swarm

swap, swear and swindle
incentive system for swarm

viktor trón, aron fischer, dániel a. nagy, zsolt felföldi, nick johnson

draft version May 2016
ETHERSPHERE ORANGE PAPERS 1

Contents

1 Web Hosting and Incentivization **3**
- 1.1 Web 1.0 . . . 3
- 1.2 Web 2.0 . . . 4
- 1.3 Peer-to-peer networks . . . 4
- 1.4 The economics of bittorrent and its limits . . . 4
- 1.5 Towards Web 3.0 . . . 5

2 Bandwidth Incentives **6**
- 2.1 Accounting . . . 7
  - 2.1.1 Triggers for payment and disconnect . . . 7
  - 2.1.2 Negotiating chunk price . . . 8
  - 2.1.3 Modes of payment . . . 8
- 2.2 Charging for Retrieval . . . 9

3 Storage Incentives **10**
- 3.1 Compensation for storage and guarantees for long-term data preservation . . . 10
- 3.2 Owner-side handling of storage redundancy . . . 13
  - 3.2.1 Loss-tolerant Merkle trees and erasure codes . . . 13
  - 3.2.2 The special case of the last chunks in each row . . . 15
  - 3.2.3 Benefits of CRS merkle tree . . . 16
- 3.3 Registered nodes and Ensured ARchival (SWEAR) . . . 17
- 3.4 Litigation on loss of content (SWINDLE) . . . 18
  - 3.4.1 Submitting a challenge . . . 18
  - 3.4.2 The outcome of a challenge . . . 19
  - 3.4.3 Publicly accessible receipts and consumer driven litigation . . . 20
- 3.5 Receipt forwarding or chained challenges . . . 21
  - 3.5.1 Forwarding chunks . . . 21
  - 3.5.2 Collecting storer receipts and direct contracts . . . 22
  - 3.5.3 Chaining challenges . . . 22
  - 3.5.4 Multiple receipts and local replication . . . 25
- 3.6 Pricing, deposit, accounting . . . 25
  - 3.6.1 Pricing . . . 25
  - 3.6.2 Deposit . . . 26
  - 3.6.3 Accounting and settlement . . . 26

4 Conclusion **28**

5 Glossary **28**

Bibliography **30**

Abstract

This paper [a] is the result of our research into incentives for decentralised storage and distribution in the context of implementing an incentive layer for swarm.
Swarm is a distributed storage platform and content distribution service, a native base layer service of the Ethereum web 3 stack. The primary objective of Swarm is to provide a decentralized and redundant store of Ethereum's public record, in particular to store and distribute dapp code and data as well as block chain data. From an economic point of view, it allows participants to efficiently pool their storage and bandwidth resources in order to provide these services to all participants.
The goal is a peer-to-peer serverless hosting, storage and serving solution that is DDOS-resistant, has zero-downtime, is fault-tolerant and censorship-resistant as well as self-sustaining due to a built-in incentive system.
As the storage layer for interactive web 3 applications and blockchains, the swarm must satisfy some key requirements. The swarm has to show dynamic scalability, i.e. it must adapt to sudden surges in popular demand, and at the same time the swarm must also preserve niche content and ensure its availablility. In this document we aim to present an incentive structure for nodes in the swarm, that is in alignment with and conducive to these requirements. Putting aside the – otherwise very important – question of search and content promotion, this paper focuses on the issues of bandwidth and storage.

[a] The authors are indebted to Gavin Wood, Vitalik Buterin and Jeffrey Wilcke for their daring vision of the next generation internet and for their original concepts of web 3 and swarm. We thank Vitalik Buterin, Piper Merriam, Taylor Gerring, Anish Mohammed, Martin Becze, Christian Reitwiessner for their comments and suggestions. Their names here by no means imply endorsement. All errors are ours.

1 Web Hosting and Incentivization

In order to both appreciate the problems we are trying to solve and understand the demands we make on the swarm, a little history is useful.

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

1.1 Web 1.0

In the times of Web 1.0, in order to have your content accessible by the whole world, you would typically fire up a web server or use some web hosting (free or cheap) and upload your content that could be navigated through a set of html pages. If your content was unpopular, you'd 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 HA clusters and fat pipes and with the growth of your popularity, your costs grew, without any obvious way to cover them. There were very few practical ways to let (let alone *make*) your audience share the burden of information dissemination directly.

The common wisdom at the time was that it would be ISP's that would come to the rescue, since in the early days of the Web revolution, bargaining about peering arrangements between ISP's involved arguments about where providers and where consumers are and which ISP is making money from the other's network. Indeed, when there was a sharp disbalance between originators of TCP connection requests (i.e. SYN packets), it was customary for the originator ISP to pay the recipient ISP, which made the latter somewhat incentivized to help hosters of popular content. In practice, however, this incentive structure usually resulted in putting a free *pr0n* or *warez* server in the server room to tilt 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 typically owned their content.

1.2 Web 2.0

The transition to Web 2.0 changed much of that. Context-sensitive targeted advertizing offered a Faustian bargain to content producers. As in "We give you scalable hosting that would cope with any traffic your audience throws at it, but you give us substantial control over your content; we are going to track each member of your audience and learn – and own – as much of their personal data as we can, we are going to pick who can and who cannot see it, we are going to proactively censor it and we may even report on you, for the same reason. Thus, millions of small content producers created immense value for very few corporations, getting only peanuts (typically, free hosting) in exchange.

1.3 Peer-to-peer networks

At the same time, however, the P2P revolution was gathering pace. Actually, P2P traffic very soon took over the majority of packets flowing through the pipes, quickly overtaking the above mentioned SYN-bait servers. If anything, it proved beyond doubt that using the hitherto massively underutilized upstream bandwidth of regular end-users, they could get the same kind of availability and bandwidth for their content as that provided by big corporations with data centers attached to the fattest pipes of the internet's backbone. What's more, this could be achieved at a fraction of the cost. In particular, users retained a lot more control and freedom over their data. Finally, this mode of data distribution proved to be remarkably resilient even in the face of powerful and well-funded entities extending great efforts to shut it down.

On the other hand, even the most evolved mode of P2P file sharing, which is trackerless Bittorrent, is just that: file sharing. It is not suitable for providing the kind of interactive, responsive experience that people came to expect from web applications on Web 2.0. Simply sharing upstream bandwidth and hard-drive space and a tiny amount of computing power without proper accounting and indexing only gets you so far. However, if you add to the mix a few more emergent technologies – most importantly the blockchain – you get what we believe to deserve the Web 3.0 moniker: a decentralized, censorship-resistant way of sharing and even collectively creating *interactive* content, while retaining full control over it. The price is surpisingly low and mostly consists of the resourses supplied by the super-computer (by yesteryear's standards) that you already own or can rent for peanuts.

1.4 The economics of bittorrent and its limits

The genious of Bittorrent lies in its clever resource optimisation: If many clients want to download the same content from you, give them different parts of it and let them swap the missing parts between one another in a tit-for-tat fashion. This way, the upstream bandwidth use of a

content hoster (*seeder* in Bittorrent parlance) is roughly the same, no matter how many clients want to download it simultaneously. This solves the most painful issue of the ancient HTTP underpinning the World Wide Web.

Cheating (i.e. feeding your peers with garbage) is discouraged by the use of *hierarchical piecewise hashing*, whereby a package offered for download is identified by a single short hash, and any part can be cryptographically proven to be a specific part of the package without all the other parts, with a very small overhead.

This beautifully simple approach has three main shortcomings, somewhat related:

- There are no built-in incentives to seed downloaded content. In particular, one cannot exchange the upstream bandwidth provided by seeding one content for downstream bandwidth required for downloading some other content. Effectively, upstream bandwidth provided by seeding somebody else's content is not directly rewarded in any way.
- Typically, downloads start slowly and with delay. Clients that are further ahead in downloading have much more to offer to and much less to demand from newcomers. This results in bittorrent downloads starting as a trickle before turning into a full-blown torrent of bits. This severely limits the use of bittorrent in responsive interactive applications.
- Small chunks of data can only be shared in the context of the larger file that they are part of. We find peers sharing the content we seek by querying the Distributed Hash Table (DHT) for said file. Thus a peer sharing only part of a file needs to know what that file is in order to be found in the DHT, and conversely, if the peer doesn't know that the data chunks belong to some file the peer will not be found by users seeking that file. This commonly happens for example when the same chunks of data appear verbatim in multiple files. Also, unless their objective is simply to get the missing parts of a file from their peers, nodes are not rewarded for their sharing efforts (storage and bandwidth), just like seeders.

1.5 Towards Web 3.0

The straightforward approach of using bittorrent as a distribution mechanism for web content has been successfully implemented by *Zeronet*. However, because of the aforementioned issues with bittorrent, Zeronet fails to provide the same kind of responsive experience that users of web services came to expect.

In order to enable responsive distributed web applications (called dapps in Web 3.0 communities), *IPFS* (*[5]*) had to introduce a few major improvements over Bittorrent. The most immediately apparent novelty is the highly web-compatible URL-based retrieval. In addition, the directory (also organized as a DHT) has been vastly improved, making it possible to search for any part of any file (called *chunk*). It has also been made very flexible and pluggable in order to work with any kind of storage backend, be it a laptop with intermittent wifi, or a sophisticated HA cluster in a fiber-optic connected datacenter.

A further important innovation is that IPFS has incentivisation factored out into pluggable modules. Modules such as bitswap for example establish that it is in the interest of greedy downloaders to balance the load they impose on other nodes, and also that it is in every node's interest to host popular content. Bitswap or no bitswap, IPFS largely solves the problem of content consumers helping shouldering the costs of information dissemination.

The same problem with lack of incentives is apparent in various other projects such as zeronet or MAIDSAFE. Incentivization for distributed document storage is still a relatively new research field, especially in the context of the brand new *blockchain technology*. The Tor network has

seen suggestions ([6], [4]) but these schemes are largely academic, they are not built in at the heart of the underlying system. Bitcoin has also been repurposed to drive other systems like Permacoin ([8]) or Sia ([13]), some use their own blockchain, altcoin such as Metadisk ([15]) for Storj ([14]) or Filecoin ([3]) for IPFS [5].

What is still missing from the above incentive system, is the possibility to rent out large amounts of disk space to those willing to pay for it, irrespective of the popularity of their content; and conversely there is also a way to deploy your interactive dynamic content to be stored in the cloud - "*upload and disappear*".

The objective of any incentive system for p2p content distribution is to encourage cooperative behavior and discourage freeriding: the uncompensated depletion of limited resources. In what follows we present our current thinking for a comprehensive incentive system for swarm implemented through a suite of smart contracts. The incentive system leverages the ethereum infrastructure and the underlying value asset, Ether.

The incentive strategy outlined here aspires to satisfy the following constraints:

- It is in the node's interest irrespective of whether other nodes follow it or not.
- It makes it expensive to hog other nodes' resources.
- 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 incentive scheme. The incentives for bandwith use are designed to achieve speedy and reliable data provision while the storage incentives are designed to ensure long term data preservation, ideally solving the "upload and disappear" problem. In *Section 3* we introduce the basic functioning of the swarm incentive system and describe the Swarm Accounting Protocol (SWAP) which handles compensation for bandwidth use in realtime. In *Section 3* we turn to the problem of data preservation and offer a solution.

2 Bandwidth Incentives

The ultimate goal of swarm is that end users are served content in a safe and speedy fashion. The underlying unit of accounting must be a uniformly sized chunk of data (henceforth, simply *chunk*) since this is the delivery unit that is sourced from a single independent entity. We start from the simplest assumption that delivery of a chunk is a valuable service which is directly chargeable when a chunk is delivered to a node that sent a retrieve request.

Swarm is organized as a *content-addressed chunkstore*, whereby the addresses of chunks are derived from their hash value and come from the same address space as those of participating nodes. Thus, the same distance metric can be applied between nodes, between chunks and between a node and a chunk.

From the perspective of any individual node, the probability of a given chunk being ever requested is proportional to the inverse of its distance from it (the distance, in turn, can be interpreted as the risk of it not being requested). In other words, following the underlying routing protocol by itself incentivises nodes to prefer chunks that are closer to their own address.

In the first iteration, we further assume that nodes have no preference as to which chunks to store other than their access count which is a reasonable predictor of their profitability. As a

corollary, this entails that store requests are accepted by nodes irrespective of the chunk they try to store.

2.1 Accounting

The idea is that nodes can trade services for services or services for tokens in a flexible way so that in normal operation a zero balance is maintained between any pair of nodes in the swarm. This is done with the *Swarm Accounting Protocol* (SWAP (Swarm Accounting Protocol)), a scheme of *pairwise accounting* with negotiable prices.

2.1.1 Triggers for payment and disconnect

Each swarm node keeps a tally of offered and received services with each peer. In the simplest form, the service is the delivery of a chunk or more generally an attempt to serve a retrieve request, see later. We use the number of chunks requested and retrieved as a discrete integer unit of accounting. The tally is independently maintained on both ends of each direct connection in the peer-to-peer network for both self and the remote peer. Since disconnects can be arbitrary, it is not necessary to communicate and consent on the exact pairwise balances.

Each chunk delivery on the peer connection is accounted and exhcanged at a rate of one to one. On top of this, there is a possibility to compensate for services with ether (or other blockchain token) at a price agreed on in advance. Receiving payment should be accounted for equivalent service rendered, using the price offered.

In the ideal scenario of compliant use, the balance is kept around zero. When the mutual balance on a given connection is tilted in favour of one peer, that peer should be compensated in order to bring the balance back to zero. If the balance tilts heavily in the other direction, the peer should be throttled and eventually choked and disconnected. In practice, it is sufficient to implement disconnects of heavily indebted nodes.

In stage one, therefore, we introduce two parameters that represent thresholds that trigger actions when the tally reached them.

`Payment threshold`
: is the limit on self balance which when reached trigger a transfer of some funds to the remote peer's address in the amount of balance unit times unit price offered.

`Disconnect threshold`
: is the limit which when reached triggers disconnect from the peer.

When node A connects with peer B the very first time during one session, the balance will be zero. Since payment is only watched (and safe) if connection is on, B needs to either (i) wait till A's balance reaches a positive target credit level or (ii) allow A to incur debt. Since putting one node in positive credit is equivalent to the other incurring debt, we simply aim for (ii). In other words, upon connection we let peers get service right away and after the payment threshold is reached, we initiate compensation that brings balance up to zero.

In its simplest form, balances are not persisted between sessions (of the swarm node), but are preserved between subsequent connections to the same remote peer. Therefore balances can be stored in memory only. Freeriding is already very difficult with this scheme since each peer that a malicious node is exploiting, will provide free service only up to the value of *disconnect threshold* times unit price. While the node is running no reconnect is allowed unless compensation is paid to bring a balance above disconnect threshold.

2.1.2 Negotiating chunk price

Prices are communicated in the protocol handshake as *highest accepted chunk price* and *offered chunk price*. The handshake involves checking if the highest accepted chunkprice of one peer is less than the offered chunkprice of the other. If this is the case no business is possible and the other peer can only be compensated on a service for service basis. If payment is not possible either way, the peers will try to keep a balance until one peer's disconnect limit is reached. There is also the possibility that when A and B connect, payment is only possible in one direction, from B to A, but A cannot pay B for services. In this case if A reaches past the payment limit, it does nothing. Since this is clearly a risk for B, it may make sense to keep the connection only if B stays predominantly in red (i.e., continually downloads more), otherwise disconnect.

All in all, it is not necessary for both ends to agree on the same price (or even agree on any price) in order to successfully cooperate. Potentially different pricing of retrievals is meant to reflect varying bandwidth costs. Setting highest accepted chunk price as 0 can also be used to communicate that one is unable or unwilling to pay with tokens.

2.1.3 Modes of payment

Since transfer of ether is constrained by blocktime, actual transactions sent via the blockchain can effectively rate-limit a peer, moreover various delays in transaction inclusion might interfere with the timing requirements of accounting compensation.

Things can be improved if peers send some provable commitment to compensation directly in the *bzz protocol*. On the one hand, as long as these commitments need blockchain transactions to verify, the risk for receiver is similar: by the time failing transactions are recognised by the creditor node, the indebted node is already more in debt. Whether the balance is restored after this can only be verified by checking the canonical chain after sending the transactions. On the other hand, provable commitments have two advantages: (i) they keep the accounting real time and (ii) allow for a differential treatment of inadvertant non-payment versus deliberate cheating.

One particular implementation could use ethereum transactions directly within the bzz protocol. Unfortunately, sending them to the network is not a viable way to cash the payment they represent: If the same address is used to send transactions to multiple peers that act independently, there is no guarantee that the transactions end up in the same block or follow the order of their nonces. Therefore, while they provide basic authentication, they can fail due to insufficient balance or incorrect nonce.

Smart contracts, however, make it easy to implement a more secure payment process. Instead of a simple account, the sender address holds a *chequebook contract*. This chequebook contract is similar to a wallet holding an ether balance for the owner and allows signed cheques to be cashed by the recipient (or anyone), who simply send a transaction with the cheque as data to the contract's *cash* method.

- the contract keeps track of the cumulative total amount sent during the time of the connection.
- sender makes sure each new cheque sent increments the cumultive total amount sent.
- after connection is established, the cumulative amount for a remote peer is set based on the tally on the blockchain
- the cumulative amount for self (local peer) must be persisted since valid transactions may be in transit

the cheque is valid if:

- the contract address matches the address on the cheque,
- the cheque is signed by the payer (NodeId = public key sent in handshake)
- the signed data is a valid encoding of <contract address,beneficiary,amount>
- the cumulative total amount is greater than in the previous cheque sent.

Receiver may keep only the last cheque received from each peer and periodically cash it by sending it to the chequebook contract: a scheme that allows trusted peers to save on transaction costs.

Peers watch their receiving address and account all payments from the peer's chequebook and when they are considered confirmed, the tally is adjusted. The long term use of a chequebook provides a credit history, use without failure (bounced cheques) constitues proof of compliance. Using the cumulative volume on the chequebook to quantify reliability renders chequebooks a proper *reputation system*.

SWAP can also use a fully featured *payment channel* as mode of payment. A SWAP payment channel is an agreement between two peers to maintain an ether balance for pairwise accounting. This allows for secure offchain transactions and delayed updates where the release of locked funds is potentially constrained by escrow conditions. The channel contract can be extended to accept cheques. Both the chequebook and channel contracts have withdrawal rules where the release of funds is authorized only after a successful freeze period during which the counterparty can update the state on the blockchain with the last consensus. The details of SWAP and the channel contract will be published in a separate paper (*[12]*).

2.2 Charging for Retrieval

When a retrieve request is received the peer responds with delivery if the preimage chunk is found. As a simplification, we assume that requesters credit their peers only upon first successful delivery, while nodes receiving the request charge for their forwarding effort right away. This keeps a perfect balance if each retrieve request results in successful retrieval or the ratio of failed requests is similar for the two peers (and have small variance accomodated by the disconnect threshold). In cases that this balance is genuinely skewed, one node must be requesting non-existing chunks or the other peer has inadequate connections or bandwidth resulting in its inability to deliver the requested existing chunks. Both situations warrant disconnection.

By default nodes will store all chunks forwarded as the response to a retrieve request. These lookup results are worth storing because repeated requests for the same chunk can be served from the node's local storage without the need to "purchase" the chunk again from others. This strategy implicitly takes care of auto-scaling the network. Chunks originating from retrieval traffic will fill up the local storage adjusting redundancy to use maximum dedicated disk/memory capacity of all nodes. A preference to store frequently retrieved chunks results in higher redundancy aligning with more current usage. All else being equal, the more redundant a chunk, the fewer forwarding hops are expected for their retrieval, thereby reducing expected *latency* as well as network traffic for popular content.

Whereas retrieval compensation may prove sufficient for keeping the network in a relatively healthy state in terms of latency, from a resilience point of view, more work is needed. We may need additional redundancy to be resilient against partial network outages and we need extra incentives to ensure long-term availablilty of content even when it is accessed rarely. In the following section we address these concerns.

3 Storage Incentives

Storage incentives refer to the ability of a system to encourage/enforce preservation of content, especially if the user explicitly requires that in the fashion of 'upload and disappear' discussed in the introduction.

3.1 Compensation for storage and guarantees for long-term data preservation

Filecoin (*[3]*), an incentivised p2p storage network using IPFS (*[5]*) offers an interesting solution. Nodes participating in its network also mine on the filecoin blockchain. Filecoin can be earned (mined) through replicating other people's content and spent on having one's content replicated. Filecoin's proof of work is defined to include proof that the miner possesses a set of randomly chosen units of storage depending on the parent block. Using a strong proof of retrievability scheme, Filecoin ensures that the winning miner had relevant data. As miners compete, they will find that their chances of winning will be proportional to the percentage of the existing storage units they actually store. This is because the more missing ones need to be retrieved from other nodes, and the more their response is delayed, decreasing their chance to win the block.

With this scheme, Filecoin provides positive incentvisation on a collective level. Such a system is suspect to a 'tragedy of the commons' problem in that losing any particular data will have no negative consequence to any one storer node. This lack of individual accountability means the solution is rather limited as a security measure against lost content.

On top of this collective positive incentvisation implemented by competitive proof of retrievability mining is wasteful in terms of network traffic, computational resources as well as blockchain storage [1] and therefore unlikely to scale well. In what follows we will pursue a different approach.

While Swarm's core storage component is analogous to traditional DHTs (distributed hash tables) both in terms of network topology and routing used in retrieval, it uses the narrowest interpretation of immutable content addressed archive. Instead of just metadata about the whereabouts of the addressed content, the proximate nodes actually store the data itself. When a new chunk enters the swarm storage network, it is propagated from node to node via a process called *syncing*. The goal is for chunks to end up at nodes whose address is closest to the chunk hash. This way chunks can be located later for retrieval using kademlia key-based routing (*[7]*).

As discussed in the *section 2*, the primary incentive mechanism in swarm is compensation for retrieval where nodes are rewarded for successfully serving a chunk. This reward mechanism has the added benefit of ensuring that the popular content becomes widely distributed (by profit maximising storage nodes serving popular content they get queried for) and as a result retrieval latency is descreased.

The flipside of using only this incentive on it own is that chunks that are rarely retrieved may end up lost. If a chunk is not being accessed for a long time, then as a result of limited storage capacity it will eventually end up garbage collected to make room for new arrivals. In order

[1] If the set of chunks are not selected differently for each node, mining will resemble a DDOS on nodes that actually store the data needed for the round. Given an upper bound on nodes' storage capacity, the expected proportion of the data that needs to be retrieved increases as the network grows. This causes miners to end up competing on bandwidth. Due to competitive mining, nodes perform the same computations wasting resources. If content is known to be popular (replicated in multiple copies) or temporary (the owner can afford to lose it), checking their integrity is spurious. Without this distinction, mining is wastful even if it is non-redundant across nodes. Finally, in order to check proof of retrievability responses as part of block validation, existing data needs to be recorded on the blockchain.

for the swarm to guarantee long-term availability, the incentive system needs to make sure that additional revenue is generated for chunks that would otherwise be deleted. In other words, unpopular chunks that do not generate sufficient profit from retrievals should compensate the nodes that store them for their opportunities forgone.

A long-term storage incentivisation scheme faces unique challenges. For example, unlike in the case of bandwidth incentives where retrievals are immediately accounted and settled, long-term storage guarantees are promisory in nature and deciding if the promise was kept can only be decided at the end of its validity. Loss of reputation is not an available deterrent against foul play in these instances: since new nodes need to be allowed to provide services right away, cheaters could just resort to new identities and keep selling (empty) storage promises.

In the context of an owner paying one or more entities for long-term storage of data where losing even a small part of it renders it useless, but large enough for it not to be practical to regularly verify the availability of every bit, particual attention is required in the design of the incentive system to make the failure to store every last bit unprofitable. In the typical case, the storer receives some payment upon successfully passing a probabilistic audit. Such audits are structured in such a way that the storer always passes the audit, if she indeed has the entire data set, but there is also some non-zero probability of passing the audit if some data is lost. This probability, of course, depends on what data is lost, but in any design there is a maximum to it, typically attained by losing some very small portion of data. In case of systems that break up the data into chunks with a fixed maximum size, the probability of passing the audit with incomplete data is maximized in the case when only one chunk is lost.

Let us denote this maximum probability by p. To keep failure to store all data unprofitable, the expected payout E of subjecting oneself to an audit should be negative, if the probability of failure is greater or equal to $1 - p$. Formally: $E = pR - (1 - p)P < 0$ where R is the reward for passing the audit and P is the penalty for failing it. This yields the following important constraint on the relative amounts of reward and penalty: $P > pR/(1 - p)$. If this constraint is not met, storers doing a less than perfect – and thus unacceptable – job can still be profitable, resulting in misaligned incentives.

In the simplest case, the audit is a securely randomized spot check of having one random chunk of data in custody, with the proof of custody being the chunk itself. If the data consists of n chunks, the probability of passing the audit with one chunk missing $p = (n - 1)/n$. This yields $P > (n - 1)R$. For different audits, p is calculated differently.

Thus, in order to pay R for a successful audit, the auditor (be it the owner, a third party or even a smart contract) needs to verify whether or not the storer has comitted to losing at least P in case of failure before the audit.

Instead, we need punitive measures to ensure compliance with storage promises. These will work using a *deposit system*. Nodes wanting to sell promisory storage guarantees should have a *stake verified and locked-in* at the time of making their promise. This implies that nodes must be *registered* in advance with a contract and put up a *security deposit*.

Following *registration*, a node may sell storage promises covering the time period for which their funds are locked. While their registration is active, if they are found to have lost a chunk that was covered by their promise, they stand to loose their deposit.

In this context, *owner* refers to the originator of a chunk (the one that uploads a document to the swarm), while *storer* refers to a swarm node that actually stores the given chunk.

Let us start from some reasonable guiding principles:

- owners need to express their risk preference when submitting to storage

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

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

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

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

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

It is also worth emphasizing that the producer and the consumer of the information may not be the same entity and it is therefore important that failure to make good on the promise to deliver the stored content is penalized even when the unserved consumer was not party to the agreement to store and provide the requested content. Litigation therefore is expected to be available to third parties wishing to retrieve content.

The simplest solution to manage storage deals is using direct contracts between owner and storer. This can be implemented with storers returning *signed receipts* of chunks they accept to store and owners paying for the receipts either directly or via escrow. In the latter case, storer only gets awarded the locked funds if they provide proof that the chunk they stored is valid. Such delayed payment solutions would enable operation entirely without litigation. The receipts collected can be used to prove commitment in case of litigation. There are other more indirect variants of litigation which do not rely on owner and storer being in direct contractual agreement, which is the case if the eventual consumer is distinct from the storer and not known to them in advance.

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

swap Nodes are in semipermanent long term contact with their registered peers. Along these connections the peers are swapping various pieces of information relating to syncing, receipting, price negotiation, auditing and offchain payments.

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

swindle A scheme to pool resources to enforce adherence to the rules, by regular auditing, policing, and eventually conscientious litigation.

As we go along, these names will reveal their secondary meanings.

Security begins at home and so the first step in securing data begins with the owner; this is the topic of the following section. Then in section *Registered nodes and Ensured ARchival (SWEAR)*

we describe how the owner hands over custody of their data to registered nodes in the swarm subject to an insurance contract (SWEAR). Finally, in section *Litigation on loss of content (SWINDLE)*, we turn to how such insurance is enforced by the ethereum smart contract based litigation system (SWINDLE).

3.2 Owner-side handling of storage redundancy

First we show how to delegate setting arbitrary *levels of reliability* to the owner. The idea is that *redundancy* is encoded in the document structure before its chunks are uploaded. This is important since this entails that reliability need not be among the parameters handled by store requests, pricing or litigation.

A simplistic method of guarateeing redundancy of a file is to split the file into chunks that are one byte shorter than the normal chunksize and add a nonce byte to each chunk. This guarantees that each chunk is different and as a consequence all chunks of the modified file are different. When joining the last byte of each chunk is ignored so all variants map to the same original. This yields a potential 256 equivalent replicas of each chunk for the owner to upload (and up to 256^x different root hashes) [2] .

Luckily there are a lot more economical ways to encode data redundantly. In what follows we spell out our proposal to introduce a scheme for *loss tolerant merkle tree*.

3.2.1 Loss-tolerant Merkle trees and erasure codes

Recall that the basic data structure in swarm is a *Merkle tree*. Assuming h the size of the hash output of the hash function used in bytes, b is the branching factor. Each node represents the root hash of a subtree or, at the last level, the hash of a $b * h$ long span (one chunk) of the file. Generically we may think of each chunk as consisting of b hashes:

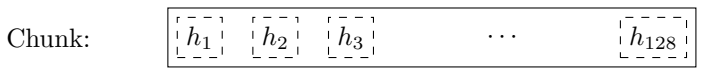
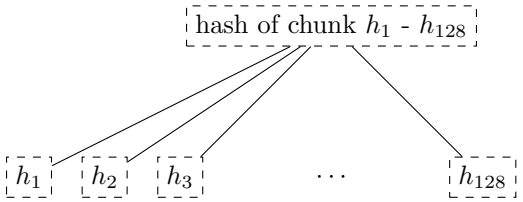


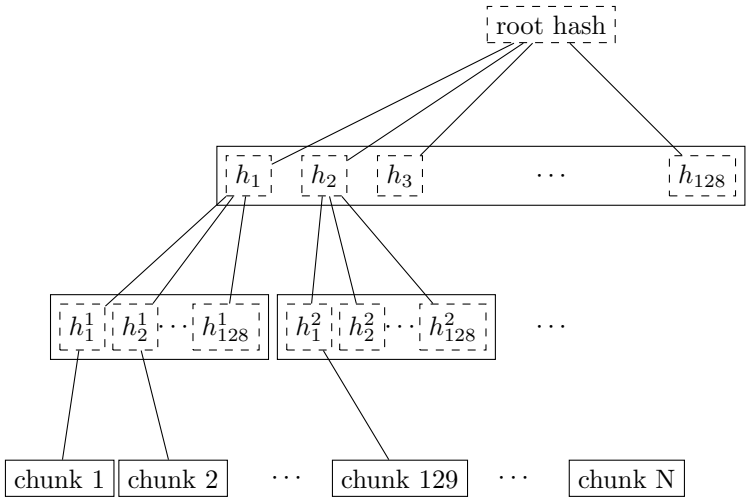
Fig. 1: A chunk consists of 4096 bytes of the file or a sequence of 128 subtree hashes.

while in the tree structure, the 32 bytes stored at the node represent the hash of the 128 children.



[2] We also explored the possibility that degree of redundancy is subsumed under local replication (section *Multiple receipts and local replication*). Local replicas are instances of a chunk stored by nodes in a close neighbourhood. If that particular chunk is crucial in the reconstruction of the content, the swarm is much more vulnerable to chunk loss or latency due to attacks. This is because if the storers of the replicas are close, infiltrating in the storers' neighbourhood can be done with as many nodes as chunk type (as opposed to as many as chunk replicas). If there is cost to sybil attacks this brings down the cost by a factor of n where n is the number of replicas. We concluded that local replication is important for resilience in case of intermittent node dropouts, however, inferior to other solutions at implementing levels of security.

Recall also that during normal swarm lookups, a swarm client performs a lookup for a hash value and receives a chunk in return. This chunk in turn constitutes another 128 hashes to be looked up in return for another 128 hashes and so on until the chunks received belong to the actual document. Here is a schematic: (figure 3.2.1):



We propose using the Cauchy-Reed-Solomon scheme (henceforth CRS (Cauchy-Reed-Solomon), see *[1]*, *[10]*) to encode redundancy directly into the swarm tree. The *CRS scheme* [3] is a *systemic erasure code* (*[2]*) which applied to a data blob of m fixed-size pieces, produces k extra pieces (so called *parity pieces*) of the same size in such a way that any m out of $n = m + k$ fix-sized pieces are to reconstruct the original blob with storage overhead of $\frac{k}{m}$.

The *chunker* algorithm using m-out-of-n CRS coding would proceed the following way when splitting the document:

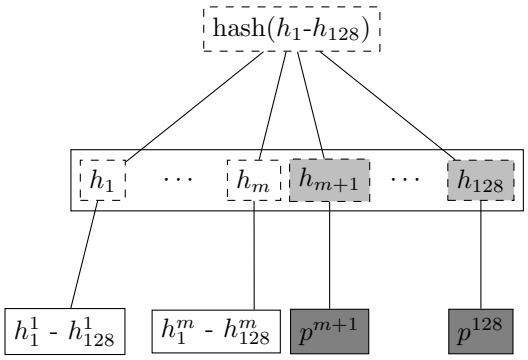
1. Set input to the data blob.
2. Read the input one chunk (say fixed 4096 bytes) at a time. Count the chunks by incrementing i. The last chunk read may be shorter.
3. Repeat 2 until there's no more data or $i \equiv 0 \mod m$
4. use the CRS scheme on the last $i \mod m$ chunks to produce k parity chunks resulting in a total of $n \leq m + k$ chunks.
5. Calculate the hashes of all the these chunks and concatenate then to result in the next chunk (of size $i \mod m$ of the next level. Record this chunk as the next
6. If there is more data repeat 2. otherwise
7. If the next level data blob has more than one chunk, set the input to this and repeat from 2.
8. Otherwise remember the blob as the root chunk.

Assuming we fix the branching factor of the swarm hash (chunker) as $n = 128$ and $h = 32$ as the size of the *SHA3 Keccak hash*. This gives us a chunk size of 4096 bytes.

Let us now suppose that we start splitting our input document data into chunks, and after each m chunks then add $k = n - m$ parity check pieces using a Reed-Solomon code so that now any m-out-of-n chunks are sufficient to reconstruct the document. On the next level up the chunks

[3] There are open source libraries that implement Reed Solomon or Cauchy-Reed-Solomon coding. See *[9]* for a thorough comparison.

are composed of the hashes of the m data chunks and the k hashes of the parity chunks. Let's take the first m of these and add an additional k parity chunks to those such that any m of the resulting n chunks are sufficient to reconstruct the origial m chunks. And so on and on every level. In terms of availability, every subtree is equally important to every other subtree at this level. The resulting data structure is not a balanced tree since on every level i the last k chunks are parity leaf chunks while the first m are branching nodes encoding a subtree of depth $i-1$ redundantly. A typical piece of our tree would look like this: (figure 3.2.1)



This pattern repeats itself all the way down the tree. Thus hashes h_{m+1}^1 through h_{128}^1 point to parity data for chunks pointed to by h_1^1 through h_m^1. Parity chunks p^i do not have children and so the tree structure does not have uniform depth.

3.2.2 The special case of the last chunks in each row

If the number of file chunks is not divisible by m, then we cannot proceed with the last batch in the same way as the others. We propose that we encode the remaining chunks with an erasure code that guarantees at least the same level of security as the others. Note that it is not as simple as choosing the same redundancy. For example a 50-of-100 encoding is much more secure against file loss than a 1-of-2 encoding even though the redundancy is 100% in both cases. Overcompensating, we could say that there should always be the same number of parity chunks even when there are fewer than m data chunks so that we always end up with m-out-of-n. We repeat this procedure in every row in the tree.

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 of course is that any number of copies of the same chunk all have the same hash and are therefore indistinguishable in the swarm. Thus when there is only a single chunk left over at some level of the tree, we'd have to apply some transformation to it to obtain a second (but different) copy before we could generate more parity chunks.

In particular this is always the case for the root chunk. To illustrate why this is critically important, consider the following. The root hash points to the root chunk. If this root chunk is lost, then the file is not retrievable from the swarm even if all other data is present. Thus we must find an additional method of securing and accessing the information stored in the root chunk.

Whenever a single chunk is left over ($m = 1$) we propose to append an extra padding byte to the chunk not counting towards its size. In swarm, each 4096 byte chunk is actually stored together with 8 bytes of meta information - currently only storing the size of the subtree encoded by the chunk. Since the subtree size determines exactly what span of the chunk is substantive data,

the padding differential byte is easily ignored when the document is assembled [4] .

3.2.3 Benefits of CRS merkle tree

Assuming p is the probability of losing one piece, if all n pieces are independently stored, the probability of loosing the original content is p^{n-m+1} which is exponential while extra storage is linear. These properties are preserved if we apply the coding to every level of a swarm chunk tree. Fixing both the branching factor b (essentially chunk size) as well as redundancy m, we can keep the decoding overhead (quadratic in file size) a constant, which means processing can scale (being linear in the number of nodes, i.e., file size (logarithmic with parallelisation). Very importantly, however, this causes the reliability to exponentially converge to zero, defeating the purpose of using erasure codes. This loss of reliability, however, can be overcome by conducting regular *auditandrepair* checks (*[11]*) which provide exponential increase in reliability using loglinear resources. We are currently looking at this *divide and conquer* approach to the scaling of erasure codes, the ref chunk tree does not only ensure file availability, but also offers further benefits of increased resilience and ways to speed up retrieval.

All chunks are created equal

A tree encoded as suggested above has the same redundancy at every node [5]. This means that chunks nearer to the root are no longer more important than chunks closer to the leaf nodes. Every node has an m-of-128 redundancy level and no chunk after the root chunk is more important than any other chunk [6] .

Self healing

Any client downloading a file from the swarm can detect if a chunk has been lost. The client can reconstruct the file from the parity data (or reconstruct the parity data from the file) and resync this data into the swarm. That way, even if a large fraction of the swarm is wiped out simultaneously, this process should allow an organic healing process to occur and it is encouraged that the default client behavior should be to repair any damage detected. In order to prevent damage, nodes can conduct integrity audits at regular intervals that detect loss and initiate repair.

[4] Note that the typical values for k will be in the single digits so a single byte will allways suffice. Note that in the special cornercase when the singleton leftover chunk is a full chunk, we end up having an oversized chunk.

[5] If the filesize is not a multiple of 4096 bytes, then the last chunk at every level will actually have a higher redundancy even than the rest.

[6] If nodes are compensated only for serving chunks, then less popular chunks are less profitable and more likely to be deleted; therefore, if users only download the data chunks and never request the parity chunks, then these are more likely to get deleted and ultimately not be available when they are finally needed. Another approach would be to use non-systemic coding. A systemic code is one in which the data remains intact and we add extra parity data whereas in a non-systemic code we replace all data with parity data such that (in our example) all 128 pieces are really created equal. While the symmetry of the non-systemic approach is appealing, it leads to forced decoding and thus to a high CPU usage even in normal operation. Moreover it breaks random access property of the chunk tree making it impossible to stream media files from the swarm. Luckily the problem is solved by the automated audit scheme which audits the integrity of all chunks and does not distinguish between data or parity chunks.

Improving latecy of retrievals

In the original *Kademlia* ([7]), alpha represented the number of peers (within the relevant Kademlia bin) that are queried simultaneously during a lookup. Setting alpha at 3 (as suggested there) is impractical for swarm because the peers do not report back with new addresses as they would do in pure Kademlia but instead forward all queries to their peers. Swarm is coded this way to make use of semi-stable longer-term devp2p connections. Setting alpha to anything greater than 1 thus increases the amount of network traffic substantially – setting up an exponential cascade of forwarded lookups (but it would soon collapse back down onto the target of the lookup).

However, setting alpha=1 has its own downsides. For instance, lookups can stall if they are forwarded to a dead node or there could be large delays before a query is complete (even if all nodes are live).

In an erasure coded setting we can in a sense have the best of both worlds. Issueing a lookup request not just for the data chunks but for the parity chunks, the client could accept the first m of every 128 chunks queried to get some of the same benefits of faster retrieval that redundant lookups provide without a whole exponential cascade. This only makes sense if the computational overhead when using CRS decoding is shorter than the latency we would otherwise expect if instead of any m chunks, all the m data chunks had to be retrieved. Note that the quadratic complexity of erasure codes with filesize does not apply here since the coding is performed on fixed-size chunks incurring a constant overhead per node.

3.3 Registered nodes and Ensured ARchival (SWEAR)

Once the owner has prepared their data they upload the chunks to the swarm where they are replicated and stored. To decrease the risk that the data will be lost, the owner may purchase storage promises from other nodes as a form of insurance. Before a node can sell these promises of long-term storage however, it must first register via a contract on the blockchain we call the *SWEAR* (Secure Ways of Ensuring ARchival or SWarm Enforcement And Registration) contract. The SWEAR contract allows nodes to register their public key to become accountable participants in the swarm by putting up a deposit. Registration is done by sending the deposit to the SWEAR contract, which serves as colleteral if terms that registered nodes 'swear' to keep are violated (i.e., nodes do not keep their promise to store). *Registration* is valid only for a set period, at the end of which a swarm node is entitled to their deposit. Users of Swarm should be able to count on the loss of deposit as a disincentive against foul play as long as enrolled status is granted. As a result the deposit must not be refunded before the registration expires.

Registration in swarm is not compulsory, it is only necessary if the node wishes to sell promises of storage. Nodes that only charge for retrieval can operate unregistered. The incentive to register and sign promises is that they can be sold for profit. When a peer connection is established, the contract state is queried to check if the remote peer is a registered node. Only registered nodes are allowed to issue valid receipts and charge for storage.

When a registered node receives a request to store a chunk, it can acknowledge accepting it with a signed receipt. It is these signed receipts that are used to enforce penalties for loss of content through the SWEAR (Secure Ways of Ensuring ARchival or SWarm Enforcement And Registration) contract. Because of the locked collateral backing them, the receipts can be viewed as secured promises for storing and serving a particular chunk up until a particular date. It is these receipts that are sold to nodes initiating requests. In some schemes the issuer of a receipt can in turn buy further promises from other nodes pontentially leading to a chain of

local contracts.

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

3.4 Litigation on loss of content (SWINDLE)

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

3.4.1 Submitting a challenge

Nodes provide signed receipts for stored chunks which they are allowed to charge arbitrary amounts for. The pricing and deposit model is discussed in detail in section *Pricing, deposit, accounting*. If a promise is not kept and a chunk is not found in the swarm anyone can report the loss by putting up a *challenge*. The response to a challenge is a *refutation*. Validity of the challenge as well as its refutation need to be easily verifyable by the contract. For now, we can just assume that the litigation is started by the challenge after a user attempts to retrieve insured content and fails to find a chunk. Litigation will be discussed below in the wider context of regular integrity audits of content in the swarm.

The challenge takes the form of a transaction sent to the *SWINDLE* (Secured With INsurance Deposit Litigation and Escrow) relevant swarm contract in which the challenger presents the receipt(s) of the lost chunk. Any node is allowed to send a challenge for a chunk as long as they have a valid receipt for it (not necessarily issued to them).

This is analogous to a court case in which the issuers of the receipts are the defendants who are guilty until proven innocent. Similarly to a court procedure public litigation on the blockchain should be a last resort when the rules are abused despite the deterrents and positive incentives.

The same transaction also sends a deposit covering the upload of a chunk. The contract verifies if the receipt is valid, ie.,

- receipt was signed with the public key of a registered node
- the expiry date of the receipt has not passed
- sufficient funds are sent alongside to compensate the peer for uploading the chunk in case of a refuted challenge

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

A challenge is open for a fixed amount of time, the end of which essentially is the deadline to refute the challenge. The challenge is refuted if the chunk is presented (additional ways are discussed below). Refutation of a challenge is easy to validate by the contract since it only

involves checking that the hash of the presented chunk matches the receipt. This challenge scheme is the simplest way (i) for the defendants to refute the challenge as well as (ii) to make the actual data available for the nodes that needs it.

In normal operation, litigation should be so rare that it may be necessary to introduce a practice of regular *audits* to test nodes' compliance with distribution rules. In such cases the challenge can carry a flag which when set would indicate that providing the actual chunk, (ii) above, is unnecessary. In order to reduce network traffic, in such cases presenting the chunk can be replaced by providing a *proof of custody*. Note that in order not to burden the live chain, audits could happen off chain and they would only make it to the blockchain if foul play is detected. Conversely, if such auditing is a regular automated process, then litigation will typically be initiated as part of escalating a failed audit. *[11]* describes such an audit protocol using the smash proof of custody construct.

3.4.2 The outcome of a challenge

Successful refutation of the challenge is done by anyone sending the chunk or a proof of custody thereof as data within a transaction to the blockchain. Upon verifying the format of the refutation, the contract checks its validity by checking the hash of the chunk payload against the hash that is litigated or validating the proof of custody. If the refutation is valid, the cost of uploading the chunk is compensated from the deposit of the challenge, with the remainder refunded.

In order to prevent DoS attacks, the deposit for compensating the swarm node for uploading the chunk into the blockchain should actually be substantially higher than (e.g., a small integer multiple of) the corresponding gas price used to upload the demanded chunk.

The contract also comes with an accessor for checking that a given node is challenged (potentially liable for penalty), so the accused nodes can get notified to present the chunk in a timely fashion.

If a challenge is refuted within the period the challenge is open, no deposit of any node is touched. After successful refutation the challenge is cleared from the blockchain state.

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

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

If refutation of litigation is found to be common enough, sending transactions is not desirable since it is bloating the blockchain. The audit challenges using the smash proof of custody described in *[11]* enable us to improve on this and make litigation a two step process. Upon finding a missing chunk, the litigation is started by the challenger sending an audit request [7] .

Playing nice is further incentivized if a challenge is allowed to extend the risk of loss to all nodes that have given a promise to store the lost chunk. This means that when one storer is

[7] See *[11]* for the explanation of particular audit types. In fact any audit challenge when fail should be escalated to the blockchain. The smash smart contract provides an interface to check validity of audit requests (as challenges) and verify the various response types (as refutations).

challenged, all nodes that have outstanding receipts covering the (allegedly) lost chunk stand to lose their deposit. Holders of receipts by other swarm nodes can punish them as well for losing the chunk, which, in turn, incentivizes whoever may hold the chunk to present it (and thus refute the challenge) even if they are not the named defendant first accused.

Owners express their preference for storage period. As for storage period, the base unit used will be a *swarm epoch*. The swarm epoch is the minimum interval a swarm node can register for.

Nodes can choose to gamble of course by selling storage receipts without storing the chunk, in the hope of being able to retrieve the chunk from the swarm as needed. However, since storers have no real way to trust other nodes to fall back on, the nodes that issue receipts have a strong incentive to actually store the chunk themselves. Collecting receipts from several nodes therefore means that several replicas are likely to be kept in the swarm. Slogan: more receipts means more redundancy.

A priori this only works, however, in the simplest system in which the owner needs to receive and keep all the receipts signed by the storers.

3.4.3 Publicly accessible receipts and consumer driven litigation

End-users that store important information in the swarm have an obvious interest in keeping the receipt available for litigation. The storage space required for storing a receipt is a sizable fraction of that used for storing the information itself, so end users can reduce their storage requirement further by storing the receipts in Swarm as well. Doing this recursively would result in end users only having to store a single receipt, the root receipt, yet being able to penalize quite a few Swarm nodes, in case only a small part of their stored information is lost.

A typical usecase is when content producers would like to make sure their content is available. This is supported by implementing the process of collecting receipts and putting them together in a format which allows for the easy pairing of chunks and receipts for an entire document. Storing this document-level receipt collection in the swarm has a non-trivial added benefit. If such a pairing is public and accessible, then consumers/downloaders (not only creators/uploaders) of content are able to litigate in case a chunk is missing. On the other hand, if the likely outcome of this process is punishment for the false promise (burning the deposit), motivation to litigate for any particular bit lost is slim.

This pattern can be further extended to apply to a document collection (dapp/website level). Here all document-level root receipts (of the sort just discussed) can simply be included as metadata in the manifest entry for the document alongside its root hash. Therefore a manifest file itself can store its own warranty. The question arises what happens if the hash of this entire collection is not found, if this is a possibility then all the effort in insuring the chunks was futile [8] .

[8] One proposal is to introduce a special content addressed storage, whereby litigation information (notably the receipt from the guardian) is stored at an address derivable from the swarm hash. The address would be derived from the hash by flipping its first bit which would guarantee that the receipt is stored at an opposite end of the swarm. This would make litigation on the chunk level independent of document-level structures and would allow any third party to initiate audits and litigation against a loss chunk knowing only the hash. It is unclear whether this would work though: if a chunk is not found due to it not having been retrieved for some time, chances are high that the receipt has also not been accessed and has been deleted too.

3.5 Receipt forwarding or chained challenges

In this section we zoom in on the swapping and elaborate how owners initiate storage requests, how chunks find their storers and how information is passed around between peers so that it creates an incentive compatible resilient system with last resort litigation.

3.5.1 Forwarding chunks

In normal swarm operation, chunks are worth storing because of the possibility that they can be profitably sold by serving retrieve requests in the future. The probability of retrieve requests for a particular chunk depends on the chunk's popularity and also, crucially, on the proximity to the node's address.

Nodes are expected to forward all chunks to nodes whose address is closer to the chunk. This *forwarding* is the normal syncing protocol. It is compatible with the pay-for-retrieval incentivisation: once a retrieve request reaches a node, the node will either sell the chunk (if it has it) or it will pass on the retrieve request to a closer node. There is no financial loss from syncing chunks to closer nodes because once a retrieve request reaches a closer node, it will not be passed back out, it will only be passed closer. In other words, syncing only serves those retrieve requests that the node would never have profited from anyway and thus it causes no financial harm due to lost revenue.

For insured chunks, a similar logic applies - even more so because there is a positive incentive to sync. If a chunk does not reach its closest nodes in the swarm before someone issues a retrieval request, then the chances of the lookup failing increase and with it the chances of the chunk being reported as lost. The resulting litigation poses a burden on all swarm nodes that have ever issued a receipt for the chunk and therefore incentivises nodes to do timely forwarding. The audit process described in *[11]* provides additional guarantees that chunks are forwarded all the way to the proximate nodes.

Swarm assumes that nodes are content agnostic, i.e., whether a node accepts a new chunk for storage should depend only on their storage capacity [9] . Registered nodes have the option to indicate that they are full capacity. This effectively means they are temporarily not able to issue receipts so in the eyes of connected peers they will act as unregistered. As a result, when syncing to registered nodes, we do not take no for an answer: all chunks legitimately sent to a registered node can be considered receipted. If the node already has the chunk (received it earlier from another peer), the receipt is not paid for.

The purpose of the receipt is to prove that a node closer to the target chunk than the node itself received the chunk and will either store it or forward it. This is exactly what synchronisation does, therefore, proving (in)correct synchronisation is a potential substitute for receipt based litigation. If we further stipulate that registered nodes need to sign sync state and able to prove a particular chunk was part of the synced batch, we can get away without storing individual receipts altogether. Instead we implement the persistence of receipts as part of the chunkstore mechanism on the one hand and the passing of receipts as part of the syncing mechanism on the other.

An advantage of using sync state as receipt is that when litigation takes place, one can point fingers to a node which already had the chunk at the time of syncing as long as it is registered.

[9] We will use a double masking techique as a basic measure to ensure plausible deniability.

3.5.2 Collecting storer receipts and direct contracts

There are a few schemes we may employ. In the first, a storage request is forwarded from node to node until it reaches a registered node close to the chunk address. This storer node then issues a receipt which is passed back along the same route to the chunk owner. The owner then can keep these receipts for later litigation.

Explicit direct contracts signed by storers held by owners has a lot of advantages. On top of its transparency and simplicity, this scheme enables owners to make sure that any degree of redundancy (certainty) promise is secured by deposits of distinct nodes via their signed promises. In particular it allows owners to insure their chunks against a total collateral higher than any individual node's deposit. Also insuring a chunk against different deposits for varying periods is easy. Unfortunately, this rather transparent system has caveats.

First of all, forwarding back receipts creates a lot of network traffic. The only purpose of receipts is to be able to use them in litigation, which is very rare, rendering virtually all this traffic spurious. Moreover, passing it back to the owner does not solve the distribution of receipts to third parties who want to initiate litigation in case of a lookup failure.

Secondly, since availability of a storer node cannot always be guaranteed, getting receipts back from storers may incur indefinite delays. The owner (who submits the request) needs a receipt that can be used for litigation later. If this receipt needs to come from the storer, then the process requires an entire roundtrip.

Furthermore, deciding on storers at the time the promise is made has a major drawback. If the storage period is long enough the network may grow and new registered nodes come online in the proximity of the chunk. It can happen that routing at retrieval will bypass this storer. Though syncing makes sure that even in these cases the chunk is passed along and reaches the closest nodes, their accountability regarding this old chunk cannot be guaranteed without further complications.

To summarize, explicit transparent contracts between owner and storer necessitate forwarding back receipts which has the following caveats:

- spurious network traffic
- delayed response
- potential non-accountability after network growth

3.5.3 Chaining challenges

The other model is based on the observation that establishing the link between owner and storer can be delayed, allowing it to take place at the time of litigation. Instead of waiting for receipts issued by storers, the owner direcly contracts their (registered) connected peer(s) and they immediately issue a receipt for storing a chunk.

When registered nodes connect, they are expected to have negotiated a price and from then on are obligated to give receipts for chunks that are sent their way according to the rules. This enables nodes to guarantee successful forwarding and therefore they can immediately issue receipts to the peer they receive the request from. Put in a different way, registered nodes enter into contract implicitly by connecting to the network and syncing.

When issuing a receipt in response to a store request for the first time, a node becomes an entrypoint for a chunk to the world of insured chunks. In this case the node acts as the *guardian* of the chunk in question. The receipt(s) that the owner gets from their connected peer can

be used in a challenge. Since the transaction immediately settles, the owner can *upload and disappear*. The guardian in turn obtains a receipt from the node they are forwarding to and so on creating a chain of contracts all the way to the node proximate to the target chunk, who in turn will act as the *custodian* of the chunk.

When it comes to litigation, the nodes play a blame game; challenged nodes defend themselves not necessarily by presenting the chunk (or proof of custody), but by presenting a receipt for said chunk issued by a registered node closer to the chunk address, a *nearer neighbour*. Thus litigation will involve a chain of challenges with receipts pointing from owner via forwarding nodes all the way to the custodian who must then present the chunk or be punished.

The litigation is thus a recursive process where one way for a node to refute a challenge is to shift responsibility and implicate another node to be the culprit. The idea is that contracts are local between connected peers and blame is shifted along the same route as what the chunk travels during sycing (restricted to registered nodes).

The challenge is constituted in submitting a receipt for the chunk signed by a node. (Once again everybody having a receipt is able to litigate) [10] . Litigation starts with a node submitting a receipt for the chunk that is not found. This will likely be the receipt(s) that the owner received directly from the guardian. The node implicated can refute the challenge by sending either the direct refutation (audit response or the chunk itself depending on the size and stage) to the blockchain as explained above or sending a receipt for the chunk signed by another node. This receipt needs to be issued by a nearer neighbour (a registered peer closer to the chunk address than the node itself). In other words, if a node is accused with a receipt, it needs to provide a valid receipt from a nearer neighbour. These validations are easy to carry out, so verification of chained challenges is perfectly feasible to add to the smart contract.

If a node is unable to produce either the refutation or the receipts, it is considered a proof that the node had the chunk, should have kept it but deleted it. This process will end up blaming the custodians for the loss. If syncronisation was correctly followed and all the nodes forwarding kept their receipt, then eventually the blame will point to the node that was closest to the chunk to be stored at the time the request was received. If an audit request for a chunk is not responded to, the audit request is delegated to the guardian, and travels the same trajectory as that the original store request (see figure 2). Analogously, if a chunk is not found and the case is escalated to litigation on the blockchain, then finger pointing will also follow the same path (see figure 3) [11] .

When the network grows, it can happen that a custodian finds a new registered node closer to its chunk. This means they need to forward the original store request, the moment they obtain a receipt they can use it in finger pointing, they cease to be custodians and the ball is in the new custodian's court. Such change of custodian can also happen if you buy receipts from a node whose membership expires before the storage period of the insurance ends or simply suspended. In these cases the chunk will have a new custodian. It turns out that chained receipting very elegantly solves the problem of dynamic functional roles that is necessitated by dropouts, new nodes as well as variable membership terms. With the direct owner–storer contracting scheme discussed above this would still need to be solved.

[10] There is no measure to prevent double receipting, i.e., the same node can sell storage insurance about the same chunks to different parties.

[11] In the latter case the transaction is more metaphorical, finger pointing is mediated by state changes in the blockchain: when a node gets notified of a challenge (via a log event) they are sending in their receipts as a refutation and as a result the new closer node gets challenged.

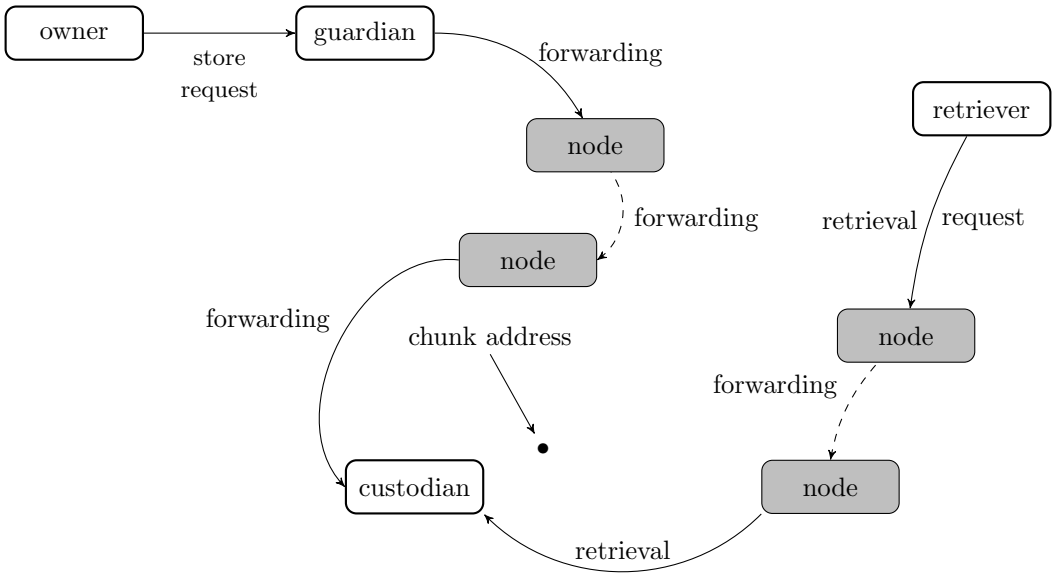


Fig. 2: The arrows represent local transactions between connected peers. In normal operation these transactions involve the farther nodes (1) sending store request (2) receiving delivery request (3) sending chunk (4) sending payment (5) receiving a receipt.

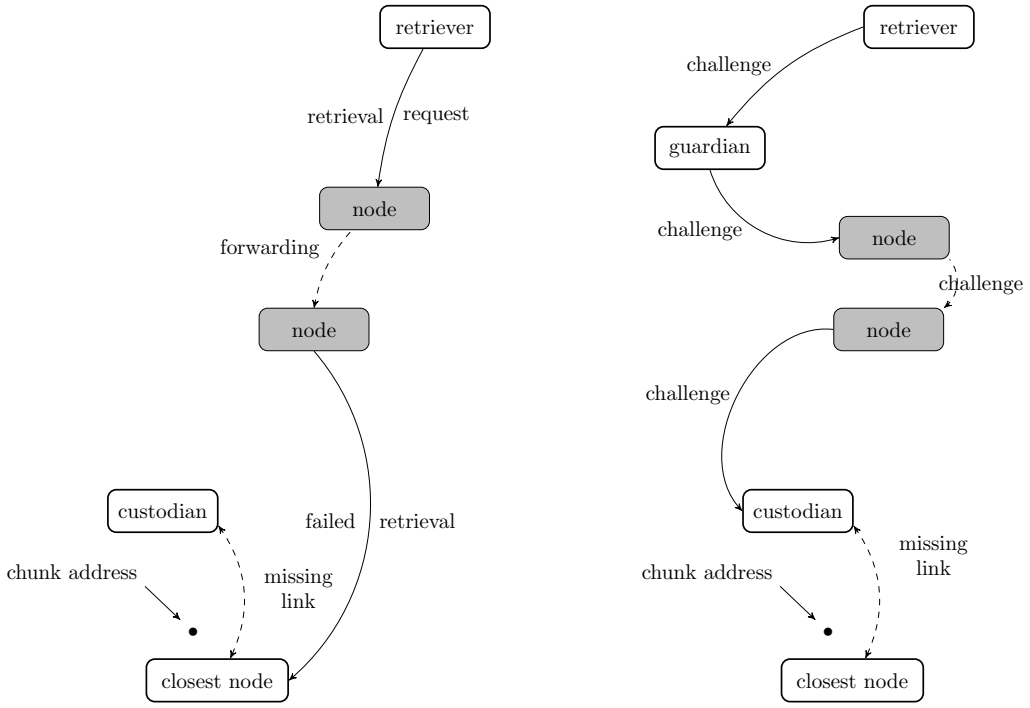


Fig. 3: The arrows represent local transactions between connected peers. Following a failed lookup (left), the guardian is sent an audit/request and the edges correspond to audit requests forwarded to the peer that the node originally got the receipt from (right). Analogously, when a case is escalated to litigation on the blockchain, the chain of challenges follow the same trajectory.

3.5.4 Multiple receipts and local replication

As discussed above, owners can manage the desired security level by using erasure coding with arbitrary degree of redundancy. However, it still makes sense to require that more than one node actually store the chunk. Although the cloud industry is trying to get away from the explicit x-fold redundancy model because it is very wasteful and incurs high costs – erasure coding can guarantee the same level of security using only a fraction of the storage space. However, in a data center, redundancy is interpreted in the context of hard drives whose failure rates are low, independent and predictable and their connectivity is almost guaranteed at highest possible speed due to proximity. In a peer-to-peer network scenario, nodes could disappear much more frequently than hard drives fail. In order to guarantee robust operation, we need to require several local replicas of each chunk (commonly 3, see *[15]*). Since the syncing protocol already provides replication across the proximate bin, regular resyncing of the chunk may be sufficient to ensure availability in case the custodian drops off. If this proves too weak in practice we may require the custodian to get receipts from two proximate peers who act as cocustodians. The added benefit of this extra complexity is unclear.

3.6 Pricing, deposit, accounting

In this section we explore the pricing, accounting and settlement of storage services. We conclude that the fully featured version of the SWAP protocol is ideal to manage both unregistered use as well as registered use, delayed as well as immediate payments.

3.6.1 Pricing

We posited in the introduction that registered nodes should be allowed to compete in quality of service and factor their certainty of storage in their prices. Market pricing of storage is all the more important once we realise that unlike gas, system-wide fixed storage price is neither easy nor necessary.

Gas is the accounting unit of computation on the ethereum blockchain, it is paid in as ether sent with the transaction and paid out in ether to the miner as part of the protocol. The actual price of gas for a block is fixed system-wide yet it is dictated by the market. It needs to be fixed since accounting for computation needs to be identical across all nodes of the network. It still can be dictated by the market since the miners the providers of the service gas is supposed to pay for, have a way to 'vote' on it. Miners of a block can change the gas price (based on how full the block is) [12] . Also such a mechanism of voting by service providers is not available. Note that in principle there is some information on the blockchain which could be used to inform prices: the number of (successful) litigations. If there is an increase in the percentage of litigations (number of proven charges normalised by the number of registered nodes), that is indication that system capacity is lower than the demand, therefore prices need to rise. The other direction, however, when prices need to decrease has no such indicator: due to the floor effect of no litigation (quite expected normal operation), information on the blockchain is inconsequential as to whether the storage is overpriced.

Hence we conclude that fixed pricing of storage is not viable without central authority or trusted third parties. Instead we assume that storage price is negotiated between peers and accepting the protocol handshake and establishing the swarm connection implicitly constitutes an arrangement.

[12] To mitigate against extreme price volatilty, one can regulate the price by introducing restrictions on rate of change (absolute upper limit of percentage of change allowed from block to block).

3.6.2 Deposit

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

Assuming s is a system-wide security constant dictating the ratio between price and deposit staked in case of loss, for an advertised price of p, the minimum deposit [13] is $d = s \cdot p$. Price per chunk per epoch is freely configurable and dictated by supply and demand in the free market. Nodes are free to follow any price oracle or form cartels agreeing on price. Finally variable deposits are inherently at odds with a chained more of operation (local contracting with forwarding).

3.6.3 Accounting and settlement

In the context of contractual agreements, forwarding of a chunk is equivalent to subcontracting for service provision that has a price. Since receipts are promises about the future, it is not in the interest of the buyer to pay before the promise is proved to have been kept. However, delayed payments without locked funds leave storers vulnerable to non-payment.

In order to lock funds nodes could use an escrow contract on the blockchain, however, burdening the blockchain with pairwise accounting is unnecessary. With a *two-way payment channel*, the parties can safely lock parts of their balance as well as do accounting off chain.

Advance payments (i.e., payment settled at the time of contracting, not after the storage period ends) on the other hand, leave the buyers vulnerable to cheating. Without limiting the total value of receipts that nodes can sell, a malicious node can collect more than their deposit and disappear. Having forfeited their deposit, they still walk away with a profit even though they broke their promise. Given a network size and a relatively steady demand for insured storage (in chunk epoch), the deposit could be set sufficiently high so this attack is no longer economical [14] .

Another idea is to allow payment by installments, which would similarly keep the total income under a threshold. However, this means that the validity of a receipt can no longer be established, since non-payment of any of the obligations would void the contract.

We can combine the best of both worlds. On the one hand we can lock the total price of storing a chunk for the entire storage period, and tie the release of funds to an escrow condition. This

[13] Although it never matters if the deposit is above the minimum, but it can happen that a peer wants to lower their price without liquidating their funds in anticipation of an opportunity to raise prices in the future.

[14] This could be further improved by enforcing a fixed maximum total value of receipts one node can issue. Without central registry, we need to rely on the receipts. We stipulate that receipts issued by storers contain their cumulative volume of receipted promises (counted in chunk-epoch). They would also report that number to the blockchain every epoch and keep it under a threshold. The node is incentivised to underreport this number but that can be detected and punished (any node who received a higher number, sends their receipt to the blockchain). Likewise, it can also be detected if the node issued two subsequent receipts with non-increasing ranges, hence the current volume can be considered trusted. In the special case that each chunk is insured for the same length period, the current value of insured storage (counted in chunk-epochs) can be calculated since volume = cumulative volume - cumulative expired volume. Thanks to Nick Johnson for proposing this idea.

eliminates the storer's distrust due to potential insolvency of the cheque's issuer. As long as funds are locked and the escrow condition is acceptable for the storer, the settlement is immediate and the storer (guardian, forwarder, custodian) party can safely issue a receipt for the entire storage period. Since payment is delayed it is no longer possible to collect funds before the work is complete, which eliminates a *collect-and-run attack* entirely. Release of locked funds in installments can be tied to audits via the escrow release conditions, i.e., the installment is released on the condition that the node provides a proof of custody.

The enhanced version of the SWAP protocol uses a fully-fledged state-channel/payment channel beside the chequebook and is a perfect candidate for implementing these features. The blockchain implementation and configuration of the payment channel, registration and litigation is discussed in a forthcoming paper (*[12]*).

To conclude the section on storage incentives we summarize the various modes of operation particpants may choose to demand and supply content availability.

The owner does not need to be a registered, guardians, auditors, forwarders and custodians do. On all levels (chunk, document, collection), an owner can choose to take on the role of auditor and (therefore no need for guardian) and store whatever metadata they need for the proof of custody. If the content is of public (dis)interest, the owner can publish the receipt with the content hash so that third party consumers can litigate in case of chunk loss. Owners may wish to preserve content for long periods of time without retrieving the content but for reasons of increased liquidity allow the storer to withdraw in installments. Similarly, if an owner wishes to renew a storage agreement after it expired, payout needs to happen without the owner wanting to see the data. To prevent collect-and-run storers, in all these cases payout need to be tied to proof of custody as an escrow condition. Simple merkle proof challenge is available, infinitely repeatable, only needs to remember the root hash and are logarithmic in network traffic. Auditing with simple Merkle proofs is not outsourceable in the strict sense, if the owner want to upload and disappear, the only way the auditor can prove the audits retrospectively to the owner is by recording them on the blockchain. If repeated this ends up paying at least twice the price of storage on the blockchain losing on transaction costs as well on the way. The solution is to pregenerate seeds and precalculate a secret by applying some irreversible function of the seed and the proof of custody. The network traffic can be reduced to constant per chunk at the cost of storing precalculated audit secrets. The secrets can be forgotten if their hashes are remembered, these can be published so third parties can verify audits. Owners can outsource the storage of these masked secrets safely, notably to storers themselves who can conversely prove to the owner their secret is correct. This use trades storage for network traffic, but since each new audit requires constant storage, it does not scale for fixed chunks. Owners can mitigate this by packaging entire collections under the same seeds and calculate single secrets, this way any storage period can be covered by any desired rate of audit as long as there is enough documents bundled together. More realistically, owners can hand over an arbitrary sized collection (or document or chunk) to third party insurers who aggregate them, generate seeds, conduct audits, guarantee and prove to the owner the integrity of their content according to the agreement. See *[11]* for details about the audit schemes and their capabilities.

This is where we stand at the moment. If this line withstands expert scrutiny, in a forthcoming publication (*[12]*) we hope to put together the pieces of this puzzle and offer a formal specification of the above modes of operation backed up with smart contracts providing accounting and finance (swap), registration and deposit handling (swear) and auditing, litigation and defence (swindle).

4 Conclusion

This paper explored ways of incentivising smooth operation in a peer to peer document storage and content delivery system and honed in on a particular proposal for swarm, an ethereum base layer service. Our approach uses SWAP, the Swarm Accounting Protocol to do pairwise accounting of micropayments relevant in charging for bandwidth. The channel allows swapping service for service in chunk retrieval and allows joining the network without funds. A chequebook contract is used to issue cheques as instruments of delayed payment, which can be cashed by the counterparty at any point to redeem promised funds as long as the sender is solvent. Data preservation in long term storage is incentivised on an individual level both by compensation as well as penalty in case of chunk loss. The loss of insured tokens is a major offence punishable by suspension of account and forfeiture of application-global deposit. Various ways of escrow conditions on the release of funds are able to capture quite a few usecases including pay in installments depending on successful audit. Swarm secures storage with proof of custody audits, valid audits can be tied to escrow conditions of delayed payments. We only need to combine this with locked funds, immediate settlement and receipting to allow users to simply upload and disappear. We presented a technique of erasure coding applicable to the swarm hash tree, which makes it possible for clients to manage the level of storage security and redundancy within their own remit.

5 Glossary

owner node that produces/originates content by sending a store request

storer node that accepted a store request and stores the content

guardian the first node to accept a store request of a chunk/

custodian node that has no online peer that is closer to a chunk address

auditor node that initiates an audit by sending an audit request

insurer node that is commissioned by an owner to launch audit requests on their behalf

swear the component of the system that handles membership registration, terms of membership and security deposit

swindle the component of the system that handles audits and escalates to litigation

swap information exchange between connected peers relating to contracting, request forwarding, accounting and payment

SWAP Swarm Accounting Protocol, Secured With Automated PayIns. And the name of the suite of smart contracts on the blockchain handling delayed payments, payment channels, escrowed obligations, manage debt etc.

SWEAR Storage With Enforced Archiving Rules or Swarm Enforcement And Registration the smart contract on the ethereum blockchain which coordinates registration, handles deposits and verifies challenges and their refutations

sworn node, registered node, swarm member a node which registered via the SWEAR contract and is able to issue storage receipts until the expire of its membership

suspension punative measure that terminates a node's registered status and burns all available deposit locked in the SWAR contract after paying out all compensation

registration nodes can register their public key in the SWEAR contract by sending a transaction with deposit and parameters to the SWEAR contract they will have an entry

audit special form of litigation where possession of a chunk is proved by proof of custody. The litigation does not stop but forces node to iteratively prove they synced according to the rules.

SWINDLE Secured With INsurance Deposit Litigation and Escrow the module in the client code that drives the iterative litigation procedure, initiates litigation in case loss of a chunk is detected and respond with refutation if the node itself is challenged.

proof of custody A cryptographic construct that can prove the possession of data without revealing it. Various schemes offer different properties in terms of compactness, repeatability, outsourceability.

audit An integrity audit is a protocol to request and provide proof of custody of a chunk, document or collection.

Erasure codes An error-correcting scheme to redundantly recode and distribute data so that it is recovered in full integrity even if parts of it are not available.

bzz protocol The devp2p network communication protocol Swarm uses to exchange information with connected peers.

chequebook contract A smart contract on the blockchain that handles future obligations, by issuing signed cheques redeemable to the beneficiary.

syncing The protocol that makes sure chunks are distributed properly finding their custodians as the node closest to the chunk's address.

deposit The amount of ether that registered nodes need to lock up to serve as collateral in case they are proven to break the rules (lose a litigation).

registration Swarm nodes need to register their public key and collateralise their service if they are to issue receipts for storage insurance.

litigation A process of challenge response mediated by the blockchain which is initiated if a node is found suspect not to keep to their obligation (to store a chunk). The idea is that both challenge and its refutation is validated by a smart contract which can execute the terms agreed in the breached contract or any condition of service delivery.

chunk A fixed-sized datablob, the underlying unit of storage in swarm. Documents input to the API are split into chunks and recoded as a Merkle tree, each node corresponding to a chunk.

content addressing A scheme whereby certain information about a content is index by the content itself (with the hash of the content).

receipt signed acknowledgement of receiving (or already having) a chunk.

SMASH proof Secured with Masked Audit Secret Hash: a family of proof of custody schemes

CRASH proof Collective Recursive Audit Secret Hash: a proof of custody scheme for collections

References

[1] Johannes Bloemer, Malik Kalfane, Richard Karp, Marek Karpinski, Michael Luby, and David Zuckerman. An xor-based erasure-resilient coding scheme. Technical Report, International Computer Science Institute, 1995. Technical Report TR-95-048.

[2] Vitalik Buterin. Secret sharing and erasure coding: a guide for the aspiring dropbox decentralizer. 2014. URL: https://blog.ethereum.org/2014/08/16/secret-sharing-erasure-coding-guide-aspiring-dropbox-decentralizer.

[3] Filecoin. Filecoin: a cryptocurrency operated file storage network. 2014. URL: http://filecoin.io/filecoin.pdf.

[4] Mainak Ghosh, M. Richardson, B. Ford, and R. Jansen. A TorPath to TorCoin: proof-of-bandwidth altcoins for compensating relays. Technical Report, petsymposium, 2014. URL: https://www.petsymposium.org/2014/papers/Ghosh.pdf.

[5] IPFS. Interplanetary file system. 2014. URL: http://ipfs.io/ipfs.pdf.

[6] Rob Jansen, Andrew Miller, Paul Syverson, and Bryan Ford. From onions to shallots: rewarding tor relays with TEARS. Technical Report, DTIC Document, 2014.

[7] Petar Maymounkov and David Mazieres. Kademlia: a peer-to-peer information system based on the xor metric. In *Peer-to-Peer Systems*, pages 53–65. Springer, 2002.

[8] Andrew Miller, Ari Juels, Elaine Shi, Bryan Parno, and Jonathan Katz. Permacoin: repurposing bitcoin work for data preservation. In *Security and Privacy (SP), 2014 IEEE Symposium on*, 475–490. IEEE, 2014. URL: https://www.cs.umd.edu/~elaine/docs/permacoin.pdf.

[9] James S Plank, Jianqiang Luo, Catherine D Schuman, Lihao Xu, Zooko Wilcox-O'Hearn, and others. A performance evaluation and examination of open-source erasure coding libraries for storage. In *FAST*, volume 9, 253–265. 2009.

[10] James S Plank and Lihao Xu. Optimizing Cauchy Reed-Solomon codes for fault-tolerant network storage applications. In *Network Computing and Applications, 2006. NCA 2006. Fifth IEEE International Symposium on*, 173–180. IEEE, 2006.

[11] Viktor Tron and Aron Fischer. Smash-proof: auditable storage in swarm secured by masked audit secret hash. Technical Report, Ethersphere, 2016. Ethersphere Orange Papers 2. URL: bzz://ethersphere.sw/orange-papers/2/smash.pdf.

[12] Viktor Tron, Aron Fischer, and Daniel A Nagy. State channels on swap networks: claims and obligations on and off the blockchain (tentative title). Technical Report, Ethersphere, 2016. Ethersphere Orange Papers (to be published). URL: bzz://ethersphere.sw/orange-papers/3/state-channels.pdf.

[13] David Vorick and Luke Champine. Sia: simple decentralized storage. Technical Report, Sia, 2014. URL: http://www.siacoin.com/whitepaper.pdf.

[14] Shawn Wilkinson, Tome Boshevski, Josh Brandoff, and Vitalik Buterin. Storj: a peer-to-peer cloud storage network. Technical Report, storj, 2014. v1.01. URL: https://storj.io/storj.pdf.

[15] Shawn Wilkinson and Jim Lowry. Metadisk a blockchain-based decentralized file storage application. Technical Report, Metadisk, 2014. URL: http://metadisk.org/metadisk.pdf.