hash is the BMT hash of the chunk data using Keccak-256 prefixed with the reserve sample salt as a base hash. The ordering is the ascending integer order reading the 32-byte modified hash as a big endian encoded 256-bit integer.

reserve. One way to guard against this attack is to save each chunk together with its time of storage[21] in the local database. Pairwise synchronisation of chunks between neighbours with the pull-sync protocol respects this ordering by time of storage. We require that live syncing, i.e., syncing of chunks received after the peer connection started has a latency not longer than an agreed constant duration called maximum syncing latency (or *max sync lag* for short). Peer connections lagging more with syncing are by protocol not counted as legit storer nodes. This restriction ensures that malicious nodes can not back-date new chunks more than the max sync lag without losing their storer status.

In order to reach consensus, we must ensure that all chunks received by any node in the neighbourhood not later than l should reach every node of the neighbourhood before the claim phase. If we choose l as 2 times the allowed sync lag then every chunk landing first with a node has time to arrive at each node to be safely included in a consensual sample.[22]

Storage depth and honest neighbourhood size

In order to decide which reveal represents the truth for the current round, one submission out of all reveals is selected randomly with a probability proportional to the amount of stake the revealer has. More precisely: the amount of stake per neighbourhood size, i.e., stake density. The reserve sample hash and the reported storage depth thus revealed are considered the truth for the current round.

Now we can understand why nodes will report actual storage depth correctly. If a node chooses to play with a larger neighbourhood than the neighbours, it will be selected more often than the others. However, as the committed storage depth decreases as compared to peers, the node's stake is counted with an exponentially deflated value relative to the peers reporting a deeper storage radius, making such an attack costly.

Overreporting storage depth is possible as long as the the node falls into this narrower proximity of the neighbourhood selection anchor. Therefore, a systematic exploit requires the malicious actor to control a staked node in each subneighbourhood of the true honest neighbourhood. On top of this, the winners also need to show evidence that the set of chunks within their storage depth do fill their reserve. The actual integer

21 Using the timestamp within the postage stamp to define the minimum age on would not solve the consensus problem since chunks with old postage stamp could be circulated towards the end of sampling and cause disagreement between neighbours.

22 Instead of actually monitoring neighbour connections and abstain from committing to a sample in case of excessive lag, one can just choose a small enough sample size.

values of the transformed chunk addresses in the sample hold information regarding the size of the original sampled set. Requiring the size of the sampled set to fall within the expected range (with sufficient certainty) translates to imposing a constraint on the upper bound of the values of the sample. This construct is called proof of density.

Note that the sample-based density proof can be spoofed if the attacker is mining content filtering chunks in such a way that the transformed chunk addresses form a dense enough sample, then uses its own postage batches to stamp them. Further hardening against such attacks can be achieved by requiring additionally a commitment to the entire set of postage stamps and similarly proving from a randomised sample the custody of a sufficient quantity. Due to this requirement, fraudulent claimants must not only generate the content, but must also have enough storage slots to fake the sample. This would require the attacker to purchase postage batches in the magnitude of the entire network or keep track of and store the actual postage stamps existing in the network. The former imposes a prohibitive cost on the attacker, whereas, in the latter case, the malicious claimant must bear the risk of relying on honest neighbours for the post-hoc retrieval of witness chunk data needed for the proof of entitlement.

Skipped rounds and rollover

If there is no claim in a given round, the pot simply rolls over and increase the outpayment for the next round of the redistribution. This policy is by far the easiest to implement, resulting in the lowest gas expenditures.[23]

The eight rules of entitlement

Here we summarise the eight rules of validating a claim (with committing and revealing a reserve commitment and then submitted evidence as proof of entitlement; see also table 2):

REPLICATION

Since liars get frozen (i.e., nodes that had revealed reserve commitment hash or storage depth different from the winner are excluded from the game for a period), nodes in a neighbourhood are incentivised to replicate their reserve by syncronising the chunks they store using the pull-sync protocol.

23 One might argue for reimbursing honest nodes for their transaction costs. Thereby, nodes with really small stakes can still participate and in general nodes are less exposed to variance in the probabilistic outpayments.

REDUNDANCY

The stake is used as weights in determining the within-neighbourhood probability of a node being selected as winner. This implies no benefit in submitting multiple claims. Operators running multiple nodes in one neighbourhood (sharing storage) therefore have no advantage over running a single node with the same total stake. Assuming this disincentive to proliferate is effective, staking can be regarded as guarantee for true redundancy.

RESPONSIBILITY

At the time of revealing it is checked if the neighbourhood selection anchor falls within the node's radius of responsibility, i.e., belong to the covered range of addresses whose proximity to the node's overlay address is not less than their reported storage depth.

RELEVANCE

Using a witness proof with the reserve commitment hash as root, we show evidence that an arbitrarily chosen segment in the reserve sample packed address chunk is the address of a witness chunk. A valid postage stamp signed off on this witness chunk address is presented to show that storing that chunk in the reserve is relevant to someone (and is paid for).

RETENTION

A segment inclusion proof is provided as evidence that the chunk data has been retained in full integrity.

RECENCY

The salt used for the transformed reserve sample is derived from the current round's random nonce proving that the RS must have been compiled recently. The witness and segment indexes are derived from the next game's random seed ensuring that at the time of compilation and commitment, no compressed or partial retention of chunk data would have been sufficient.

RETRIEVABILITY

The chunk is shown to be retrievable by proximity based routing, i.e. its address belongs to the range of addresses covered by the neighbourhood: the chunk's proximity order to the node's overlay address is not less than their reported storage depth.

RESOURCES

Resource retention checks the volume of resources constituting the reserve by estimating the sampled set size via chunks density and stamps density.

proof of	construct used	attacks mitigated
REPLICATION	Shelling-game over reserve sample	non-syncing, laggy syncing
REDUNDANCY	share of reward proportional to stake	shared storage, over-application
RESPONSIBILITY	proximity to anchor	depth/neighbourhood misreporting
RELEVANCE	scarcity of postage stamps	generated data
RETENTION	segment inclusion proof	non-storage, partial storage
RECENCY	round-specific salt for reserve sample	create proof once and forget data

RETRIEVABILITY	proximity of chunk	depth over-reporting
RESOURCES	density-based reserve size estimation	targeted chunk generation (mining)

Table 2: r8: proofs used as evidence for entitlement to reward.

3.4.5 *Pricing and network dynamics*

In this section, we first put the redistribution scheme in the context of self-sustainability, and provide a simple solution for price discovery.

For Swarm to be a truly self-sustaining system, the unit price of storage rent must be set in a way that is responsive to demand and supply. Ideally, the price is automatically adjusted based on reliable signals resulting in dynamic self-regulation. The guiding insight here is that the information storer nodes provide when they apply for the redistribution game, also serves as a price signal. In other words, the redistribution game serves as a decentralised price oracle.

Splitting and merging of neighbourhoods

The storage depth represents the proximity radius within which the neighbourhood's storer nodes keep all chunks with valid postage stamps and fill their reserve.

If the volume of newly issued storage slots from recently purchased batches (*ingress rate*) and the volume of expired storage slots (*outgress rate*) balance out, the storage depth remains unchanged. But see what happens if the volume of reserved chunks increase? Now, since the client's reserve capacity is constant, after a while, nodes are able to fill up their capacity with chunks that are at most one proximity order closer to them than the farthest chunks were previously, i.e., their storage depth increases. When the volume of reserved chunks doubles, the storage depth increases by one.

In order to store this excess data under the same redundancy constraints the network requires double the number of nodes. If all else is equal, double the network-wide reserve, double the postage revenue and therefore double the overall pot that gets redistributed. When neighbourhoods split as they are absorbing the new volume, they simultaneously release the chunks in the PO bin of their old depth, i.e., the chunks now stored by their sister nodes.

Utilisation rate is an organic way to introduce pressure against fully maxing out a node's reserve with critical content, and thereby enable early detection of capacity

pressure. This provides sufficient safety buffer for the triggered incentives to take effect. For instance, if utilisation rate is 1/8, the storage depth is up to 3 PO-s shallower than the reserve depth.[24] Now the ingress can be really high and bring the reserve depth down to storage depth. When the tendency of a closing gap between the potential (reserved) and actual (observed) utilisation of the DISC is detected, any incentive change will have the buffer to take effect without target redundancy being threatened.

Number of honest nodes as price signal

Since the storage capacity is maxed out, the ratio of supply and demand is directly seen in the number of honest nodes playing the Schelling game.

We assume that if nodes are staying in the network for a longer period, their doing so testifies to their profitability. For a stable swarm, neighbourhoods need only 4 (balanced) nodes within as neighbourhood. Assuming equal stake (or more precisely, assuming that relative stake equalises profitability of node operators) if there are n nodes in a neighbourhood, their long term profit is equally shared, this amount is optimised if there are exactly four nodes ($n = 4$). This number can be more, since opportunistic operators may start their nodes in a complete neighbourhood in anticipation of a neighbourhood split due to capacity demand. As these nodes stay in, the same long term winnings of the neighbourhood gets distributed among more nodes than optimal. However, the fact that nodes tolerate this implies that the reward is too much (the price is too high), and the network can tolerate a decreasing price.

On the other hand, if the number of honest revealers is less than the neighbourhood redundancy requirement, it signals capacity shortage and therefore requires the storage rent to increase.

Parameterisation of the price oracle

The rule for updating the price from one round to the next is that the current price is multiplied by a value m which depends on the number of honest revealers in the round. Mathematically, $p_{t+1} = mp_t$, where p_t is the price in round t (and p_{t+1} is then the price in the following round). We define the multiplier m in terms of the number of honest revealers r and a stability parameter σ governing how quickly the price should increase or decrease, all other things equal.

24 The narrative of this scenario is that uploaders with underutilised batches subsidise extra redundancy for everyone.

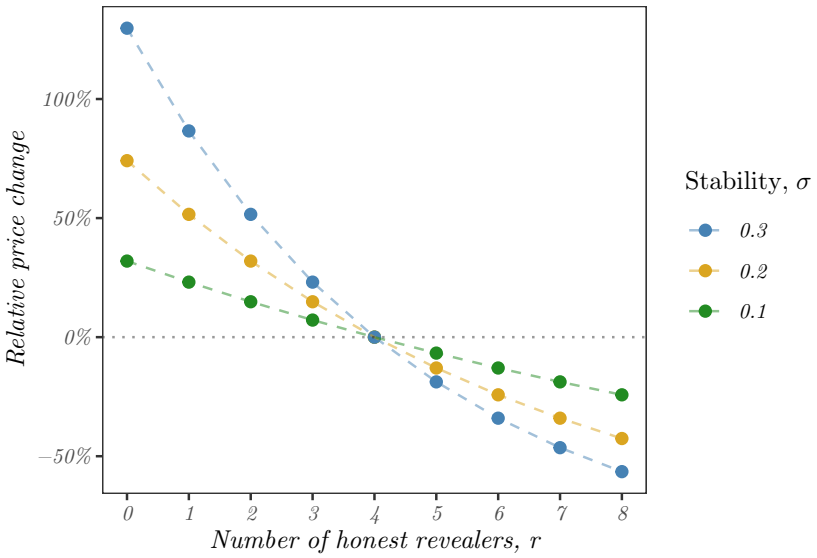

Figure 37: Adaptive pricing. The relative change in price (y-axis), mathematically expressed as the price in the next round divided by the price this round minus one ($p_{t+1}/p_t - 1$), is displayed against the number of honest revealers r in the current round (x-axis). This is done so for three different values of the stability parameter σ (colours). The points are the actual price change values; the connecting dashed lines are for visual aid only. The dotted horizontal line highlights the point at which no price change happens. Price change is exactly zero for any σ when the number of honest revealers is four. Otherwise, larger values of σ lead to larger relative price changes as the number of honest revealers is varied.

In particular, we choose $m = 2^{\sigma(4-r)}$, and therefore we have $p_{t+1} = 2^{\sigma(4-r)} p_t$. This expresses how the deviation $4 - r$ of the number of revealers from the optimal value of 4 maps to an exponential change in price. The stability parameter σ determines the generic smoothness of price changes across rounds, i.e., how many rounds it takes for the price of rent to double in case of a consistent signal of the lowest degree of undersupply (or halve in case of a consistent oversupply). Figure 37 illustrates how the price model works.

Two minor adjustments are applied to this simple model. First, r is capped at some value $r_{\max}$ (chosen to be 8 in our case). That is, r should actually be interpreted as the minimum of the number of honest revealers and $r_{\max}$. Second, the price is never allowed to drop below some predetermined minimum $p_{\min}$. That is, in case the price drop from one round to the next would bring the price below $p_{\min}$, it will instead be kept at $p_{\min}$.

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 38).

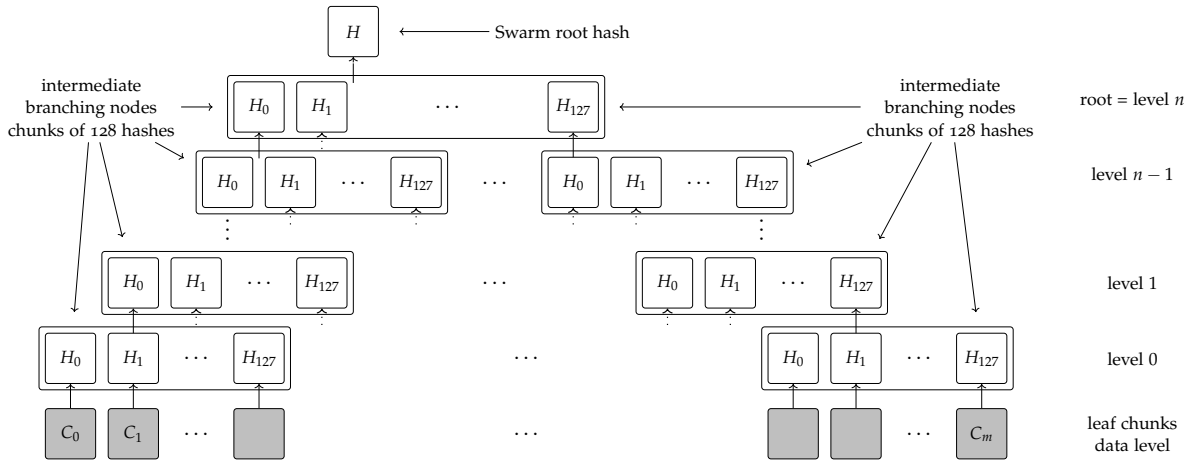

Figure 38: Swarm hash: data input is segmented to 4-kilobyte chunks (gray), that are BMT hashed. Their hashes are packaged into intermediate chunks starting on level 0, all the way until a single chunk remains on level n.

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 **??**) 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.

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 39. By recursively unpacking these from the root chunk down, we can arrive at a sequence of data chunks.

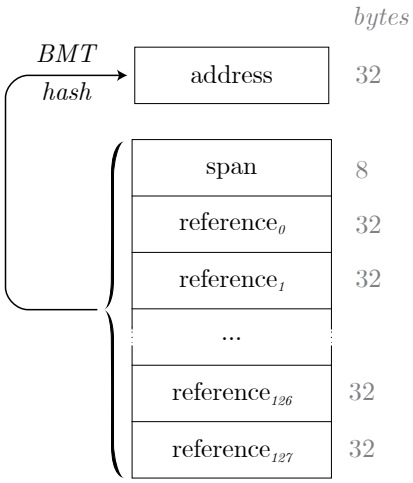

Figure 39: Intermediate chunk. It encapsulates references to its children.

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 40).

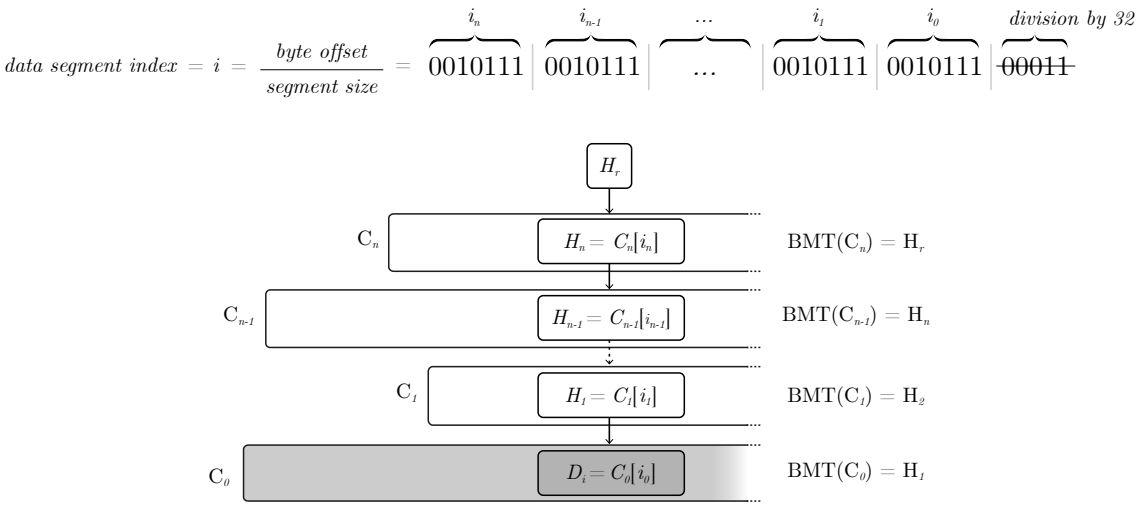

Figure 40: Random access at arbitrary offset with Swarm hash. The arbitrary offset informs us how to traverse the Swarm hash tree.

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 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 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 **??**) where the length of the sequence corresponds to the depth of the Swarm hash tree (see figure 41).

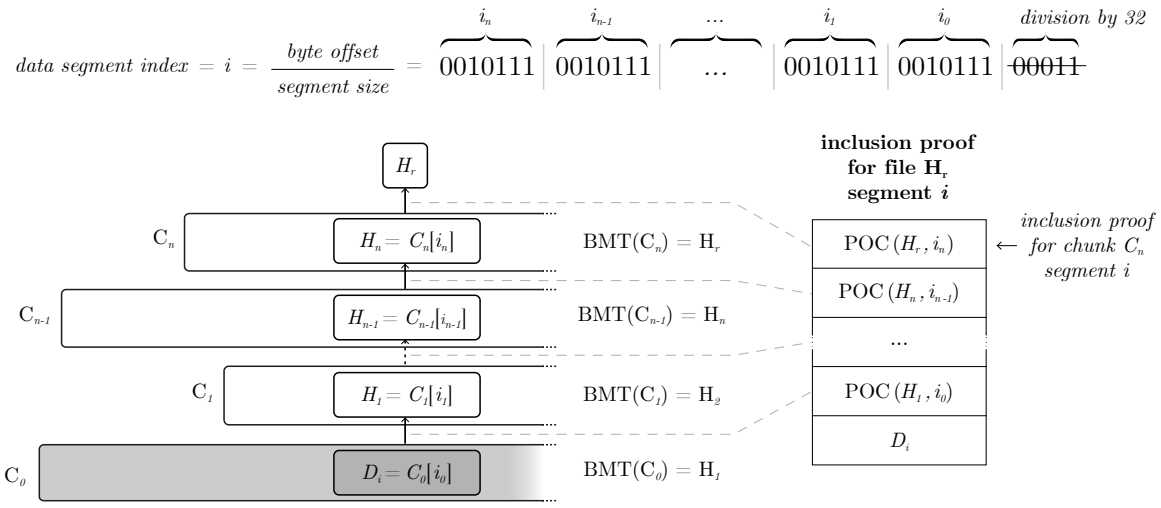

Figure 41: Compact inclusion proofs for files. If we need to prove inclusion of segment i, after division by 32 (within-segment position), we follow groups of 7 bits to find the respective segment of the intermediate node.

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 **??**). 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 42.

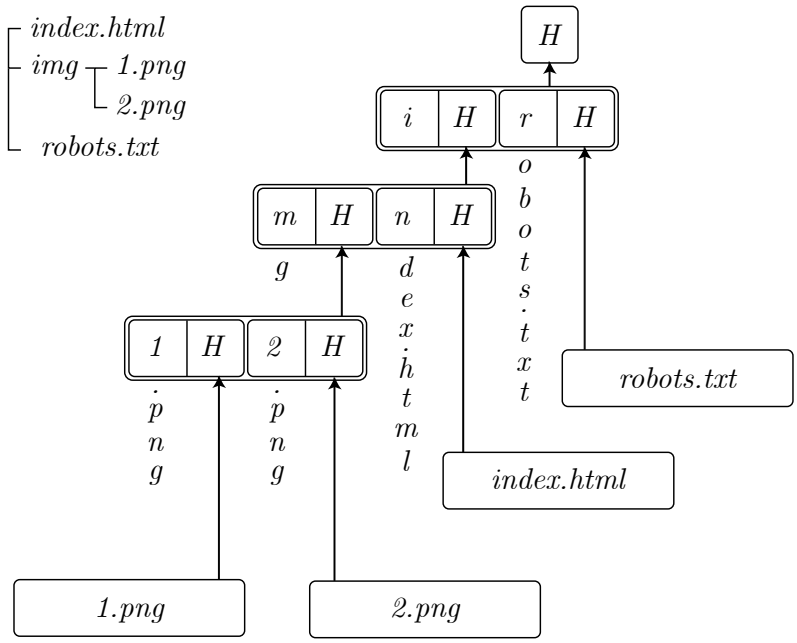

Figure 42: Manifest structure. Nodes represent a generic trie node: it contains the forks which describe continuations sharing a prefix. Forks are indexed by the next byte of the key, the value of which contains the Swarm reference to the child node as well as the longest prefix (compaction).

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 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)

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

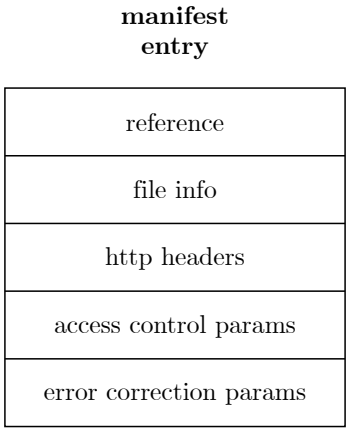

Figure 43: Manifest entry is a data structure that contains the reference to a file including metadata about a file or directory pertaining to the assembler, access control and http headers.

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 **??**) 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 **??**).

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

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

from a full API enabling insert, update and remove through the bzz manifest API (see **??**). 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 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 **??**. 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 **??**) 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 **??**). 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.

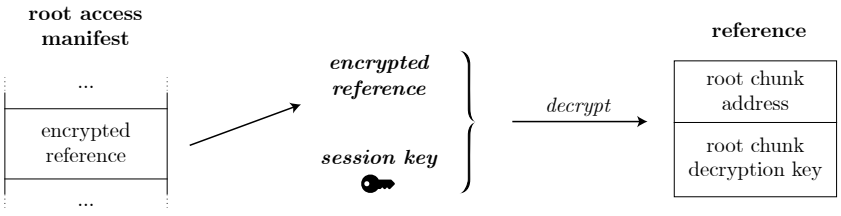

Figure 44: Access key as session key for single party access.

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 44. In the case of multiple parties, an additional mechanism is used for turning the session key into the access key.

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 45 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.

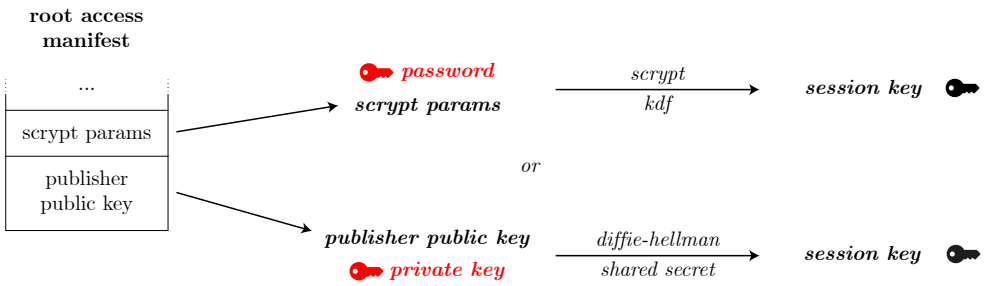

Figure 45: Credentials to derive session key.

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).

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 **??**.

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 46. 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.

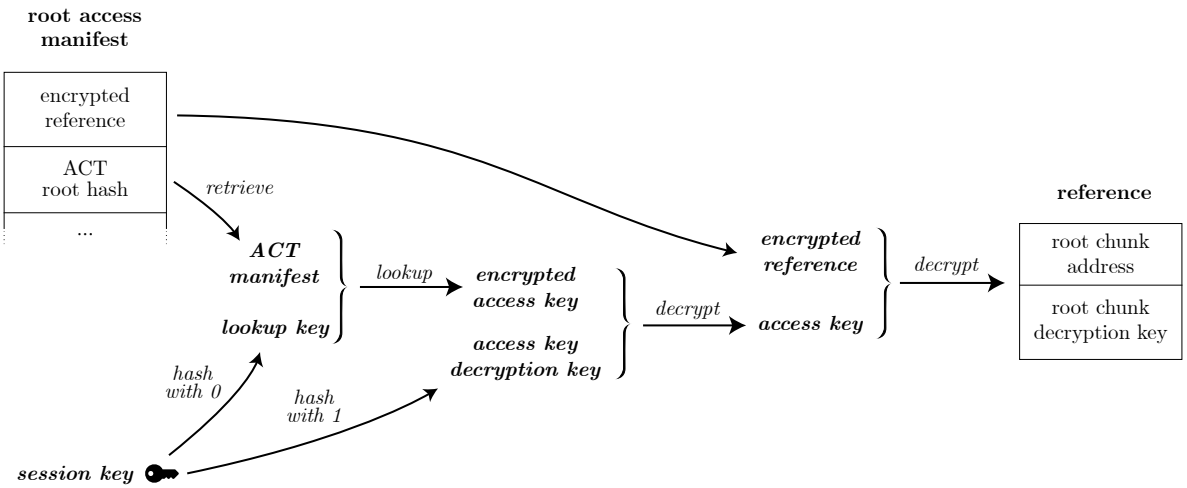

Figure 46: Access control for multiple grantees involves an additional layer to get from the session key to the access key. Each user must lookup the global access key specifically encrypted to them. Both the key to look up and the key to decrypt the access key are derived from the session key which in turn requires their credentials.

- 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.

- 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 **??**

4.3 FEEDS: MUTABILITY IN AN IMMUTABLE STORE

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 47.

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

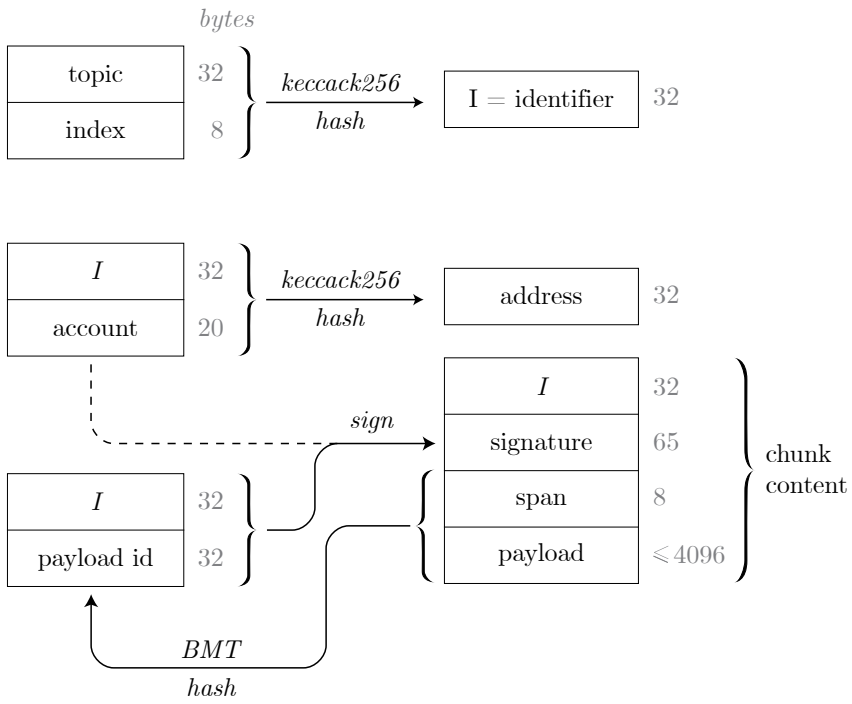

Figure 47: Feed chunks are single owner chunks where the identifier is the hash of the topic and an index. Indexes are deterministic sequences calculated according to an indexing scheme. Subsequent indexes for the same topic represent identifiers for feed updates.

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.

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 48. 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 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

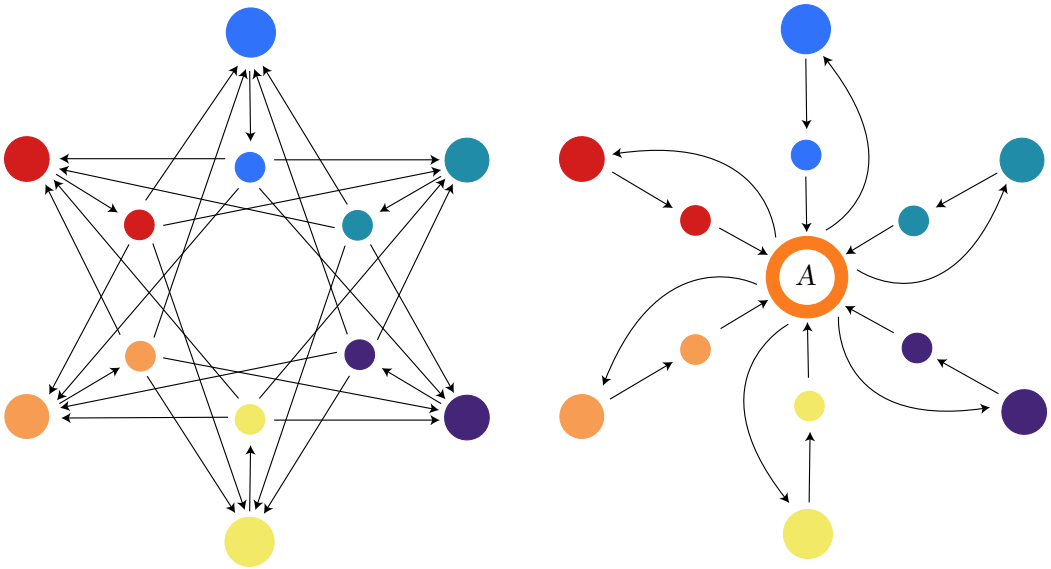

Figure 48: Feed aggregation serves to merge information from several source feeds in order to save consumers from duplicate work. **Left:** 6 nodes involved in group communication (discussing a post, having real time chat, or asynchronous email thread). Each node publishes their contribution as outbox 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.

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.

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.

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.

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 49.

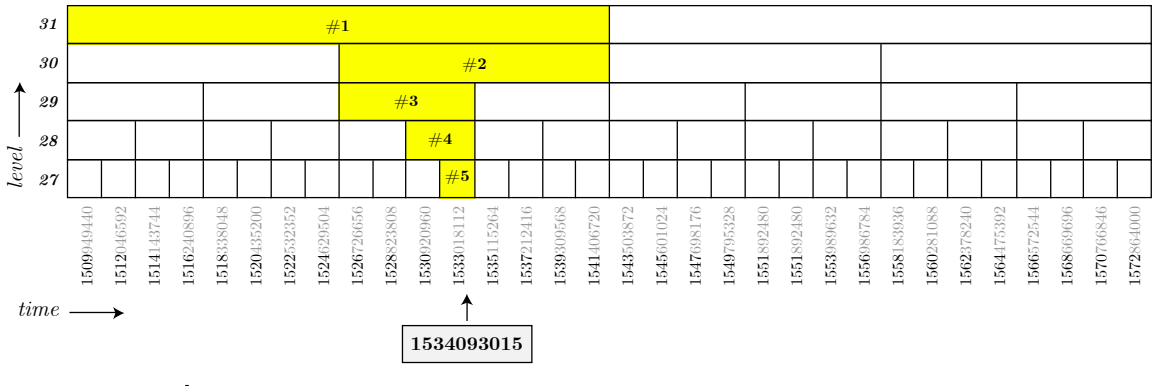

Figure 49: Epoch grid showing the first few updates of an epoch-based feed. Epochs occupied are marked in yellow and are numbered to reflect the order of updates they represent.

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.

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 50. 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.

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 51.

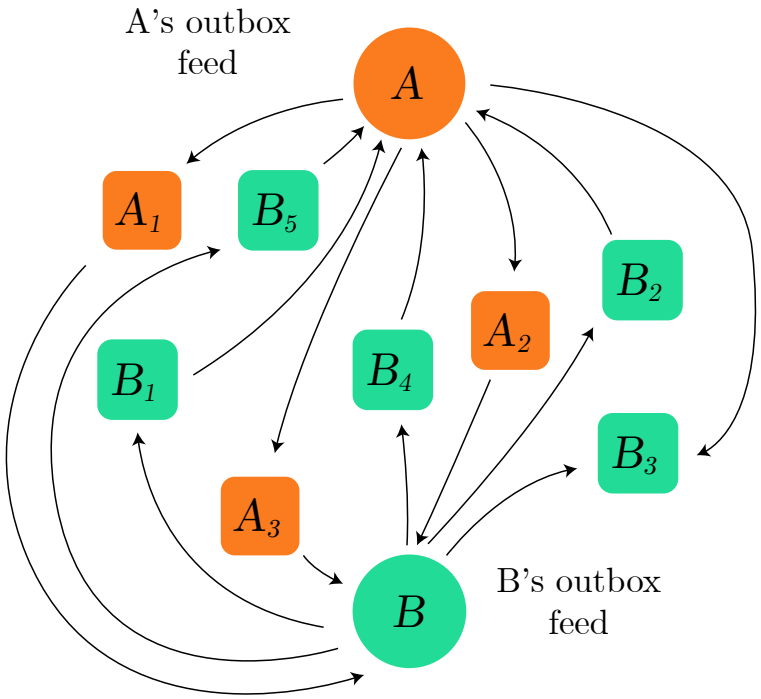

Figure 50: Swarm feeds as outboxes for private communication. Outbox feeds represent consecutive messages from a party in a conversation. The indexing scheme can follow a key management system with strong privacy which obfuscates the communication channel itself and renders interception attacks prohibitively expensive.

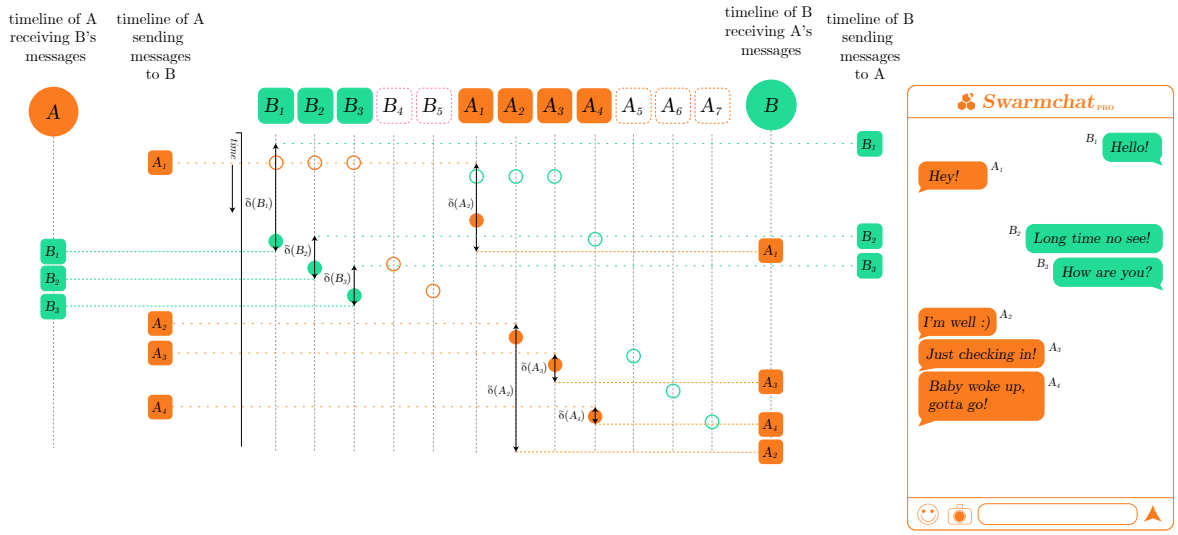

Figure 51: Advance requests for future updates. The diagram shows the time of a series of events during instant messaging between two parties A and B using outbox feeds. The columns designate the neighbourhood locations of the feed update addresses. Circles show the time of protocol messages arriving: 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.

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.

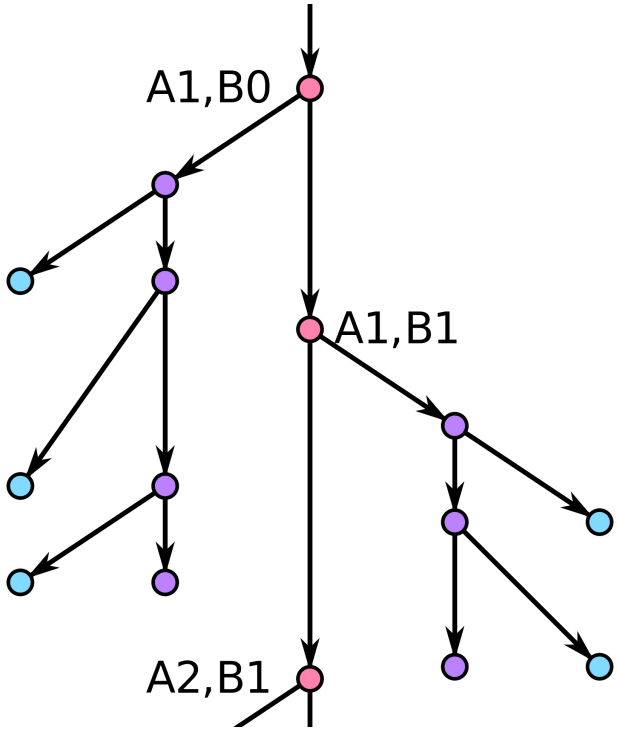

Figure 52: Future secrecy for update addresses

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 52. As a result of this measure the underlying communication channel is obfuscated, i.e. intercepting an outbox update

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

chunk and knowing its index, reveals nothing about previous or 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.

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.

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

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 54:

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.

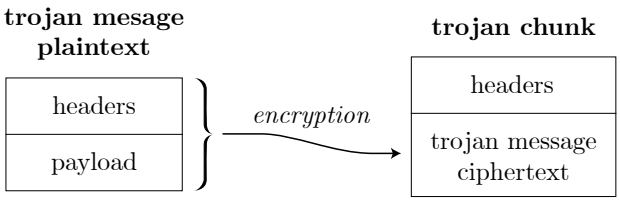

Figure 53: A pss message is a Trojan chunk that wraps an obfuscated topic identifier with a Trojan message, which in turn wraps the actual message payload to be interpreted by the application that handles it.

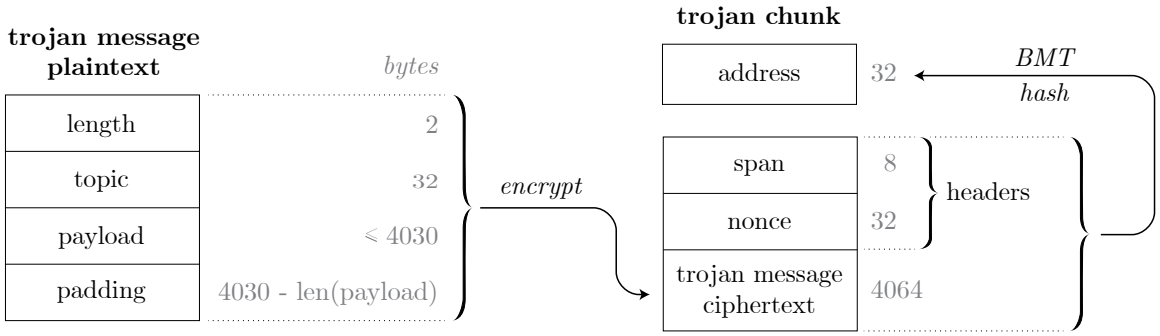

Figure 54: The Trojan chunk wraps an asymmetrically encrypted Trojan message.

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 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 **??**). 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.

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.

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 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.

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–

7 It makes sense to use the postage batch uniformity depth (see 3.3) 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.

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}.

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

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.

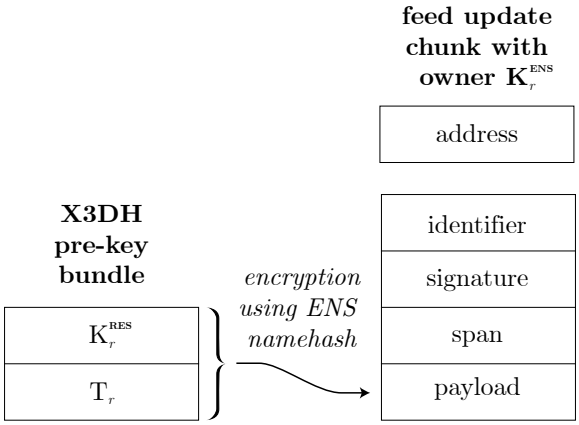

Figure 55: The X3DH pre-key bundle feed update contains the resident key and resident address and is optionally together encrypted with the ENS name hash to prove uniqueness and provide authentication.

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 55.

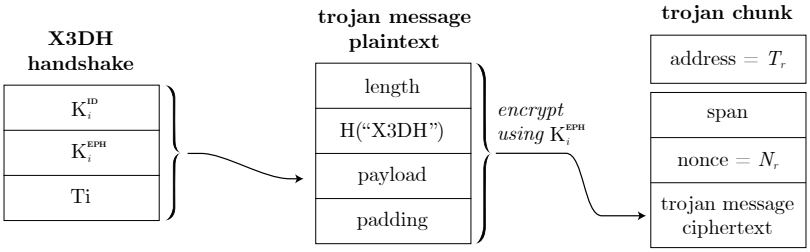

Figure 56: X3DH initial message. Initiator retrieves the ENS owner as well as the latest update of responder's pre-key bundle feed containing the resident key and resident address. Initiator sends their identity key and an ephemeral key to responder's resident address using the resident key for encryption.

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 56) 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.

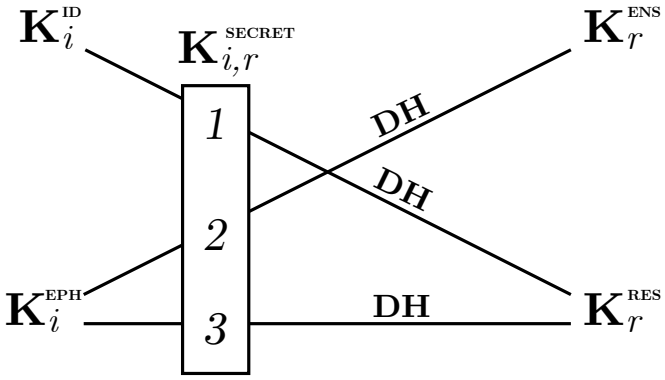

Figure 57: X3DH secret key. Both parties can calculate the triple Diffie-Hellmann keys and xor them to get the X3DH shared secret key used as the seed for the post-handshake protocol.

After the responder receives this information, both parties have all the ingredients needed to generate the triple Diffie-Hellmann shared secret, see figure 57.[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.

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.

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 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 58.

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

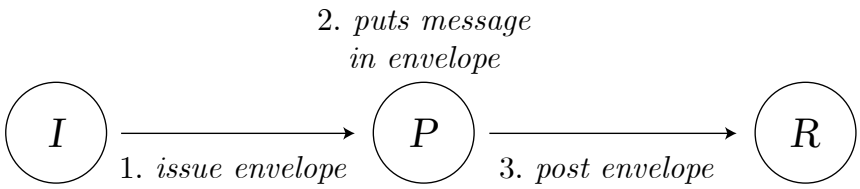

Figure 58: Stamped addressed envelopes timeline of events. Issuer I creates the envelope encrypted for P with an identifier such that P as the single owner of the chunk, produces an address that falls in the recipient R's neighbourhood. As a result (only) P can fill the envelope with arbitrary content and then use simple push-syncing to post it to R.

same account the issuer used as an input when they mined the address. If they do this, the chunk will be valid. See figure 59.

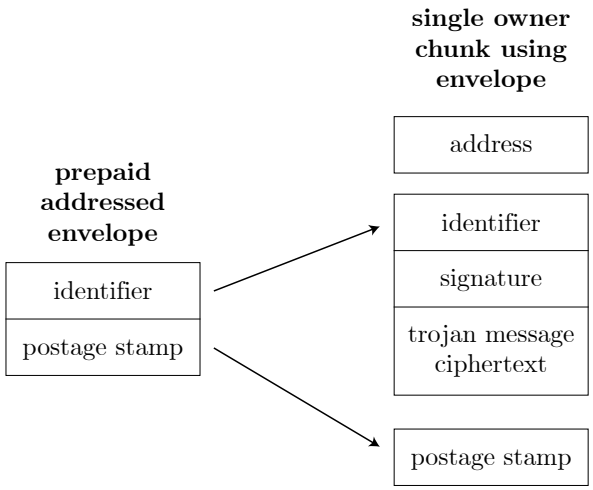

Figure 59: A stamped addressed envelope issued for P and addressed to R consists of an identifier which is mined so that when used to create a single owner chunk owned by P, it produces an address which falls within R's neighbourhood. This allows P to construct a message, sign it against the identifier and using the postage stamp, post it using the network's normal push-syncing to R for free.

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.

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 60. 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.

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 61 for the timeline of events.

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.

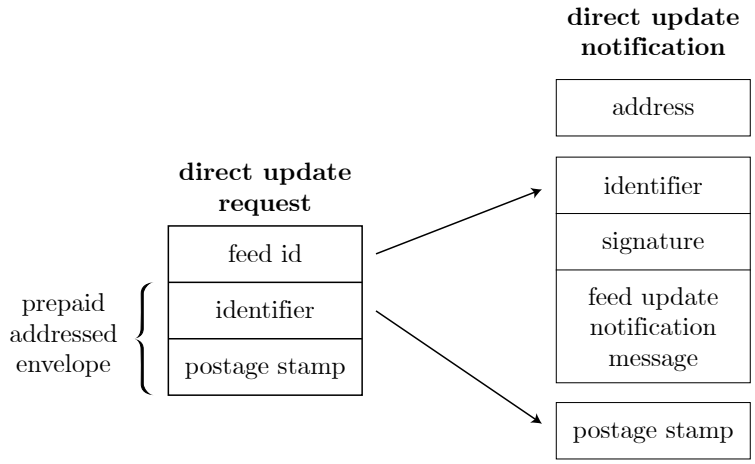

Figure 60: A direct notification request contains a reference to a feed and wraps a pre-paid envelope mined for P (the publisher or a known distributor of the feed) and addressed to recipient R. The response is identical to the process used for generic stamped addressed envelopes and only differs in that the message is supposed to be the feed update or a reference to its content.

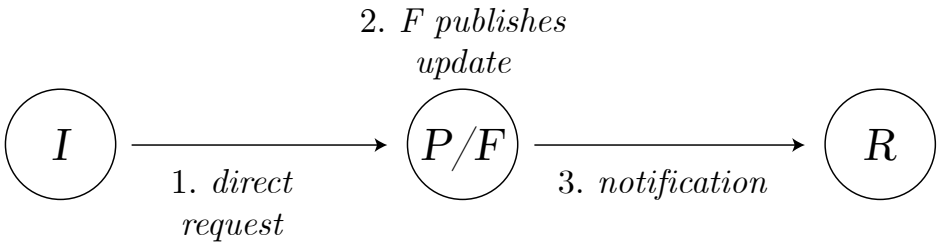

Figure 61: Direct notification from publisher timeline of events. Issuer I constructs a prepaid envelope for the publisher or a known distributor of the feed P/F and addressed to recipient R. Together with a feed topic I, it is sent to P/F wrapped in a pss trojan message. P/F receives it and stores the 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.

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 62 for the structure of neighbourhood notifications and notification requests.

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 63 for the timeline of events when using neighbourhood notifications.

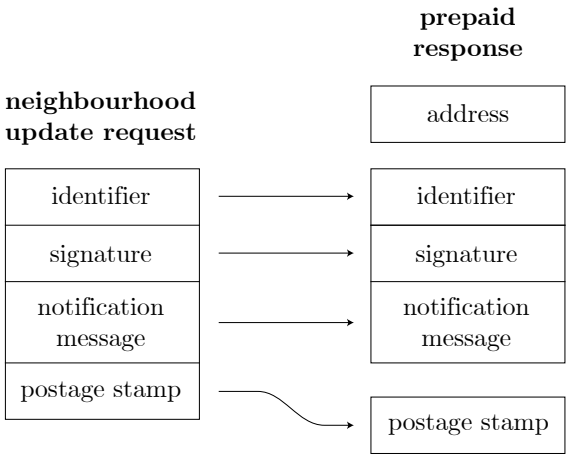

Figure 62: Neighbourhood notification requests are not only including identifier and postage stamp but also a notification message together with the signature attesting it against the address. Thus P needs to construct the actual notification and when publisher F posts their update to P-s neighbour, P can post the notification to recipient R.

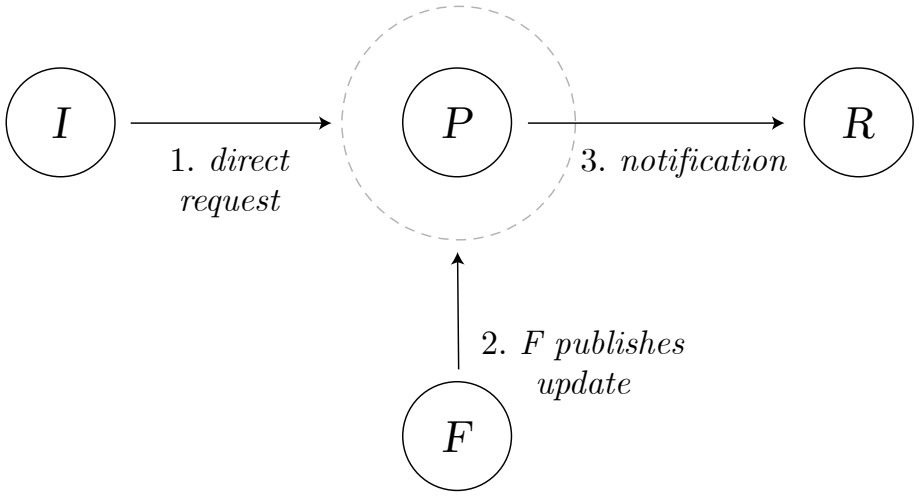

Figure 63: Neighbourhood notification timeline of events. Issuer I mines an identifier for a single owner chunk with themselves as owner such that the chunk address falls in recipient R's neighbourhood. The issuer, being the owner also must sign the identifier against a prefabricated reminder and remember which node should be notified. When P syncs the feed update the notification is sent to R, delivered by the usual push-syncing.

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 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.

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 65. 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,

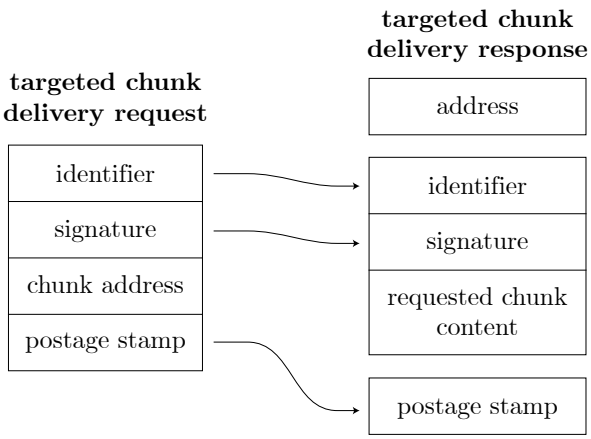

Figure 64: Targeted chunk deliveries are similar to neighbourhood notifications in that they are not only pre-addressed and pre-paid but also pre-signed. Similarly, the identifier is mined so that given issuer I as the owner, it produces an chunk address that falls in recipient R's neighbourhood. Here 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.

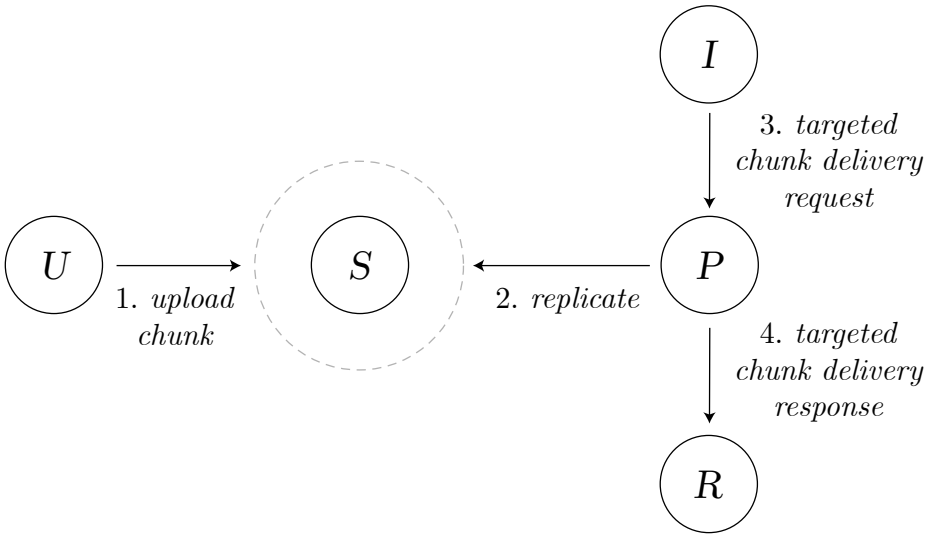

Figure 65: Targeted chunk delivery timeline of events. Uploader U uploads a chunk to Swarm and it lands with a storer node S. Nodes within P replicate (and pin) this chunk. Now, if issuer I wants this chunk delivered to R, it will take the prepaid envelope addressed to R and sends it to the neighbourhood of a known host unencrypted so that anyone who has the chunk can construct the notification and send or store it for later.

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 multiple tartgets, it is sufficient that the initial Trojan request matches any one of these targets.

Table 3 summarises various properties of the three notification-like constructs.

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 3: 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 cross-neighbourhood redundancy 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 through *stewardship*. 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.

In section 5.3, we discuss how an immutable chunk store can support user-controlled deletion of content.

5.1 CROSS-NEIGHBOURHOOD REDUNDANCY: ERASURE CODES AND DISPERSED REPLICAS

First, in 5.1.1, we introduce erasure codes, which we then show how to apply to files in Swarm in 5.1.2. In 5.1.3, we describe a construct that enables cross-neighbourhood redundancy for singleton chunks that completes erasure coding. Finally, in 5.1.4, we show how systematic codes allow multiple retrieval strategies of erasure coded files while preserving random access.

5.1.1 *Error correcting codes*

Error correcting codes are commonly used in the context of data storage and data transfer to guarantee data integrity even in the face of a system fault. Error correction schemes define how to rearrange the original data adding redundancy to its representation before upload or transmission (*encoding*) so that it can correct corrupted data or recover missing content after retrieval or reception (*decoding*). Different schemes are evaluated by quantifying their strength (*tolerance* in terms of the rate of data corruption and loss) as a function of their cost (*overhead* in terms of storage and computation).

In the context of computer hardware architecture synchronising arrays of disks is crucial to provide resilient storage in data centres. *Erasure coding*,in particular, frames the problem as follows: how does one encode the stored data into shards distributed across the disks so that the data remains fully recoverable in the face of an arbitrary probability that any one disk becomes faulty. Similarly, in the context of Swarm's distributed immutable chunk store, this can be reformulated as follows: how does one encode the stored data into chunks distributed across neighbourhoods in the network so that the data remains fully recoverable in the face of an arbitrary probability that any one chunk is not retrievable.[1]

Reed-Solomon coding (RS) [Bloemer et al., 1995, Plank and Xu, 2006, Li and Li, 2013] is the father of all error correcting codes and also the most widely used and implemented.[2] When applied to data of m fixed-size blocks (message of length m), produces an encoding of $m + k$ *codewords* (blocks of the same size) in such a way that having any m out of $m + k$ blocks is enough to reconstruct the original data. Conversely, k puts an upper bound on the number of *erasures* allowed (number of blocks unavailable) for full recoverability, i.e., it expresses (the maximum) *loss tolerance*.[3] k is also the count of *parities*, quantifying the data blocks added in the encoding on top of the original volume, i.e., it expresses *storage overhead*. While RS is, therefore, optimal for storage (since loss tolerance cannot be higher than storage overhead), it has high bandwidth demands[4] for a local repair.[5] The decoder needs to retrieve m chunks to recover a

1 It is safe to assume that the retrieval of any one chunk will fail with equal and independent probability.
2 For a thorough comparison of an earlier generation of implementations of RS see [Plank et al., 2009].
3 Error correcting codes that has a focus on correcting data loss are referred to as *erasure codes*, a typical scheme of choice for distributed storage systems [Balaji et al., 2018].
4 Both the encoding and the decoding of RS codes takes $O(mk)$ time (with m data chunks and k parities). However, we found computational overhead both insignificant for a network setting as well as undifferentiating.
5 Entanglement codes [Estrada-Galinanes et al., 2018, Estrada-Galinanes et al., 2019] require a minimal bandwidth overhead for a local repair, but at the cost of storage overhead that is in multiples of 100%.

particular chunk unavailable. Hence, ideally, RS is used on files which are supposed to be downloaded in full,[6]but are inappropriate for use-cases needing only local repairs.[7]

When using RS, it is customary to use *systematic* encoding, which means that the original data forms part of the encoding, i.e., the parities are actually added to it.

5.1.2 *Erasure coding in the Swarm hash tree*

Swarm uses the *Swarm hash tree* to represent files. This structure is a Merkle tree [Merkle, 1980], whose leaves are the consecutive segments of the input data stream. These segments are turned into chunks and are sent off to the Swarm nodes to store. The consecutive chunk references (address or address and encryption key) are written into a chunk on a higher level. These so called *packed address chunks* (PACs) constitute the intermediate chunks of the tree. The branching factor b is chosen so that the references to the children fill up a full chunk. With reference size 32 or 64 (hash size 32) and chunk size 4096 bytes, b is 128 for unencrypted and 64 for encrypted content (see figure 66).

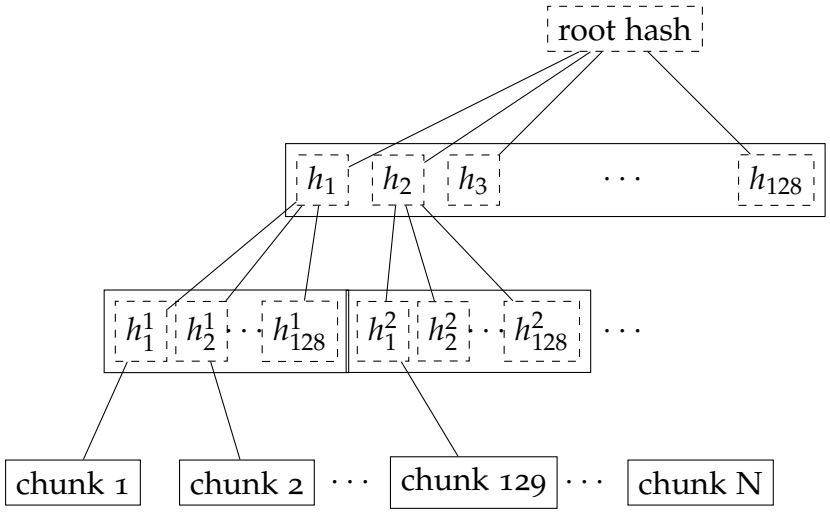

Figure 66: The Swarm tree is the data structure encoding how a document is split into chunks.

Note that on the right edge of the hash tree, chunks (the last chunk of each level) may be shorter than 4K: in fact, unless the file is exactly $4 \cdot b^n$ kilobytes long, there is always at least one *incomplete chunk*. Importantly, it makes no sense to wrap a single

6 Or, in fragments large enough to include the data span over which the encoding is defined, such as videos.

7 Usecases requiring random access to small amounts of data (e.g., path lookup) benefit from simple replication to optimise on bandwidth, which is suboptimal in terms of storage [Weatherspoon and Kubiatowicz, 2002].

chunk reference in a PAC, so it is attached to the first level where there is open chunks. Such *"dangling" chunks* will appear if and only if the file has a zero digit in its b-ary representation.

During file retrieval, a Swarm client starts from the root hash reference and retrieves the corresponding chunk. Interpreting the metadata as encoding the span of data subsumed under the chunk, it decides that the chunk is a packed address chunk if the span is larger than the maximum chunk size. In case of standard file download, all the references packed within the PAC are followed, i.e., the referenced chunk data is retrieved.

PACs offer a natural and elegant way to achieve consistent redundancy within the swarm hash tree. The input data for an instance of erasure coding is the chunk data of the children, the equal sized bins corresponding to chunk data of the consecutive references packed in it. The idea is that instead of having each of the b references packed represent children, only m would, and the rest of the $k = b - m$ would encode RS parities (see figure 67).

The *chunker* algorithm incorporating PAC-scoped RS encoding would work as follows:

1. Set the input to the actual data level and produce a sequence of chunks from the consecutive 4K segments of the data stream. Choose m and k such that $m + k = b$ is the branching factor (128 for unencrypted, 64 for encrypted content).
2. Read the input one chunk at a time. Count the chunks by incrementing a counter i.
3. Repeat step 2 until either $i = m$ or there's no more data left.
4. Use the RS scheme on the last $i \leq m$ chunks to produce k parity chunks resulting in a total of $n = i + k \leq b$ chunks.
5. Concatenate the references of all these chunks to result in a packed address chunk (of size $h \cdot n$ of the next level) of the level above. If this is the first chunk on that level, set the input to this level and spawn this same procedure from step 2.
6. If the input is consumed, signal end of input to the next level and quit the routine. If there is no next level, record the single chunks as the root chunk and use the reference to refer to the entire file.

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.

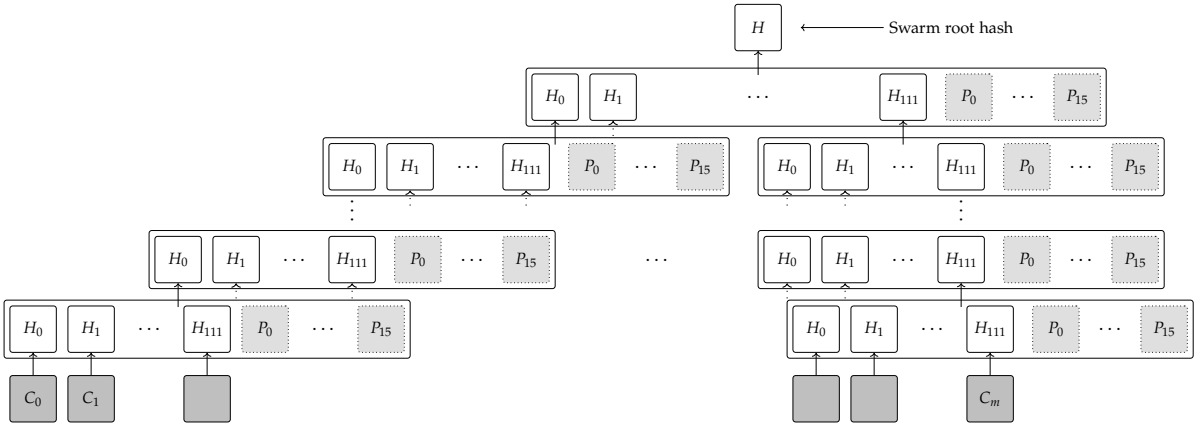

Figure 67: The Swarm tree with extra parity chunks using $m = 112$ out of $n = 128$ RS encoding. Chunks p_0 through p_{15} are parity data for chunks H_0 through H_{111} on every level of intermediate chunks.

5.1.3 *Incomplete chunks and dispersed replicas*

If the number of file chunks is not a multiple of 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.[8] 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 would need to replicate the chunk in such a way that (1) ideally the replicas are dispersed in the address space in a balanced way, yet (2) their address can be known by retrievers ideally only knowing the reference to the original chunk's address.[9]

The solution is to reference the root hash with its BMT root hash (before it is hashed with the metadata). By making the metadata fully or partially nonsemantic, we virtually obtain the degrees of freedom to generate entropy to mine the overall chunk

8 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.

9 We could create replicas by having the chunk data 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. This has the disadvantage that replicas are not reuploadable by parties other than the publisher.

hash into a sufficient number of neighbourhoods. This construct is called dispersed replicas and the exact method for finding them is as follows:

Let's now assume h is the BMT roothash of the chunk. Given n bits available to find a nonces to generate 2^k perfectly balanced replicas, initialise a chunk array ρ of length 2^k and start with $i = 0$ and $C = 0$.

1. create the metadata by copying onto the n available bits the bits of the big endian binary encoding of i resulting in the metadata bytes m.
2. calculate the the chunk hash by prepending m to h and use the Keccak256 base hash to hash them $a_i := H(m \oplus h)$, and record $c_i = CAC(d, m)$
3. calculate the bin this hash belongs by decoding k-bit prefix as big endian binary number j between $0 \leq j < 2^k$.
4. if $\rho[j]$ is unassigned, then let $\rho[j] := c_i$ and increment C.
5. if $C = 2^k$, then quit
6. increment i, if $i = 2^n$ then quit
7. repeat from step 1

With this solution we are able to provide an arbitrary level redundancy for storage of data of any length. Note that if n is small, then generating all 2^k balanced replicas may not be achievable, if $n < k$ this is certainly not possible. In general given n, k at least m miss has probability $\left(\frac{2^k - m}{2^k}\right)^{2^n}$.

5.1.4 *Strategies of retrieval*

When downloading, systemic per-level erasure codes allow for 3 strategies of use:

— *direct with no recovery* – For all intermediate chunks, the RS parity chunks are ignored. Retrieval is involving only the original chunks, no recovery. This is the cho ice for random access.
— *selective with fallback* – For all intermediate chunks, first retrieve m chunks judged to be the fastest to download (e.g., the m closest to the node), and then fall back to $j \leq k$ if and only if retrieval of j out of m chunks is unavailable. This saves some bandwidth and the corresponding costs at the expense of speed: if a selected chunk is missing, fallback to RS is delayed.
— *race* – the client initiates requests for all $m + k$ and will need to wait only for the first m to be delivered in order to proceed. This is equivalent to saying that the k slowest chunk retrievals can be ignored, therefore this strategy is optimal for latency at the cost of cost.

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 DATA STEWARDSHIP: PINNING, REUPLOAD AND RECOVERY

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 **??**).

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.[10]

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 **??** while related data structures are formalised in **??**.

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

10 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.

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 **??**).

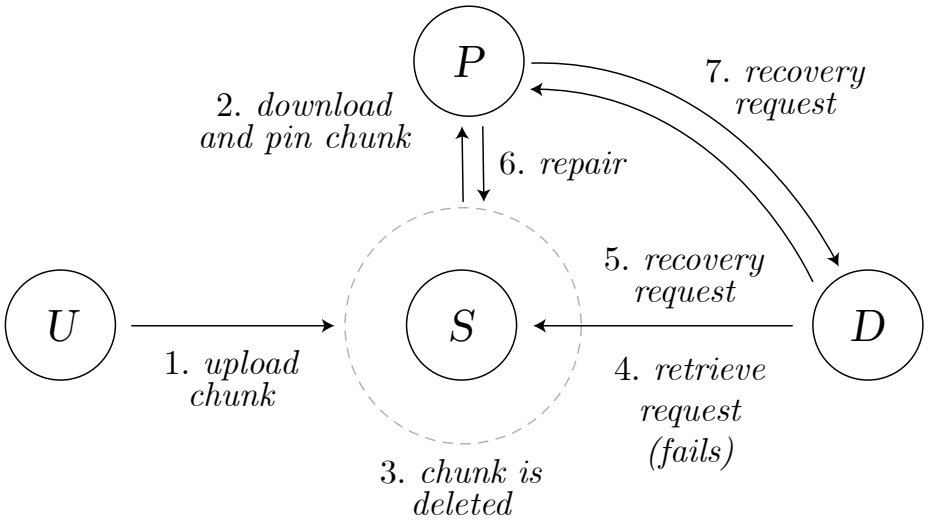

Figure 68: The missing chunk notification process is similar to the targeted chunk delivery. Here a downloader mines the identifier to match their own address, i.e. a self-notification. If downloader *D* encounters a missing chunk (a request times out), they send a recovery request to one of several 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.

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 69.

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.

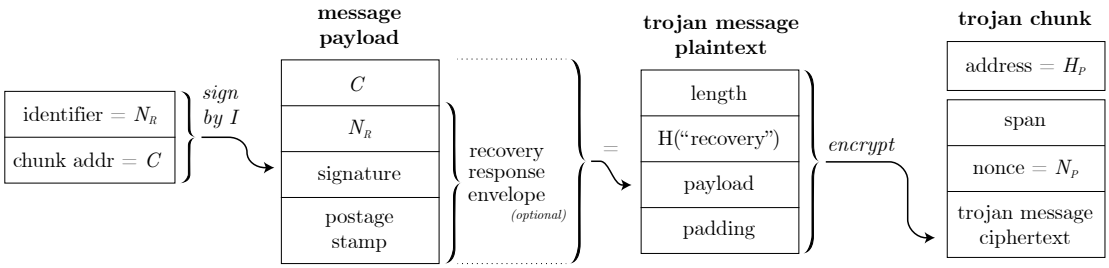

Figure 69: A recovery request is a Trojan chunk that is used as missing chunk notification. It is unencrypted and its payload is structured as a pss message containing the address of the chunk to be recovered. Optionally, it also includes a special recovery response envelope, an identifier with a signature attesting to the association of the identifier and the missing chunk hash.

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.

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.

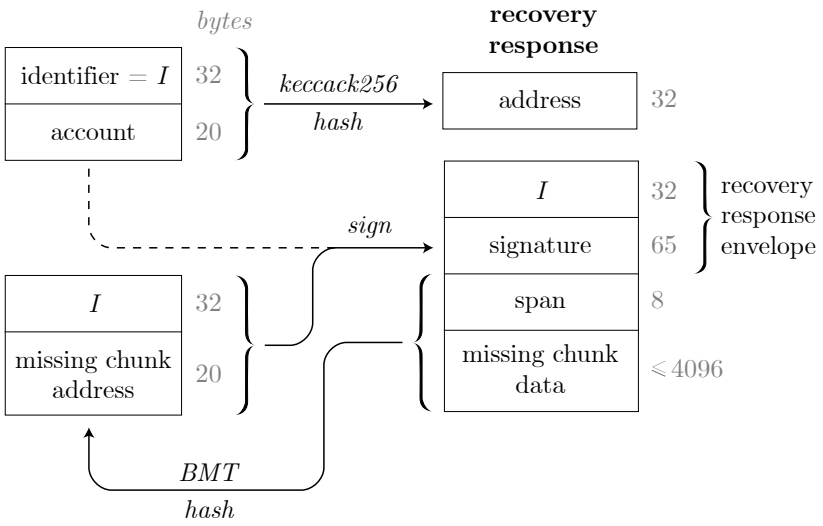

Figure 70: Recovery response is a single owner chunk that wraps the missing chunk. Anyone in possession of the headers and the chunk data is able to construct a valid single owner chunk and send it to the downloader for free.

5.3 DREAM: DELETION AND IMMUTABLE CONTENT

The increasing trend of digitisation and the ensuing ubiquity of personal data in the cloud touch on sensitivities more than ever before. In response to the former, regulators often formulate requirements on removability that fails to recognise that it is virtually impossible to make material once seen become unseen. Entities that operate infrastructure underpinning digital publishing are in control of the technology serving the content and are able to implement "removal" by denying access. Such centralised gate-keeping has its regulatory appeal as it can be thought of as an instrument of law enforcement. Although such intervention rarely solves the breach of unrealistic privacy rights, the gate-keeper provides a well-defined responsible party with the effective course of action: not serving censored content. While this gives the unwary a phony sense of security, it only exemplifies the bigger problem of censorship: publishing platforms have the technological means to filter content and centralised control makes exerting influence on content cheaper. What starts out as sensible measures of content curation with time becomes extant censorship. This is all the more problematic with social platforms that gained quasi-monopolistic status due to netwtork effects. As a result of high switching cost, innocent content creators ever more often end up being 'deplatformed'. Similarly, the ability to identify hosts and deny access to content via legal decree gives powers to be the means to infringe on freedom of speech.

In the decentralised paradigm of web 3, there is no longer an operating entity behind either the publishing platform or the hosting infrastructure. Therefore, the cost of censorship is raised beyond the realm of feasibility. And yet, the potential permanent exposure of personal data is perceived as scary to most and some novel solution is called for to restore the sense of security gate-keepers purportedly represent.

5.3.1 *Deletion and revoking access*

First, notice that any reference to removal of information in the sense of erasing it from all physical storage devices is both unenforceable and impractical. Even the strictest data protection audits do not require the erasure of offending data from backup tapes. In general, the tacit assumption is that information ordered to be removed should become inaccessible through *typical precedented methods of access*.

In what follows we formulate the strongest meaningful definition of deletion applicable to decentralised storage and offer a construction that implements it. Importantly, such a construction is purely technical, referring to primary actors' capabilities and costs as opposed to procedural, referring to intermediaries' obligations to respect the rights of primary actors.

Primary actors here are the *uploader* that wishes to share content by granting read access to a number of parties, called *downloaders*. Granting access is defined as providing some canonical *reference* to the content that the system can eventually resolve to the full information meant to be disclosed. Any party that is privileged to access this information, is able to store, re-code and potentially disseminate it. This provides a myriad of ways to make accessible the content at any later point in time to get round whatever process would qualify as deletion or removing access. There is, by definition, no protection possible against such adversity. As a consequence, any legally sound notion of deletion (retrospective revocation of access) is meant to be interpreted in a narrower sense: *the viability to replay the same access method* for a cost lower than the *full cost* of storing the content.[11]

Let us now define deletion as a scheme for uploading content with access revocation that has the following requirements:

11 That is, the total cost of storage paid for the full size for the entire period kstarting from the time access was revoked uptill the attempted breach.

– *specialisation* – Uploader is able to choose at the time of publishing a specialised construct[12] allowing retrospective revocation of access.
– *sovereignty* – Uploader is the sovereign owner of the data and is in sole possession of the means to selectively revoke access from any party previously granted access.[13]
– *security* – After revocation of access, a grantee is unable to access the content using the same reference that the access was granted with, or any other cue shorter than the deleted content.[14]

Note that normally uploaded content may be *forgotten*: if nobody pays for storing it and it is not accessed sufficiently often, the chunks constituting it will be garbage collected. However, chunks with expired postage stamps cannot be regarded as reliably deleted to satisfy the requirements of sovereignty and security.

5.3.2 *Construction*

The goal of this section is to arrive at a formal construction of a DISC-based revocable access model. We will restrict our scheme to *chunks*, the fundamental fix sized storage unit of Swarm's DISC model. The proposed *dream* construct implements a deletable content storage and access model conforming to the requirements of specialisation, sovereignty, and security. The use of the word 'dream' alludes to the somewhat unexpected finding that such a construct is even possible in the immutable DISC model. On top of this, as customary in swarm, it serves as a mnemonic acronym, resolving to the 5 *dream attributes* expressing requirements of access control.

D *deniable* – the dream key serves as a one time pad for decryption. Since multiple content chunks (in fact any arbitrary content) can use the same dream pad, the key's association to any content is plausibly deniable.

R *revocable* – access granted by dream keys is revocable. Revoking access from all parties including oneself is taken as deletion.

E *expirable* – the scheme allows for one-time use, i.e., the key can only be retrieved once.

12 From a user's perspective, content that is meant to be reliably deleted should be uploaded as such. The costs of uploading such content is allowed to be a (small integer) multiple of the cost of regular, censorship-resistant but non-deletable uploads.

13 The uploader's credentials are necessary to delete their own deletable content. Deletables must also allow access control, i.e., only available to a specific set of recipients.

14 In other words, if the downloader has *not* stored at least as much information as the deleted information itself, they will have no way of retrieving it.

A *addressable* – access can be granted to a neighbourhood, only clients operating a node with an overlay address in a given range are able to access.

M *malleable* – the construct is resilient to churn and dynamic change of network size, is reusable across independent grantees and upgradeable.

The scheme is built on top of DISC's APIs and can be implemented entirely as a second layer solution. Despite its rich feature set, the scheme does not use complex cryptography, but relies on the interplay of various component subsystems.

Assume we have a pseudo-random deterministic function that generates a longer (e.g., chunk-sized = 4K) sequence c from a key-sized (32 byte) generator g.[15]

The central construct called a *dream* is a chunk-sized one time pad which acts as a decryption key for the deletable content. The dream chunk is collectively created by a set of nodes following a network protocol. Each node in the sequence of participating peers receives a piece of data and uses it as input together with data present in their reserve to produce an output sent to the neighbourhood of the next node in the chain. As long as each node in the chain performs the same calculation and forwards the same result, the same one time pad can be construed and ultimately the deletable content can be retrieved and read.

The insight here is that retrieving a deletable chunk's references involves calculations using a set of immutable chunks under the control of the uploader. The uploader's ability to change this underlying set holds the key to providing the necessary mutability required by any notion of deletion.

Let b be the batch ID (a 32-byte hash) of a postage batch owned by U and let $Chunks(\gamma, b, p)$ stand for all chunks stamped by b at block γ in the bucket designated by the pivot p. Define $\overline{\chi}(\gamma, b, p)$ as the chunk stamped by batch b in bucket designated by p older than γ whose address is closest (has minimal XOR distance) to pivot p.

$$\overline{\chi} \quad : \quad \Gamma \times Batches \times Segment \mapsto Chunks \tag{1}$$

$$\overline{\chi}(\gamma, b, p) \quad \stackrel{\text{def}}{=} \quad \underset{c \in Chunks(\gamma,b,p)}{\arg\min} \; \chi(p, \textsc{Address}(c)) \tag{2}$$

15 For example, the Keccak sponge function used throughout Ethereum for hashing does have this capability. Alternatively, the blockcipher encryption using initial nonce g could be applied to a fixed constant chunk such as all zeros.

If batch b is underutilised, a bucket designated by p may not contain any chunks belonging to the batch bucket. In such cases there is no chunk closest to p, thus the *BBB* function is undefined.

Let us now define the OTP update function $\Delta(\gamma, b)$ for batch ID and input address p as follows:

$$\begin{aligned} \Delta &: \Gamma \times \mathit{Batches} \mapsto \mathit{Chunks} \mapsto \mathit{Chunks} && (3)\\ \Delta(\gamma, b) &: \mathit{Chunks} \mapsto \mathit{Chunks} && (4)\\ \Delta(\gamma, b)(c) &\stackrel{\text{def}}{=} \mathcal{G}[4K](\textsc{Address}(\overline{\chi}(\gamma, b, c))) && (5) \end{aligned}$$

Since uniformity depth of batches is meant to be larger than nodes' storage depth, all chunks in a batch bucket designated with p is in the reserve of every node in the neighbourhood designated by p.

If swarm grows and there are nodes whose storage depth is higher than the batch's uniformity depth, then the neighbourhood designated by p may not contain all the chunks in the bucket, thus compromising the computability of the OTP update function.

As long as 1) the bucket of batch b designated by p is not empty and 2) the neighbourhoods include (one or more) buckets then the update function is computable by any storer node within the neighbourhood designated by p.

Since the task function can be applied to its own output, we can define *dream path*:

$$\begin{aligned} \Pi(b, g) &\stackrel{\text{def}}{=} c_0, \ldots, c_n \in \mathit{Chunks}\{n\} && (6)\\ c_i &\stackrel{\text{def}}{=} \begin{cases} \mathcal{G}[4K](g) & \text{if } i = 0 \\ \Delta(b, c_{i-1}) & \text{otherwise} \end{cases} && (7) \end{aligned}$$

The dream path function is a pseudo random generator defined by finite recursion. A *dream* is a particular pairing of a generator and a one time pad that are input and output of a dream path. Given a dream path of length n and grantee overlay address a and a batch b with uniformity depth d, uploader needs to find the generator g, such that the n-th chunk of the dream path should land in a's neighbourhood (i.e., $PO(a, \textsc{address}(c_{n-1})) \geq d$).

$$
\begin{aligned}
dream(b, n, a) &\subset Keys \times Chunks && (8)\\
\langle g, k \rangle &\in dream(b, n, a) && (9)\\
&\iff && (10)\\
&k = \Pi(b, g)[n] \wedge && (11)\\
&PO(a, H_{bmt}(k)) \geq d && (12)
\end{aligned}
$$

The uploader is able to construct the dream pad. As a result they can also calculate k given g. Since $H(k)$ is uniform, the chance of $PO(a, H(k)) \geq d$ is 1 in 2^d.

Now we can turn to the definition of a dream, a network protocol based access method that is deniable, revocable, expirable, addressable, malleable. In order for downloaders to calculate the dream pad, they must rely on the network, whereby each recursive step is calculated by a node in the neighbourhood designated by the respective input chunk's address.

In order to guarantee the correct termination, the following criteria must be fulfilled:

— all buckets of batch b designated by $p_0, \ldots, p_{n-1}$ amust be non-empty
— uniformity depth of the batch must remain higher than the storage depth of nodes in the neighbourhoods designated by $p_0, \ldots, p_{n-1}$.
— the output of each step produced by a node in the neighbourhood designated by p_i must be sent to the neighbourhood designated by p_{i+1}.
— nodes in the neighbourhoods on each hop of the dream path (designated by $p_0, \ldots, p_{n-1}$) must be incentivised to compute the OTP update as well as send off the resulting output chunk to the next neighbourhood.

Figure 71: The dream protocol

Let us define a network protocol *dream* (see figure 71). Those who play the protocol, listen to single owner chunks wrapping the input. We assume that they extract the batch identifier b, and the payload CAC c as parameters to the OTP update function and then calculate the output pad, wrap it as a dream chunk and send it to the network towards the address of the output chunk so that the destination neighbourhood can calculate the next step. If the protocol is followed up to n steps, then $k = c(n)$ is received by the target node at address a.

We now turn to the construction of dream chunks. Uploader U wishes to grant downloader D (at overlay address a) revocable access to content chunk C. U chooses a postage batch b it owns that is completely filled, a step-count n, and a depth d. U creates a dream generator $\langle g, k \rangle \in dream(b, d, n, a)$.

U calculates the "ciphertext" $C' = C \veebar k$ and uploads it to swarm obtaining $r = H(C')$ as parity reference. Now U creates the *dream reference* as $ref(C) = \langle r, b, g \rangle$ which can be given to the downloader grantee D.[16]

5.3.3 *Analysis*

D in possession of a dream chunk reference as $\langle r, b, g \rangle$ calculated by U as $ref(C)$ constructs $p = \mathcal{G}[4K](g)$ and wraps it as a dream chunk and sends it to swarm. When it receives the dream chunk for b, it extracts payload k.

D retrieves the parity chunk C' using reference r and then decodes $C = C' \veebar k$. The retrieval process is trivially correct as long as

— The parity chunk C' is retrievable via normal retrieval.
— The dream protocol is adhered to by cooperating nodes.
— The batch bucket bottom has not changed for the neighbourhoods that are on the dream path.

We are now turning to access revocation. Deletion of content is equivalent to revoking all access.

Uploader U had granted D (at address a) access to C through the dream reference $\langle r, b, g \rangle$. U revokes access from D by uploading extra chunks to batch b such that $\overline{\chi}$ changes.

As long as there is a single honest neighbourhood walked by the protocol, such that the nearest chunk to pivot p from batch b changes, then D can no longer retrieve C.

To prove this, first let us assume that there is a bucket designated by some p_i on the dream path such that the $\overline{\chi}$ for the bucket changed since uploader constructed the dream. When the participating node in the neighbourhood designated by p_i calculates the OTP update function, the result will be completely different and the dream chain will not terminate.

16 So far, references to swarm content included an address (*Swarm hash* for unencrypted content) or an address with a decryption key (for encrypted content). The dream reference now defines a third type of reference, serialised in four segments comprising the parity address, the batch id, the initial generator and a decryption key.

5.3.4 *Security*

The reference does not leak either the step-count or any other information about the neighbourhoods touched. Neither do the participants in the protocol know which position they are at in the chain or anything about the other neighbourhoods except for the following one to which they are push-syncing their chunk.

The construction is deniable. Beside our sensitive content C, take A, any uncontroversial content chunk. When producing C', the owner also produces $A' = A \veebar k$ and upload it. When asked about k, producing A makes the denial of other content including C more plausible.

A powerful adversary can infiltrate every neighborhood of Swarm and archive all information that has ever been uploaded (without being able to decipher it surely) and keep logs of what has been uploaded in what order, which would allow it to serve up deleted content on demand. But there is a huge price on such indiscriminate archiving of all Swarm's content and that is the only way to reliably defeat the dream construction.

We now turn to the discussion how to calibrate the step-count given a security model. Let us assume a network-wide neighbourhood infiltration rate of $\frac{1}{2}$, meaning that half of the neighbourhoods in the network is assumed to be malicious and colluding.

Assuming that when revoking access, the owner uploaded new chunks in each neighbourhood on the dream path. Given a particular dream chain, However, if even a single neighbourhood on the dream path is honest they respect the newly arrived chunks and divert the dream path, making it never terminate with the grantee.

Thus for a breach of access to happen, all neighbourhoods must be malicious. In our security model a neighbourhood is malicious with uniform and independent probability. For an overall infiltration rate of 1 out of k, the chance of all neighbourhoods on a given random dream path being malicious is k^{-n}. For a security requirement of success rate of σ "nines", i.e error rate less than $10^{-\sigma}$, we can formulate the requirement as

$$\frac{1}{k^n} < \frac{1}{10^{-\sigma}} \tag{13}$$

Now, expressing k as 10^{κ}, taking the logarithm of both sides and multiply by -1, we get

$$n > \frac{\sigma}{\kappa} \tag{14}$$

With one in every 10 neighbourhoods being malicious, the dream path must have as many hops as the number of nines expressing the success rate.[17]

17 I.e., with malicious node probability 0.1, a 6-hop-long path will effectively revoke access wtith certainty of 99.9999%.

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.

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 how long it will take for the process to complete. Tags also record partial root hashes which can then be used to resume an upload in the event it is interrupted before completion. 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 **??**. Requests can specify Swarm specific options such as:

- `tag` – use this upload tag – generated and returned in response header if not specified.
- `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 of a randomly generated key.
- `pin` – pin all chunks of the upload if set.
- `parities` – apply RS 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 **??**.

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, 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 **??**). 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 **??**.

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 **??**). 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 `rs` attribute to the number of parity chunks (see **??**).

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. `rs` 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 **??**, 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 **??**). 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 **??**. The `bzz` URL scheme provides the `access` API endpoint for access control (AC, see the API specification in **??**). 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 **??** 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 **??**, 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 **??** 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:

- `rs` – structure with attributes needed to use RS 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 `rs` attribute is given, there are two strategies to follow:

- *fallback* – For all intermediate chunks, the RS 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 RS is delayed.
- *race* – for all intermediate chunks, the RS parities are retrieved right away together with the rest of the children chunks. The first n (assuming k as value of `rs`, 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-RS-STRATEGY=fallback` or completely disabled by setting `SWARM-RS-STRATEGY=disabled`. The number of parities for an erasure coded batch is taken from the enclosing manifest's `rs` attribute but can be overridden with the header `SWARM-RS-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 **??**). 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.

6.3 COMMUNICATION

Somewhat surprisingly, Swarm's network layer can serve as an efficient communication platform with exceptionally strong privacy properties. This chapter aims to define a comprehensive set of primitives that serve as the building blocks for a foundational communication infrastructure, covering the full range of communication modalities including real-time anonymous chat, sending and receiving messages from previously unconnected and potentially anonymous senders, mailboxing for asynchronous delivery, long-term notifications, and publish/subscribe interfaces.

Swarm core offers the lowest level entry-points for communication-related functionality:

- pss offers an API for sending and receiving trojan chunks
- bzz offers a way to upload single owner chunks. Single-owner chunk retrieval requires nothing beyond what is provided by the storage APIs

Since trojan message handling differs significantly from storage operations, it deserves its own URL-scheme called `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 should match the trojan message address (see 4.4.1).

Destination targets are represented as the buzz serialisation of the target prefixes as a file. The URL path points to this file or is a top-level domain. Thtopic is specified 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.

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.

[Balaji et al., 2018] Balaji, S., Krishnan, M. N., Vajha, M., Ramkumar, V., Sasidharan, B., and Kumar, P. V. (2018). Erasure coding for distributed storage: An overview. *Science China Information Sciences*, 61:1–45.

[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].

[Estrada-Galinanes et al., 2018] Estrada-Galinanes, V., Miller, E., Felber, P., and Pâris, J.-F. (2018). Alpha entanglement codes: practical erasure codes to archive data in unreliable environments. In *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 183–194. IEEE.

[Estrada-Galinanes et al., 2019] Estrada-Galinanes, V., Nygaard, R., Tron, V., Saramago, R., Jehl, L., and Meling, H. (2019). Building a disaster-resilient storage layer for next generation networks: The role of redundancy. *IEICE Technical Report; IEICE Tech. Rep.*, 119(221):53–58.

[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.

[Ferrante, 2017] Ferrante, M. D. (2017). Ethereum payment channel in 50 lines of code. Technical report, Medium blog post.

[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.

[Li and Li, 2013] Li, J. and Li, B. (2013). Erasure coding for cloud storage systems: A survey. *Tsinghua Science and Technology*, 18(3):259–272.

[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.

[McDonald, 2017] McDonald, J. (2017). Building ethereum payment channels. Technical report, Medium blog post.

[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.

[Poon and Dryja, 2015] Poon, J. and Dryja, T. (2015). The bitcoin lightning network: Scalable off-chain instant payments. Technical report, https://lightning.network.

[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.

[Schneier, 2019] Schneier, B. (2019). Data Is a Toxic Asset - Schneier on Security. [Online; accessed 6. Aug. 2020].

[Tremback and Hess, 2015] Tremback, J. and Hess, Z. (2015). Universal payment channels. Technical report, ?

[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.

[Weatherspoon and Kubiatowicz, 2002] Weatherspoon, H. and Kubiatowicz, J. D. (2002). Erasure coding vs. replication: A quantitative comparison. In *Peer-to-Peer Systems: First InternationalWorkshop, IPTPS 2002 Cambridge, MA, USA, March 7–8, 2002 Revised Papers 1*, pages 328–337. Springer.

[ZeroNet community, 2019] ZeroNet community (2019). Zeronet documentation.

Part III

INDEXES

GLOSSARY

access control The selective restriction of access to read a document or collection in Swarm.

access control trie A tree-like data structure containing access keys and other access information.

access key A symmetric key used for encryption of reference to encrypted data.

access key decryption key The key granted by the publisher to a party in a multi-party selective access scenario, used to decrypt the global access key.

accessible chunk A chunk that is accessible by routing a message between the requester and the node closest to the chunk.

addressed envelope A construct where the address of the single owner chunk is created before the chunk content is associated with it.

aligned incentives A mechanism of rewarding or penalising actors in a way that encourages the desired behavior.

anonymous retrieval The act of retrieving a chunk without disclosing the identity of the requestor node.

anonymous uploads Uploading data while keeping the uploader's identity hidden, leveraging the forwarding Kademlia routing.

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 bins Equivalence classes of chunks from the point of view of batch expiry, with the same proximity order and same batch.

batch depth The postage batch size specified as a power of 2.

batch size The amount of chunks that can be stamped with a postage batch. See also issuance volume.

bin ID A sequential counter per PO bin acting as an index of locally stored chunks on a node.

binary Merkle tree A binary tree in which each leaf node is labelled with the cryptographic hash of a data block, and each non-leaf node is labelled with a 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 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 subsequent block contains a cryptographic hash of the preceding block.

bzz account An account in Swarm, also referred to as Swarm base account.

bzz network ID The unique identifier assigned to the Swarm network.

cheque An off-chain payment method where the issuer signs a cheque specifying a beneficiary, a date, and an amount, which is given 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 fixed-sized data blob, the basic unit of storage in Swarm's DISC keyed by its address. Chunks can either be content addressed or single owner.

chunk span The length of data subsumed under an intermediate chunk.

chunk synchronisation The process in which a peer locally stores chunks received from an upstream peer.

chunk value The value assigned to a chunk based on the price of the postage batch it is stamped with. It determines the order of chunks when a node prioritises for garbage collection.

claim phase A phase within the redistribution round of the Schelling game.

collective information Data generated through collective effort, such as public forum discussions, reviews, votes, polls, and wikis.

collision slot The collection of maximum length prefixes that any two chunks stamped with a postage batch are allowed to share. Each stamped chunk occupies a collision slot.

commit phase A phase in the redistribution round of the Schelling game.

committed stake The amount of stake the stakers commit to in the staking contract.

content addressed chunk A chunk is content addressed when its address is determined by the chunk content itself. The address usually represents a fingerprint or digest of the data using some hash function. In Swarm, the default content addressed chunk uses the Binary Merkle Tree hash algorithm with Keccak256 base hash to determine its address.

data silo An isolated collection of information within an organization that is not accessible by other parts of the organization. In more general terms, it refers to the large datasets that organisations often keep exclusively for their own use.

data slavery Refers to a situation where individuals lack control over their personal data and do not receive sufficient remuneration for its commercial use by companies.

decentralised network A network architecture designed without any central nodes that 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 forming the Ethereum peer-to-peer network. Implemented as a set of programming libraries with the same name.

direct delivery Chunk delivery occurring in a single step via a lower-level network protocol.

direct notification from publisher The process where a recipient is directly notified of a feed update by the publisher or other parties known to have it.

disconnect threshold The debt threshold between peers that determines when a peer in debt will be disconnected.

dispersed replica A construct to create and retrieve replicas of a chunk dispersed across the address space under diverse addresses.

distributed hash table A distributed system that provides an efficient lookup service, enabling any participating node to 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 maintain a list of file locations, instead it actually stores pieces of the file directly on the node.

distributed storage A network of storage where information is stored on multiple nodes, possibly in replicated fashion.

distributed web application A client side web application that leverages 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 resilience to message loss.

duplicate chunk We define a chunk as a duplicate (or seen) if and only if it is already found in the local store.

effective demand The total number of chunks that have been successfully uploaded.

elliptic curve Diffie-Hellman A key agreement protocol that allows two parties, each possessing 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 used to describe an Ethereum node.

entanglement code An error correction code optimized for bandwidth of repair.

epoch A specific time period with a defined length, starting from a particular point in time.

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 of the same length.

epoch reference A combination of an epoch base time and level used to identify a specific epoch.

epoch-based feeds Special feeds that allow feeds with sporadic updates to be searchable.

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 A system analogous to the DNS of the old web, translating human-readable names into system-specific identifiers, i.e. references in the case of Swarm.

Ethereum Virtual Machine (EVM) A Turing-complete byte code interpreter responsible for calculating state changes by executing the instructions of smart contracts.

eventual consistency The guarantee that all chunks are redundantly retrievable once the neighbourhood peers have synchronised their content.

extended triple Diffie–Hellmann key exchange The standard method used to establish the initial parameters of a double ratchet key-chain.

FAANG Facebook, Apple, Amazon, Netflix, and Google.

fair data economy An economy of processing data characterised by fair compensation of all parties involved in its creation or enrichment.

feed aggregation The process of combining multiple sporadic feeds into a single periodic one.

feed index A component of the identifier for the feed chunk, used for identification and retrieval purposes.

feed topic A component of the identifier for the feed chunk, representing the topic or subject of the feed.

feeds Data structures 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. Feeds offer a persisted pull messaging system.

  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 that publish updates at regularly recurring intervals.

real-time feeds Feeds where the update frequencies may vary 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 social media status updates, a person's blog posts, or blocks of a blockchain.

sporadic feeds Feeds with irregular asynchronicities, i.e. updates can occur with unpredictable gaps.

forwarding Kademlia A recursive flavour of Kademlia routing that involves message relay.

forwarding lag The time it takes for healthy nodes to 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 if one or more session keys are obtained by an attacker.

garbage collection The selective purging process of removing unnecessary chunks from a node's local storage.

garbage collection strategy The process that determines which chunks are selected for removal during garbage collection.

global balance The amount of funds deposited in the chequebook to serve as collateral for the cheques.

granted access A type of selective access to encrypted content that requires 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 will result in direct error responses.

hive protocol The protocol used by nodes joining the network to discover their peers.

honest peers The set of applicants who agree with the selected truth in the round of the Schelling game.

immutable chunk store A storage system where no replace or update operation is available on chunks.

incentive strategy A strategy that utilises rewards and penalties to encourage desired behaviour.

inclusion proofs A proof that a string is a substring of another string, for instance verifying 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 network 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.

issuance volume The amount of chunks that can be stamped with a postage batch. See also batch size.

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 bin $0 <\leq i < d$, 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 the proximity order of their addresses relative to the 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 guarantees a path between any two nodes in $O(log(n))$ hops.

key derivation function A function that deterministically produces keys from an initial seed. It is often used concurrently by parties separately to generate secure messaging key schemes.

liars The set of applicants who disagree with the selected truth in the round of the Schelling game.

libp2p A framework and suite of protocols for building peer-to-peer network applications.

light node The concept of 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. Light nodes do not accept incoming connections.

load balancing The process of distributing a set of tasks over a set of nodes to make the process more efficient.

lookup key One of the keys involved in the process of allowing selective access to content for 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 the file.

maximum syncing latency The agreed maximum duration of latency for live syncing after a peer connection starts.

mining chunks An example of chunk mining is generating an encrypted variant of chunk content so that the resulting chunk address satisfies certain constraints, e.g. being closer to or farther away from a particular address.

missing chunk notification protocol A protocol used when a downloader cannot find a chunk, allowing it to initiate a recovery process and request the missing 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, it refers to peers residing within the neighbourhood depth of each other.

neighbourhood An area of a certain distance around an address.

- **neighbourhood depth** The distance from the node within which its peers are considered nearest neighbours. Also the highest PO d such that the address range designated by the d-bit-long prefix of the node's overlay contains at least 3 other peers.

neighbourhood notification Notification of a feed update which works without the issuer of the notification needing to know the identity of prospective posters.

neighbourhood selection anchor A randomly selected value that determines which neighbourhood can participate in the redistribution round.

neighbourhood size The number of nearest neighbours of a node.

net provider A node that contributes more resources to the Swarm network than it consumes.

net user A node that consumes more resources of the Swarm network than it contributes.

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** A node that is stably online.
- **storer node** A node that stores the requested chunk.

on-chain payment A payment made through a blockchain network.

opportunistic caching When a forwarding node receives a chunk, then the chunk is saved in case it may 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 used to identify each node running in the Swarm network. It is the basis for communication in the sense that it remains stable across sessions even if the underlay address changes.

overlay address space The address space of the overlay Swarm network consisting of 256-bit integers.

overlay network The connectivity pattern of the secondary conceptual network in Swarm, 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 architecture where tasks or workloads are partitioned between equally privileged participants known as peers.

pinner A node keeping a persistent copy of a chunk.

pinning The mechanism that makes content sticky and prevents it from being removed by garbage collection.

plausible deniability The ability to deny knowledge of any damnable actions committed by others.

postage batch An ID associated with a verifiable payment on the chain which can be attached to one or more chunks as a postage stamp.

pre-key bundle Contains all the necessary information that an initiator needs to know about the responder to initiate a cryptographic handshake.

prompt recovery of data The protocol used for missing chunk notification and recovery.

proof of density A construct that allows the winners of the Schelling game round to demonstrate that the chunks within their storage depth fill their reserve.

proof of entitlement Evidence provided by storer nodes in a neighbourhood to the blockchain, demonstrating that they have the required reserve.

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 A network protocol responsible for eventual consistency and maximum resource utilisation by pulling chunks by a certain node.

push syncing A network protocol responsible for delivering a chunk to its proper storer after it has been uploaded to an arbitrary node.

radius of responsibility The proximity order designating the area of responsibility.

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, 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 its consumers.

recovery host Pinning nodes that are willing to provide their pinned chunks in the context of recovery.

recovery request A request made to a recovery host to initiate the reupload of a missing chunk known to be pinned in its local store.

recovery response envelope An addressed envelope which provides a way for recovery hosts to directly and efficiently respond to the originator of the recovery request without incurring additional costs or computational burden.

recovery targets Volunteering nodes that are advertised by the publisher as keeping its globally pinned publication pinned.

redundancy In the context of the distributed chunk store, redundancy is achieved through surplus replicas or so-called parities that contribute to the resilience of chunk storage in the face of churn and garbage collection.

redundant Kademlia connectivity A Kademlia connectivity that remains intact even if some peers churn.

redundant retrievability A chunk is said to be redundantly retrievable with degree r if it is retrievable and would remain so even after any r nodes responsible for it leave the network.

Reed-Solomon coding A systemic erasure code that generates k extra 'parity' chunks when applied to data consisting of n chunks. These chunks allow for the reconstruction of the original blob as long as any n out of the total $n + k$ chunks are available.

reference count A property of a chunk used to prevent it from being garbage collected. It is increased when the chunk is pinned and decreased when it is unpinned.

relaying node A node relaying messages in the context of forwarding Kademlia.

requestor node A node that requests information from the network.

reserve A fixed size of storage space on a node dedicated to storing chunks from its area of responsibility.

reserve commitment The data provided by a node in the round of the Schelling game containing information about the reserve it is holding.

reserve depth The base 2 logarithm of the DISC reserve size, rounded up to the nearest integer.

reserve sample A part of the reserve used to test the consensus regarding the reserve content of a neighbourhood.

retrieve request A peer-to-peer protocol message that asks for the delivery of a chunk based on its address.

reveal phase A phase in the redistribution round of the Schelling game.

reward pot The total accumulated storage rent from all postage batches for a specific period.

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 A hypothetical method of implementing chunk delivery using Kademlia routing independently of the initial request.

routing The process of relaying messages via a chain of peers ever closer to the destination.

rules of the reserve Rules that define the content of the reserve on a node.

saboteurs The set of committers who either did not reveal data or revealed invalid data in the round of the Schelling game.

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.

Schelling game A mechanism that facilitates peer cooperation in redundantly storing data for the network's benefit, structured as a sequence of redistribution rounds.

second-layer payment Payments processed by an additional system superimposed on a blockchain network.

seeder A user who hosts 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 whose integrity is ensured by the association of its payload to an identifier attested by the signature of its owner. The identifier and the owner's account determine 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** The account of the owner of single owner chunk.
- **payload** Part of a 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.

sister nodes The nodes in the other half of the old neighbourhood after a neighbourhood split.

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.

stake balance The amount of stake serving as collateral by the staker.

stake density The amount of stake per neighbourhood size.

staking Mechanism used to steer nodes to provide reliable service in their neighbourhoods where they stake some amount that represents their commitment.

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.

storage depth The lowest proximity order at which a compliant reserve stores all batch bins.

storage rent The amount paid for storage by purchasing postage batches.

storage slot A designated space in a postage batch where a specific chunk is assigned.

stream provider Provides of a stream of chunks to another node upon request.

swap A Swarm accounting protocol with a tit-for-tat accounting scheme, enabling scalable microtransactions. It also includes a network protocol referred to as Swap.

Swarm base account The Ethereum account associated with a Swarm node. See also bzz account.

Swarm manifest A structure that defines a mapping between arbitrary paths and files to handle collections.

tar stream In computing, tar is a computer software utility for combining multiple files into a single archive file, often referred to as a tarball.

targeted chunk delivery A mechanism for requesting a chunk from an arbitrary neighbourhood where it is known to be stored and delivering it 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 in a shared-resource system where individual users prioritise their own interests contrary to common good, leading to the depletion or degradation of the shared resource through collective action.

Trojan chunk A chunk containing a disguised message while appearing indistinguishable from other chunks.

trustless A property of an economic interaction system where service provision is either realtime verifiable and/or providers are accountable, rewards 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, which might not remain stable between sessions.

underlay network The lowest level base network through which nodes connect using a peer-to-peer network protocol as their transport layer.

uniformity depth The number of storage slots or buckets within a postage batch specified in powers of 2.

upload and disappear A method of deploying interactive dynamic content to be stored in the cloud so it may be retrieved even if the uploader goes offline.

upload tag An object that represents an upload and tracks the progress by counting the number of chunks that have reached a specific state.

uploader An entity uploading content to the Swarm network.

upstream peer The peer that precedes some other peer in the chain of forwarding.

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 collaboratively creating interactive content, while retaining full control over it.

ZeroNet A decentralised web platform using Bitcoin cryptography and the BitTorrent network.

INDEX

O

P

R

S

T

U

W

Z

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 binary Merkle tree.

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.

RS Reed-Solomon coding.

SWAP service wanted and provided ALSO settle with automated payments ALSO send waiver as payment ALSO start without a penny, see swap.

TLD top level domain.
TTL time to live.

WWW World Wide Web.

X3DH extended triple Diffie–Hellmann key exchange.