Future-proof Storage

economic incentives for sharing disc space in the Swarm peer to peer network

The Swarm Authors[*], [†]

v4.20 - April 2023

Abstract

Swarm is a peer-to-peer network of nodes that collectively provide a decentralised storage and communication solution. Its built-in incentive system is enforced through smart contracts on the Ethereum blockchain and powered by the BZZ token. While individual nodes are assumed to pursue selfish strategies that maximise their operator's profit, the behaviour of the network as a whole attains emergent properties in alignment with the requirements of such a cloud service.

In this paper, we present a novel solution to make decentralised storage economically self-sustaining. First, we introduce Swarm's basic DISC model of storage and distribution with incentives for bandwidth sharing. Then we describe the system of postage stamps as a costly signal that lets users indicate priority of storage. The smart contract that implements this system lets users purchase postage stamps in batches and accumulates the revenue which serves as the pot to redistribute among storage providers to incentivise the contribution of their disc space. This redistribution is managed by a set of smart contracts implementing a system of probabilistic outpayments, called the redistribution game. Genuine storers can claim their reward through the contract. Part of the evidence that storers provide as proof of entitlement to the reward can be interpreted as signals of demand and supply. These are responded to by automatic price updates, which makes, in turn, storage provision a self-regulating market.

[*]Contributors are Viktor Trón, Ábel Bodó, Rinke Hendriksen, Dániel A. Nagy, Daniel Nickless, Gyuri Barabás, Viktor Tóth, Mark Bliss, Callum Toner, Peter Mrekaj, Esad Akar, Alok Nerurkar, Anatol Lupacescu, Áron Fischer.

[†]Authors thank Silur Houhuli for their feedback and review of this paper.

Contents

List of Figures

List of Tables

In section 1, we introduce Swarm's DISC model to set the scene for the incentives. Section 2 discusses postage stamps from the users and the ecosystem's perspective. In section 3 we describe a redistribution scheme that enables direct payout from the proceeds of the postage stamps and thereby distributes postage revenue to storage providers fairly.

1 The DISC model

In this section we briefly outline Swarm's underlying storage model, and discuss how rewards for bandwidth sharing already serve as an incentive for storing popular content. We conclude by stating the need for a separate subsystem to reward the sharing of spare storage capacity (i.e., disc space) to preserve content rarely requested.

1.1 Peer-to-peer storage network

The DISC (*Distributed Immutable Store of Chunks*) is the storage layer underlying Swarm. The canonical unit of storage in Swarm is called a *chunk* (see definition 9 in appendix A.1). A chunk consists of at most 4 kilobytes of data and has an address. Chunks are created through a higher-level upload mechanism that splits files and generates an index in such a way that the files can be reassembled upon download. Since a chunk's address is taken from the same address space as a node's address, it is possible to compute their *proximity* using an xor-based distance metric (see definition 6 in appendix A.1). The DISC prescribes that each chunk is stored by the nodes with an address that is close to that of the chunk itself. Nodes within some agreed proximity form a *neighbourhood* and share all chunks falling in the address range they cover.

To insert a chunk into the swarm, nodes relay the chunk via the *push-sync protocol* until it arrives at the neighbourhood it belongs to. A *receipt* confirming the storage of the chunk is then passed back to the uploader along the same route. In order to retrieve a chunk, a request with a chunk address is routed towards the relevant neighbourhood using the retrieval protocol. If any node encountered on the way already has the corresponding chunk in their local storage, it is sent back as a response. If peers in the network maintain a connectivity pattern according to a specific topology, such a peer will always exist unless the uploading peer is itself the closest. Therefore, successively relaying to a more proximate peer constitutes a routing strategy that finds a way to the node closest to any address. Routing from anywhere to the neighbourhood a chunk belongs to provides the basis for the storage and retrieval of chunks in swarm.

Neighbours continuously synchronise their chunk storage using the *pull-sync protocol.* This guarantees that all chunks that belong to a neighbourhood are stored by each neighbour in it. The redundancy this confers adds resilience in that chunks will remain available even if some of the nodes in the neighbourhood become unreachable. The synchronisation protocol also ensures that the neighbourhood's storage remains consistent as nodes go offline and new nodes join the network.

Automatic scaling of distribution is enabled because nodes who participate in routing retrieval requests may choose to retain some of the chunks they have relayed back along the request forwarding path. Economic motivation for such *opportunistic caching* is provided by the bandwidth incentives.

As nodes relay requests and responses, they keep track of their relative consumption of bandwidth with each of their peers. Within bounds, peers engage in a *service-for-service exchange.*

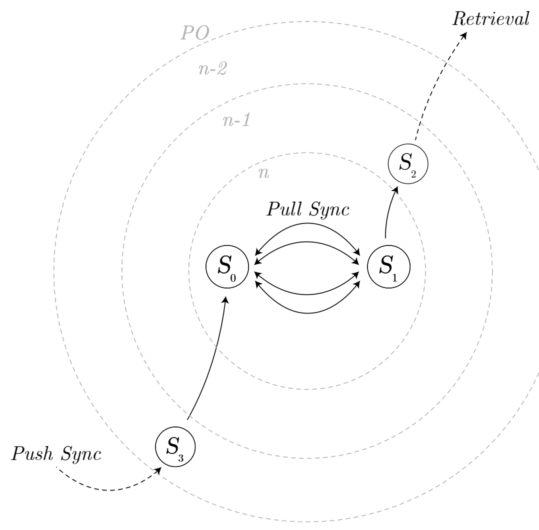

Figure 1: Push, Pull and Retrieve Protocols. The figure depicts the typical trajectory of messages. As the uploader (U) uploads a file, the chunker splits it into chunks which the push sync protocol then transports to their storer(s) (S) in their respective neighbourhood. The pull sync protocol makes sure that a neighbourhood shares all the chunks they are responsible for with the other nodes that comprise that neighbourhood.

However, once a limit is reached, the party in debt can either wait until their liabilities are amortised over time, or can pay for unthrottled service by sending *cheques* that can be cashed out in BZZ on the blockchain.

1.2 Chunk types

There are two fundamental chunk types: *content-addressed chunks* and *single-owner chunks*.

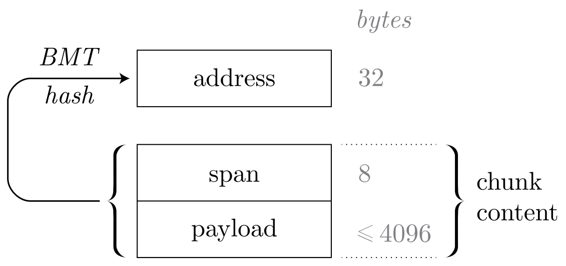

Figure 2: A content addressed chunk has an at most 4KB payload. The address is the Binary Merkle Tree hash of the payload.

The address of content-addressed chunks is based on the hash digest of its data (see figure 2). Using a hash as the chunk address makes it possible to verify the integrity of the chunk's payload. Swarm uses the *BMT hash* function based on a binary Merkle tree over small (32-byte) segments of the chunk data (see definition 8 in appendix A.1). Properties of the base hash (*Keccak256*) used in *BMT* such as *uniformity*, *irreversibility* and *collision resistance* all carry over to the *BMT hash* algorithm. As a result of uniformity, a random set of chunked content will generate addresses evenly spread in the address space, i.e., imposing storage requirements balanced among nodes.

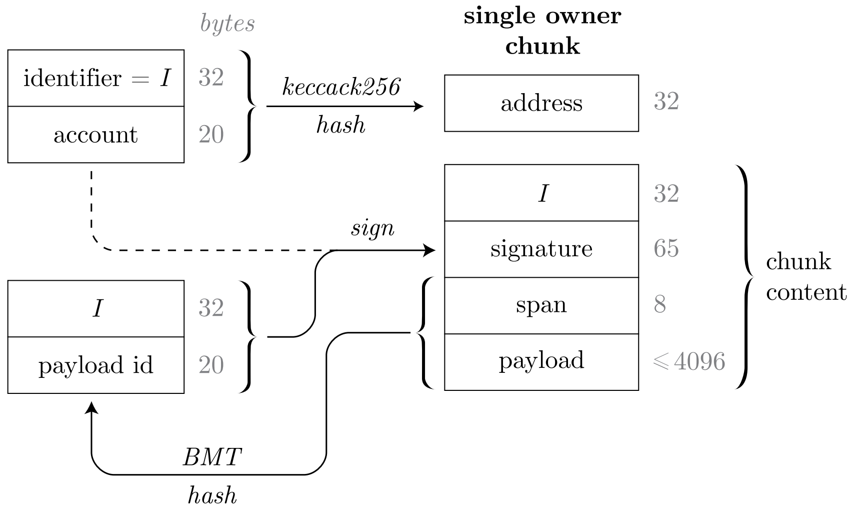

Figure 3: Single-owner chunk. The chunk content is composed of headers followed by the metadata (span) and chunk data of a wrapped content addressed chunk (payload). The first two header fields provide single owner attestation of integrity: an identifier and a signature signing off on the identifier and the content address of the payload. The address is the hash of the id and the signer account.

The address of a single-owner chunk is calculated as the hash of the owner's address and an owner defined identifier. The integrity of single-owner chunk data is warranted by the signature of the owner attesting to the association of arbitrary chunk data with the identifier (see figure 3). In other words, each identity owns part of Swarm's address space within which they are free to assign arbitrary content to an address (see definition 10 in appendix A.1).

1.3 Bandwidth incentives and caching of popular content

Syncing involves transferring chunks from the uploader to storers, i.e., from where they enter the network, to the neighbourhood where the chunk falls within the nodes' *areas of responsibility*. A storer node's role is to serve content by responding to retrieve requests with chunk data.

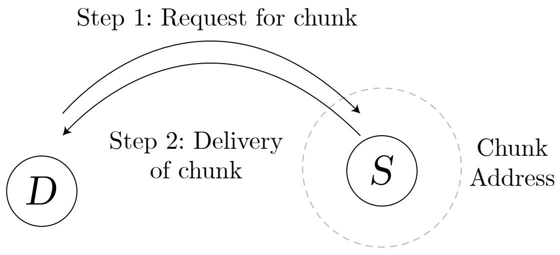

Figure 4: DISC: Distributed Immutable Store of Chunks. Direct address-based peer-to-peer storage model with forwarding Kademlia routing.

If a node on the way does not have the data itself, it is debited a small amount of BZZ to request chunks from an even closer node. If requests are differently priced based on the expected number of hops needed to route the request, a routing strategy based on selecting the lowest price is expected to mimic a proximity-based one.[1] As a consequence of forwarding Kademlia, nodes are also motivated to cache the chunks they serve because, after paying to retrieve the chunk once from a closer node, any subsequent requests for the same chunk will earn pure profit.

As new content gets added to Swarm, sooner or later, the finite storage capacity of each node will be used up. At this point nodes will need a strategy to decide which chunks should be removed to make way for new chunks. The storage cache is pruned regularly when capacity is reached by removing the chunks that were requested the longest time ago. As the recency of the last request is a reasonable predictor of popularity, chunks that generate more income as they are regularly requested will be preferentially retained. Combined with opportunistic caching, this garbage collection strategy both maximises an operator's profit from bandwidth incentives, and, on the network level, ensures that (1) storage capacity is always maximally utilised for increased quality since (2) popular content becomes widely distributed across the network and (3) thus retrieval latency for regularly requested chunks is also decreased.

The closer a node is to the chunk address, the more likely it is that a request for that chunk will end up with them. Therefore, chunks closer to a node's address will be preferentially retained. This creates a weak incentive for storer nodes to sync content. However, it presupposes that the chunk will be the subject of retrieval requests and as such have the promise of some profit. This assumption is not always guaranteed, and is vulnerable to an attack in which an adversary could simply overwhelm the network with chunks that are never requested, and as a result, causing useless chunks to replace useful ones. By attaching a cost to uploading a chunk, Swarm can mitigate such an attack.

1.4 Rarely accessed content

Although effectively providing rewards to storers and forwarders of popular chunks, the flipside of using bandwidth compensation as the only incentive is that chunks that are not accessed for a long time will eventually end up being garbage collected to make room for new arrivals. Therefore, in order for the swarm to be able to guarantee the long-term availability of data, there must be a mechanism with which uploaders can protect their content and prevent their less popular chunks from being prematurely garbage collected.

Moreover, the swarm's incentive system must also ensure that additional revenue is generated for storers of rarely retrieved chunks that would otherwise be deleted. In other words, storers of unpopular chunks that do not generate sufficient profit from retrieval requests only should be compensated for their opportunities forgone. The *postage redistribution* presented in section 3 provides for such compensation: redistribute the revenue resultant of the purchase of *postage stamps* amongst the storer nodes in a fair way.

2 Postage stamps

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

[1]Nodes also are indirectly motivated to forward the chunk to a node which is closer to the chunk and therefore realise higher forwarding profit. Such "fast forwards" (multiple PO per hop) can be done more often by a node that maintains a larger set of peer connections whose addresses are balanced.

to retain and serve, and which ones to garbage collect in the event of capacity shortage.

In this section we first introduce the concept of postage batch enabling the bulk purchase of stamps (2.1). In 2.2, we explain how limited issuance is represented and enforced. In 2.3, we introduce the notion of reserve and detail the rules governing how storer nodes keep it maximally utilised. We conclude in 2.4 with exploring the relationship between reserved capacity, effective demand and the number of nodes and their impact on the data availability.

2.1 Purchasing upload capacity

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

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

The postage contract provides endpoints to users to modify the per-chunk balance of batches, i.e., add funds to extend the validity period of the stamps issued by the batch (*top-up*) or add volume to decrease it (*dilute*). Anyone can then choose to top up the balance of a batch at a later date but only the owner can dilute it.[2]

Owners issue postage stamps in order to attach them to chunks (see definition 17 in appendix A.1). A batch has a number of *storage slots* effectively arranged over a number of equal sized buckets. Issuing a stamp means to assign a chunk to a storage slot. A stamp is a data structure comprising the following fields (see figure 5):

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

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

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

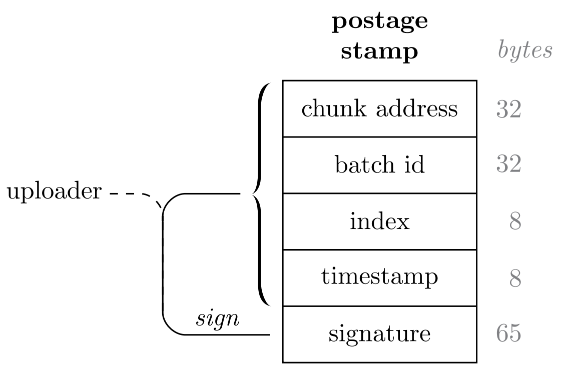

Figure 5: Postage stamp is a data structure comprised of the postage contract batch id, storage slot index, timestamp the chunk address and a witness signature attesting to the association of these four. Uploaders and forwarders must attach a valid postage stamp to every chunk uploaded.

five attributes (see definition 19 in appendix A.1):

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

All this can be easily checked by nodes in the swarm only using information available on the public blockchain (read-only endpoints of the postage contract). When a chunk is uploaded, the validity of attached postage stamp is verified by forwarders along the push-syncing route (see figure 6).

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

This system allows prepayment for storage without having to speculate on the future price of storage or fluctuations in the currency's exchange rate. At the cost of decreased certainty about the expiration date, one gains resilience against price volatility. On top of this, uploaders can enjoy the luxury of non-engagement by tying up more of the batch balance; while it serves as collateral against price increase, if that does not happen the funds can still be used up (for storing).

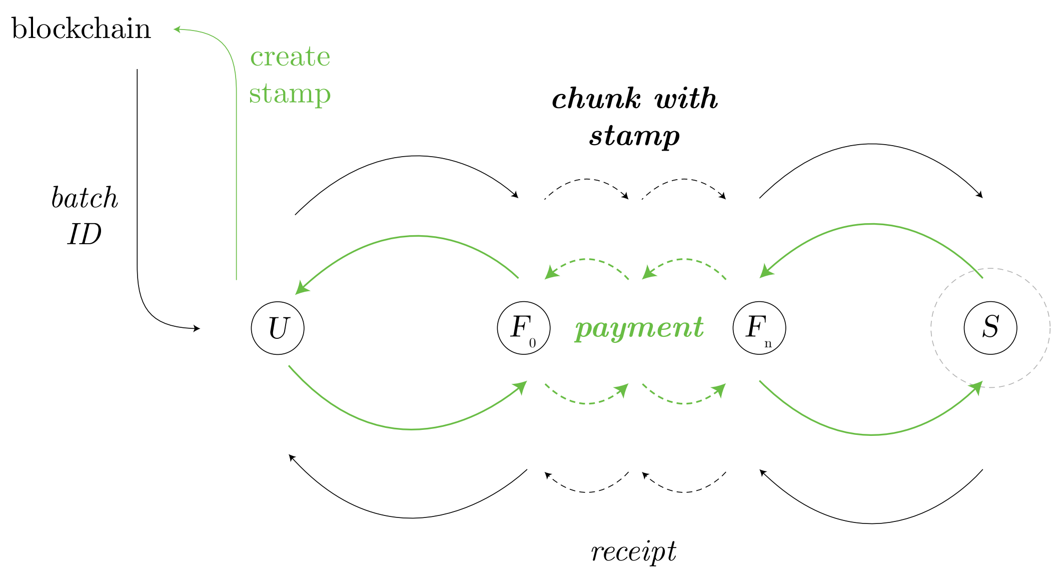

Figure 6: Postage stamps are purchased in bulk on the blockchain and attached to chunks at upload. They are passed along the push-syncing route together and their validity is checked by forwarders at each hop.

2.2 Limited issuance

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

While 1) and 2) is easily verifyable by any third party, 3) is not. In order for index collisions to be detectable by individual storer nodes, uniformity depth must be large enough to fall within nodes' area of responsibility. As long as this is maintained, all chunks in the same bucket are guaranteed to land in the same neighbourhood, and, as a result, duplicate assignments can be locally detected by nodes (see figure 7).

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

In general, the most efficient utilisation of a batch is by filling each bucket fully.[3] Continued non-uniformity (i.e., *targeted issuance*) leads to underutilised batches, and therefore a higher unit price for uploading and storing each chunk. This feature has the desired side effect that it imposes an upfront cost to non-uniform uploads: the more concentrated the distribution of chunks of an upload, the more storage slots of the postage batch remain unused. In this way,

[3]See appendix D for a detailed analysis of batch utilisation.

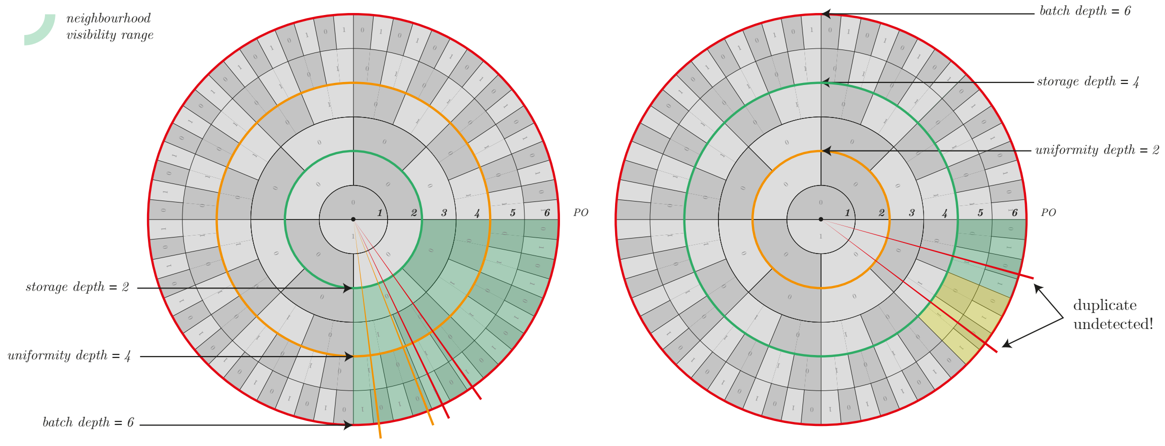

Figure 7: Batches come with 2^u equal-sized buckets (u is uniformity depth, orange circle) each containing an equal number of storage slots (2^{d-u}) adding up to batch capacity of 2^d chunks (d is batch depth, red circle). Storage slots are indexed and the index is associated with a chunk via the stamp signature. Postage stamp over-issuance is detected locally by storer nodes as long as the buckets are deeper than their storage depth (green circle), as in the diagram on the left. In this case they will receive all the chunks that are correctly assigned to the relevant bucket (orange radii) and correctly identify collisions (red radii) by forbidding indexes that are either out of range ($\geq 2^{d-u}$) or multiply assigned. In contrast, the diagram on the right shows it is not possible for a node with storage depth 4 to identify duplicates for a batch with $u = 2$

we ensure that targeted denial-of-service attacks against a neighbourhood (i.e., uploading a disproportionate number of chunks in a particular address range) is costly since the *inert cost* (due to the degree of under-utilisation of the batch) is exponential in the depth of the skew.

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

2.3 Rules of the reserve

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

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

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

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

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

The second rule expresses the constraint that the reserve for a PO is upward closed for per-chunk balance, which encodes a secondary preference among chunks of the same proximity for those stamped using a batch with higher per-chunk balance. This is incentivised by the differential absolute profit chunks promise: due to the constraint that balances are not revocable, chunks with higher balance expire later and therefore contribute more to storers' absolute profit than those expiring earlier despite the same rent paid during their period of validity.[4]

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

2.4 Reserve depth, storage depth, neighbourhood depth

Reserve depth

The potential demand for chunks to be stored in the DISC is quantified by the total storage slots of valid batches. This is calculated as the sum of the sizes of non-expired batches. Since the batches and their balances are recorded in the postage contract, the reserved DISC size is under consensus.[5]

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

The reserve depth is also the *safe lower bound* for pull-syncing, i.e, the farthest bin a neighbour-

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

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

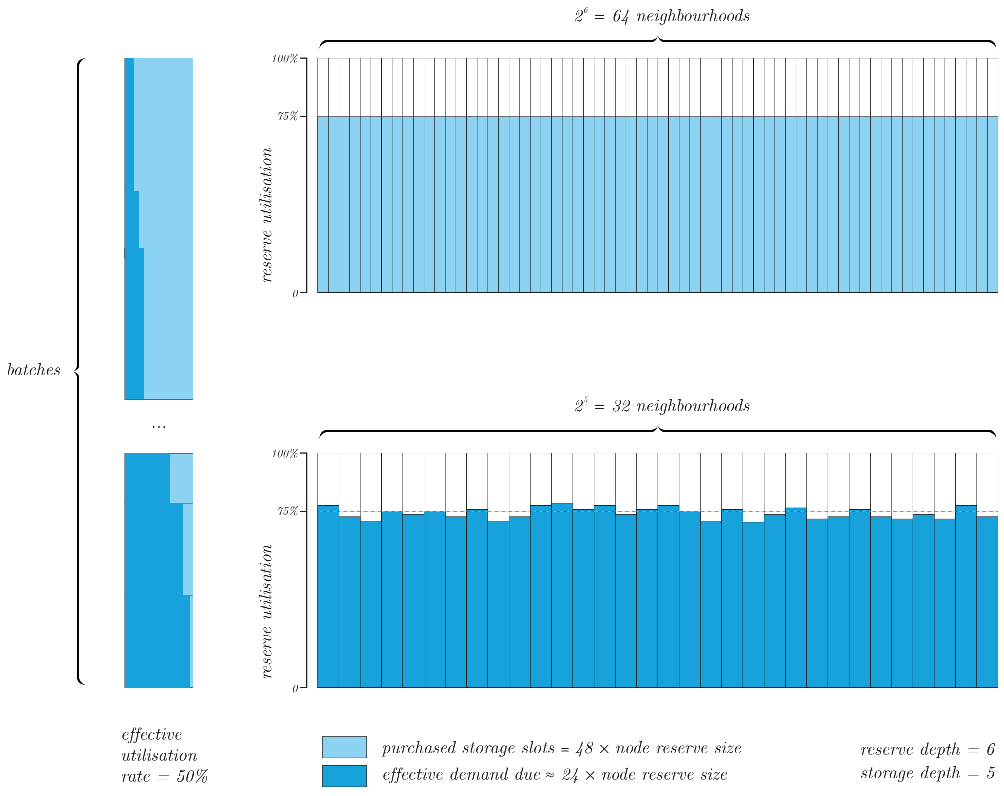

Figure 8: Potential demand for chunk storage is expressed by the total size of all batches with non-zero balance on the blockchain (left). The lower bound on neighbourhood depth to store this capacity is the reserve depth (top right). Storage depth marks the effective volume of chunks uploaded and stored in a neighbourhood's reserve (bottom right). The difference between them is a result of partial batch utilisation. The uniformity of the volume of chunks across neighbourhoods is incentivised by the efficient utilisation of postage batches.

hood needs to synchronise to guarantee storing the reserve. Conversely, if any neighbourhood marked by reserve depth has no nodes in it, the swarm is not working correctly, i.e., chunks with valid stamps are not protected from getting lost. See figure 9.

Storage depth

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

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

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

The storage depth is the *optimal lower bound* for pull-syncing, i.e, the farthest bin the node needs to synchronise with its neighbours to achieve maximum reserve utilisation.[6] Maximum reserve utilisation should be incentivised as part of the storage incentives.

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

Neighbourhood depth

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

Figure 9 details the potential relative orders of the three depths and their consequences on the health, efficiency and redundancy of the swarm.

3 Fair redistribution

The system of positive[7] storage incentives in Swarm is concerned with redistributing to storage providers the BZZ tokens that uploaders deposited within the postage contract.[8] The overall balance on the contract covers the *reward pot* which represents the total *storage rent* cumulated over all postage batches for a particular period. The storage rent must be redistributed to storage providers in a way that guarantees that their earnings are proportional to their contribution weighing in storage space, quality of service and length of commitment.

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

In section 3.1, we formulate the idea of redistribution in terms of probabilistic outpayments to allow an easy proof of fairness. Next, in 3.2, we introduce the mechanics of the redistribution game. Then, in 3.3 and 3.4 we explain how we enforce maximum utilisation of dedicated storage for persisting relevant content redundantly. We conclude in 3.5 by discussing how to read certain

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

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

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

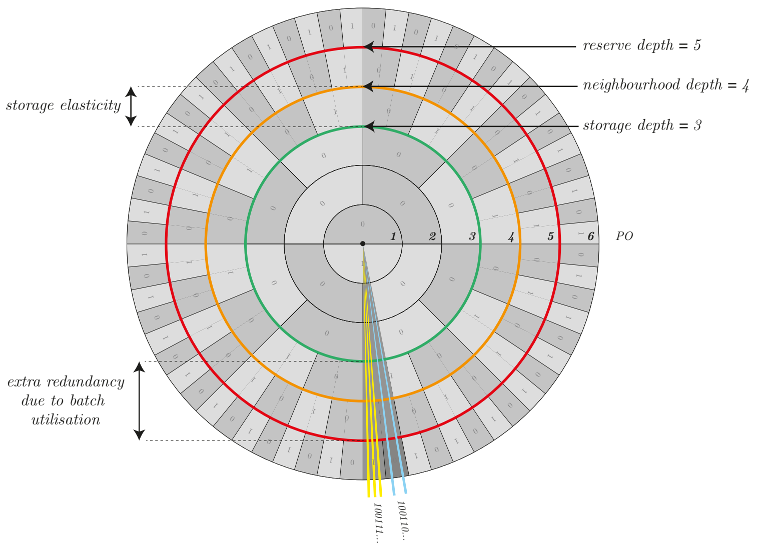

Figure 9: The 3 depths (reserve, storage and neighbourhood) express the order of magnitude of reserved capacity (potential demand, red circle), uploaded chunks (effective demand, green circle) and the number of nodes (effective supply, orange circle), respectively. Their possible orderings express different scenarios with characteristic impact on data availability. Storage depth cannot be greater than reserve depth. A gap between storage depth and reserve depth quantifies the average batch utilisation rate. The gap between storage depth and a deeper neighbourhood depth quantifies the elasticity of the storage: the difference expresses how many times the effective volume can double before redundancy goes below required. While such oversupply may be anticipatory of growth in demand, if neighbourhood depth remains deeper than storage depth long term, it may indicate excessive profits. The opposite order indicates undersupply (redundancy below the desired level).

aspects of the game as price signals that render the network self-regulating through automatic price discovery.

3.1 Neighbourhoods, uniformity and probabilistic outpayments

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

Assuming an oracle that sets the unit price of storage, the storage rent due for a period of time for a batch can be calculated. The number of rent units for a batch is the result of multiplying the size of the batch with the number of blocks in the period. The price of rent is calculated from the number of rent units multiplied by the unit price.[9] The total storage rent cumulated over all batches for the period between two outpayments constitutes the *reward pot* for the round.

Instead of dividing the reward pot among neighbourhoods regularly, the entire reward pot can be transferred to (representative nodes in) one target neighbourhood in each round. This probabilistic outpayment scheme is fair on the level of neighbourhoods as long as we can make sure

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

that over a large number of rounds the probability with which a neighbourhood is selected as the target corresponds to its relative contribution to the overall network storage. Given a constant prescribed reserve capacity and replication of the reserve content by nodes in a neighbourhood, each neighbourhood defined by storage depth contributes equally to the network.

In section 2, we wrote that uploaders are strongly incentivised to use their postage batch in a way that the chunks they stamp with it are uniformly distributed across the address space. This being true of all batches creates a situation that chunks are uniformly distributed across the DISC. In particular, the sets of chunks sharing a prefix are expected to be roughly equal in size. Therefore we expect nodes to fill their prescribed reserve capacity with chunks at the same proximity order, irrespective of their location in the address space, i.e., the storage depth is uniform across nodes and therefore across neighbourhoods.[10] With neighbourhoods at equal depth, uniform sampling of neighbourhoods can be modelled by choosing the neighbourhood which contains an anchor (called the *the neighbourhood selection anchor*[11]) randomly dropped in the address space (see figure 10).

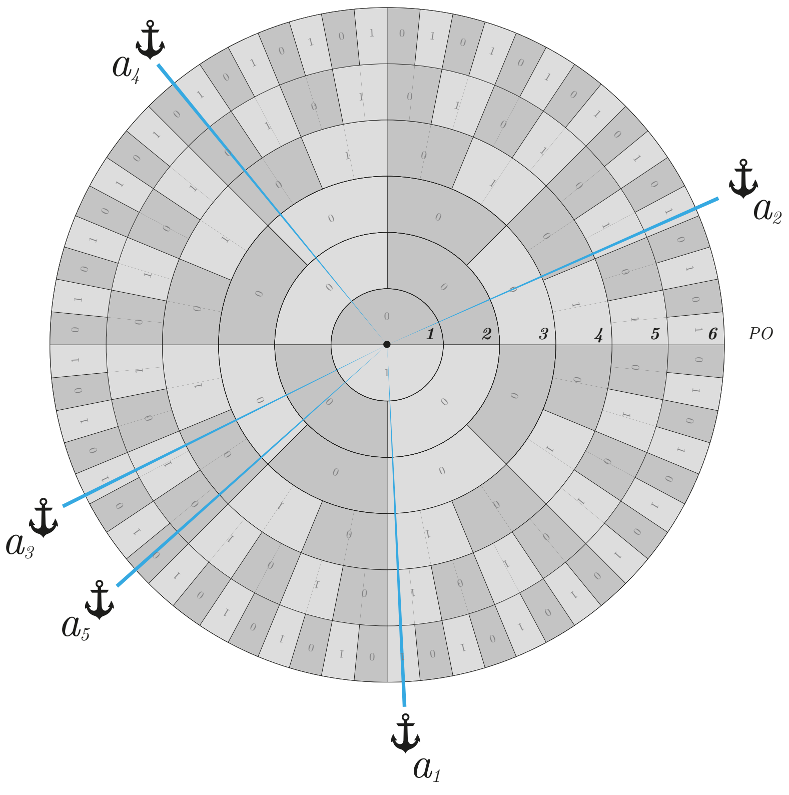

Figure 10: Neighbourhood selection and pot redistribution. The winning locality is selected by the neighbourhood selection anchor. Neighbourhoods that contain the anchor within their storage depth are invited to submit an application by committing to a consensual reserve sample.

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

[11]one of the random nonces (see definition 27) from the *random seed* of the round, see appendix B.

3.2 The mechanics of the redistribution game

The cooperation of peers to redundantly store for the network's benefit is underpinned by a *Schelling game* (defined formally in appendix A.2) aimed at proving that the peers agree on the chunks they need to store and they do, in fact, store them. The redistribution game itself is orchestrated by the game contract, one of the building blocks of the system of 4 smart contracts which collectively drive the swarm storage incentive system (see figure 11):

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

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

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

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

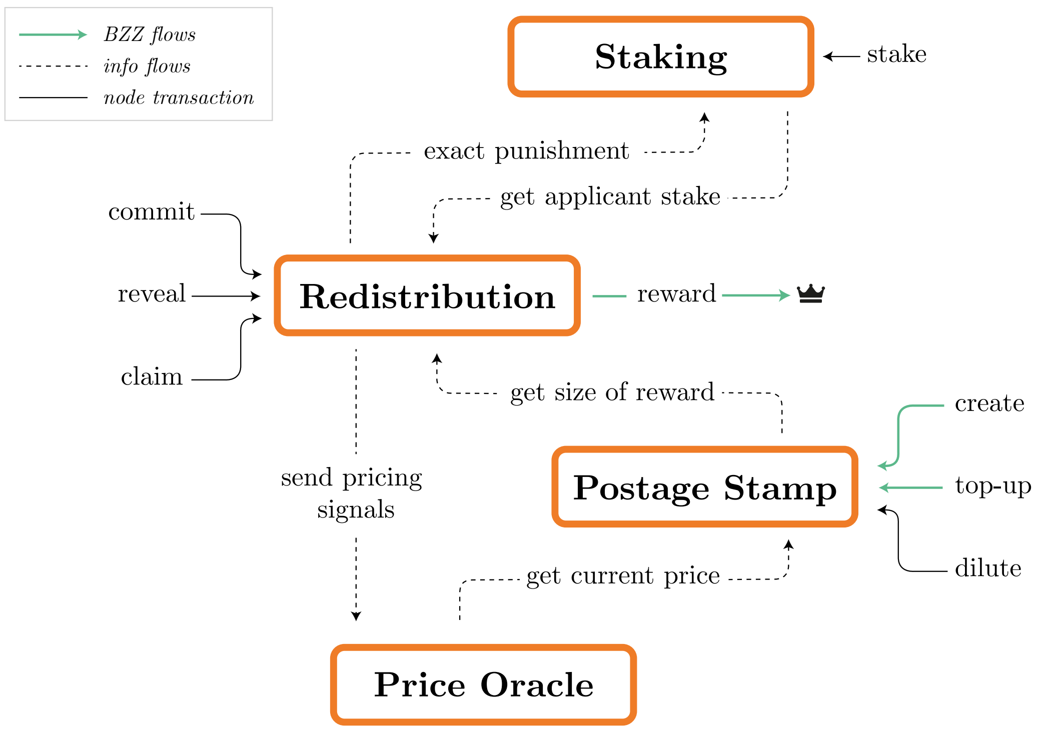

Figure 11: Interaction of smart contracts for swarm storage incentives. The figure shows with the dotted line the information flow between the four contracts comprising the storage incentive smart contract suite as well as the public transaction types they accept.

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

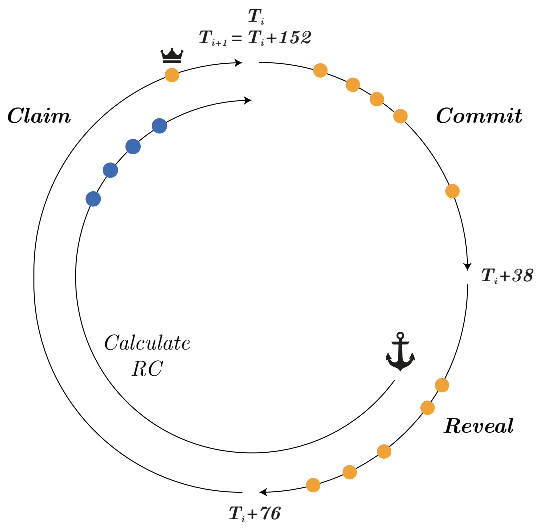

Figure 12: Phases of a round of the redistribution game. The figure displays the timeline of the repeating rounds of the redistribution game with its phases. In the context of smart contract interaction, logically starting with the commit phase, followed by reveal and claim. From the point of view of client node engagement starting with the end of the reveal phase with the neighbourhood selection anchor revealed, those in the selected neighbourhood start calculating their reserve sample only to submit it by the end of the next commit phase. If they are selected as an honest node and as a winner, they submit their proof of entitlement in a claim transaction.

Once the reveal phase is over, the *neighbourhood selection anchor* becomes known. Nodes that have the anchor within their neighbourhood[14] are allowed to participate in the following round (see figure 10).

The storer nodes in a neighbourhood are assumed to have consensus over the chunks that make up their reserve and provide evidence called *proof of entitlement* to the blockchain (discussed below in detail in 3.4). In such a game, the Nash-optimal strategy for each node is to follow the protocols and coordinate to guarantee that from the same information, all neighbouring peers will arrive at the same proof of entitlement. Since the proof of entitlement needs to be consensual but unstealable,[15] a commit/reveal scheme must be used.

In the commit phase, nodes in a neighbourhood will apply by submitting the *reserve commitment* obfuscated with an arbitrary key that they later reveal (see definition 30 in appendix A.2). The

[12]The commit and reveal phases are one quarter of the round length while the claim phase is one half.

[13]Both commit and reveal are simple and cheap transactions. The only expensive transaction is claim but that only the winner needs to submit.

[14]If storage depth is less than the anchor's proximity order relative to the overlay address.

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

smart contract receiving the commit transaction verifies that the node is staked, i.e., the registry of the staking contract contains an entry for the node's overlay with a stake that is higher than the minimum stake.

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

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

3.3 Staking

Neighbours with shared storage

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

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

CommiTted stake and stake balance

When registering in the staking contract, stakers commit to a stake denominated in rent units called *committed stake*. The committed stake must have a lower bound.[18] The amount sent with the transaction is recorded and serves as collateral called *stake balance*. Stakes can be created or topped up any time, but the update time is recorded together with the amount.

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

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

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

Participation is restricted to peers whose stake has not changed recently, thereby excluding the possibility of changing stakes after knowing the selected neighbourhood. Every time the stake of a node is queried, the contract returns the absolute committed stake in BZZ calculated as (1) the committed stake in rent units multiplied by the unit price of rent or (2) the entire stake balance, whichever is smaller (see definition 24 in appendix A.2).

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

Withdrawability of surplus stake balance

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

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

3.4 Neighbourhood consensus over the reserve

Peers applying for the round must agree on which chunks belong to their respective reserves. For this, at the very least, the applicants must consent on their area of responsibility, which can be derived from their storage depth and their overlay address. The consensus over the reserve content is tested with the identity of a *reserve sample*. The sample is the first k chunks in the reserve using an ordering based on a modified hash of the chunks (see 29 in appendix A.2). The modified chunk hash is obtained using the chunk contents and a *salt* specific to the round.[20] It is impossible for any node to construct this set unless they store all (or a substantial number of the) valid chunks together with their data at or after the time the salt is revealed.

Recency and sampling

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

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

[20]This modified hash is the BMT hash of the chunk data using Keccak-256 prefixed with the reserve sample salt as a base hash (defined in 28 in appendix A.2). The ordering is the ascending integer order reading the 32-byte modified hash as a big endian encoded 256-bit integer (see 22 in appendix A.2).

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

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

Storage depth and honest neighbourhood size

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

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

Overreporting storage depth is possible as long as the the node falls into this narrower proximity of the neighbourhood selection anchor. Therefore, a systematic exploit requires the malicious actor to control a staked node in each subneighbourhood of the true honest neighbourhood. On top of this, the winners also need to show evidence that the set of chunks within their storage depth do fill their reserve. The actual integer values of the transformed chunk addresses in the sample hold information regarding the size of the original sampled set. Requiring the size of the sampled set to fall within the expected range (with sufficient certainty) translates to imposing a constraint on the upper bound of the values of the sample. This construct is called *proof of density* (see definition 39 and appendix C).

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

Skipped rounds and rollover

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

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

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

The eight rules of entitlement

Here we summarise the eight rules of validating a claim (with committing and revealing a reserve commitment and then submitted evidence as proof of entitlement; see also table 1):[24]

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

REDUNDANCY
: The stake is used as weights in determining the within-neighbourhood probability of a node being selected as winner (see 36). This implies no benefit in submitting multiple claims. Operators running multiple nodes in one neighbourhood (sharing storage) therefore have no advantage over running a single node with the same total stake. Assuming this disincentive to proliferate is effective, staking can be regarded as guarantee for true redundancy.

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

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

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

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

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

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

[24]The first three criteria are part of the conditions for staking, committing, and revealing (see definitions 30 and 32 for replication, 24 and 25 for redundancy, 26 for responsibility in appendix A.2. Relevance, retention, recency and retrievability are proved as part of the proof of reserve (definition 37) and validated as part of the claim (38). Finally, proof of resources involves density proofs for chunks and stamps (definitions 30 and 32) validated as part of the claim (definitions 39 and 40, respectively). The specific construct of proof of entitlement is defined in 41 and its validation in 42 in appendix A.2).

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

Table 1: r8: proofs used as evidence for entitlement to reward.

3.5 Pricing and network dynamics

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

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

Splitting and merging of neighbourhoods

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

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

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

Utilisation rate is an organic way to introduce pressure against fully maxing out a node's reserve with critical content, and thereby enable early detection of capacity pressure. This provides sufficient safety buffer for the triggered incentives to take effect. For instance, if utilisation rate is 1/8, the storage depth is up to 3 PO-s shallower than the reserve depth.[25] Now the ingress can be really high and bring the reserve depth down to storage depth. When the tendency of

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

a closing gap between the potential (reserved) and actual (observed) utilisation of the DISC is detected, any incentive change will have the buffer to take effect without target redundancy being threatened.

Number of honest nodes as price signal

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

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

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

Parameterisation of the price oracle

The rule for updating the price from one round to the next is that the current price is multiplied by a value m which depends on the number of honest revealers in the round (formally defined in appendix A.3). Mathematically, $p_{t+1} = mp_t$, where p_t is the price in round t (and p_{t+1} is then the price in the following round). We define the multiplier m in terms of the number of honest revealers r and a stability parameter σ governing how quickly the price should increase or decrease, all other things equal.

In particular, we choose $m = 2^{\sigma(4-r)}$, and therefore we have $p_{t+1} = 2^{\sigma(4-r)}p_t$. This expresses how the deviation $4 - r$ of the number of revealers from the optimal value of 4 maps to an exponential change in price. The stability parameter σ determines the generic smoothness of price changes across rounds, i.e., how many rounds it takes for the price of rent to double in case of a consistent signal of the lowest degree of undersupply (or halve in case of a consistent oversupply). Figure 13 illustrates how the price model works.

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

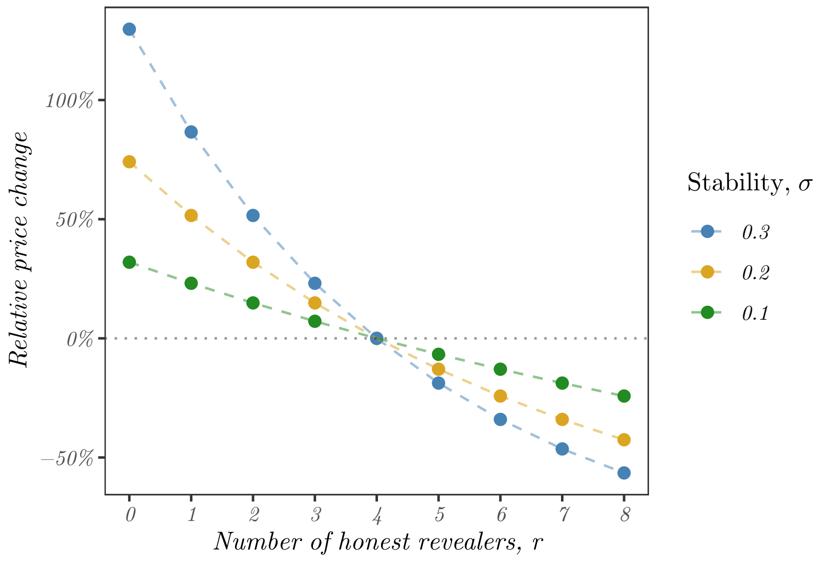

Figure 13: Adaptive pricing. The relative change in price (y-axis), mathematically expressed as the price in the next round divided by the price this round minus one ($p_{t+1}/p_t - 1$), is displayed against the number of honest revealers r in the current round (x-axis). This is done so for three different values of the stability parameter σ (colours). The points are the actual price change values; the connecting dashed lines are for visual aid only. The dotted horizontal line highlights the point at which no price change happens. Price change is exactly zero for any σ when the number of honest revealers is four. Otherwise, larger values of σ lead to larger relative price changes as the number of honest revealers is varied.

4 Conclusion

In this paper we presented and formalised a system of positive incentives for chunk storage in Swarm. This section summarises its features and analyse its benefits.[26]

Fair

The incentive system provides a fair distribution scheme, i.e., on average storer nodes receive compensation in proportion of their contribution to overall storage work performed by the swarm network in total. The amounts paid out depend on the time since the last payout and the total storage volume in the network. The rent deduced from batch balances is proportional to the time since the last payout and to storage volume purchased, thus making storage rent fair on uploaders.

Adaptive

The incentive system is adaptive to changing network size, data ingress rate and overall storage volume.

It achieves economic sustainability through self-regulating profitability by dynamically adapting the unit price of storage rent following signals of supply and demand.

[26]The areas left for future directions of research include: truely anonymous and leakless uploads with mixing postage stamps; guarantee period of availability and fixed price with *bzz future providers*; *non-compensatory content insurance* as a network service; parallel games; and last but not least finding an alternative to non-withdrawable stake for ensuring rendundancy.

Inexpensive

The system does not require any operator input, does not induce extra computational work, and incurs only minimal overhead in network traffic and storage. No significant upfront capital or advance registration is required to keep the barrier of entry low.

The reward is paid out upon submission of a valid claim directly verifiable by the smart contracts on the blockchain. The blockchain component does not require historical bookkeeping or managing registrations, and does not rely on challenges, or time-sensitive (frontrunnable) competition. Transaction cost is kept low so that the cost of required on-chain transaction do not impact the profitability of a storer node.

Robust

The system is conceptualised using requirements and corresponding evidence, which allows easier reasoning about resilience against exploits such as collusion, grieving/trolling, node and chunk proliferation attacks.

The system is resilient to swarm usage patterns - not sensitive to variance or other properties of ingress volumes per period, file sizes or postage batch size, number of batches, their balances and their owners.

Appendix

List of definitions and theorems

A Formal specification

A.1 DISC basics

Notation

- roman – predicates standing for blockchain verifiable properties
- SMALLCAPS– field of a composite, tuple member function
- Π^p – proof constructor function
- $\mathbb{V}^p$ – validator function
- `TT_FONT` – constant parameter (see appendix A.4)
- $uint[d]$ – left closed, right open integer range $\overline{[0, 2^d)}$

Sequences

Definition 1 – sequences .
Define $\tau\{n\}$, the non-polymorphic sequences of length n (non-negative) over any type τ as an indexing function:

$$\tau\{n\} \overset{\text{def}}{=} \begin{cases} \varnothing & \text{if } n = 0 \\ \overline{0, n-1} \mapsto \tau & \text{if } n > 0 \end{cases} \tag{1}$$

$$\tau+ \overset{\text{def}}{=} \bigcup_{n \in \mathbb{Z}^+} \tau\{n\} \tag{2}$$

$$\tau* \overset{\text{def}}{=} \{\varnothing\} \cup \tau+ \tag{3}$$

$$\tag{4}$$

Length function *len*:

$$len(s) \quad : \quad \tau \mapsto uint64 \tag{5}$$

$$len(s) \overset{\text{def}}{=} \begin{cases} 0 & \text{if } s = \varnothing \\ n & \text{if } s \in \tau\{n\} \end{cases} \tag{6}$$

The positional index 'at' function:

$$\texttt{[]} \quad : \quad \tau* \mapsto \times uint64 \mapsto \tau \tag{7}$$

$$s\,\texttt{[]} \quad : \quad \overline{0, len(s) - 1} \mapsto \tau \tag{8}$$

$$s\,\texttt{[}i\texttt{]} \overset{\text{def}}{=} s'(i) \tag{9}$$

Define the *concatenation* operator '$\oplus$':

$$(x \oplus y) \quad : \quad \overline{0, len(x) + len(y) - 1} \mapsto \tau \tag{10}$$

$$(x \oplus y)\,\texttt{[}i\texttt{]} \overset{\text{def}}{=} \begin{cases} x\,\texttt{[}i\texttt{]} & \text{if } 0 \leq i < len(x) \\ y\,\texttt{[}i - len(x)\texttt{]} & \text{otherwise} \end{cases} \tag{11}$$

Define slices (subsequences) with the range operator ':':

$$(s\,\texttt{[}o : o + l\texttt{]}) \quad : \quad \overline{0, l-1} \mapsto \tau \tag{12}$$

$$(s\,\texttt{[}o : o + l\texttt{]})\,\texttt{[}i\texttt{]} \overset{\text{def}}{=} s\,\texttt{[}o + i\texttt{]} \tag{13}$$

where

$$o, l \geq 0 \land \tag{14}$$
$$\mathit{len}(x) \geq o + l \tag{15}$$

Let us also define empty, prefix and suffix slices:

$$s\texttt{[}x\texttt{:}x\texttt{]} \overset{\text{def}}{=} \varnothing \tag{16}$$
$$s\texttt{[:}x\texttt{]} \overset{\text{def}}{=} s\texttt{[}0\texttt{:}x\texttt{]} \tag{17}$$
$$s\texttt{[}x\texttt{:]} \overset{\text{def}}{=} s\texttt{[}x\texttt{:}\mathit{len}(s)\texttt{]} \tag{18}$$

As a special case byte slices are defined as sequences of 8-bit integers.

Definition 2 – segmentation.
Define segment as an at most 32 long byte slice, and define segmentation of a slice of bytes as the partitioning of the slice into consecutive segments. Define segment count as the number of segments that cover a byte slice:

$$\mathit{SegCnt} : \mathit{byte}* \mapsto \mathit{uint64} \tag{19}$$
$$\mathit{SegCnt}(s) \overset{\text{def}}{=} \mathit{int}\left(\frac{\mathit{len}(s) - 1}{32}\right) + 1 \tag{20}$$

Now we can define the segment indexing function `[[]]` which maps the byte slice s and an index i to the i-the segment in the segmentation of s:

$$\texttt{[[]]} : \overline{\mathit{byte}* \mapsto \mathit{uint64}} \mapsto \mathit{Segment} \tag{21}$$
$$s\texttt{[[]]} : 0, \mathit{SegCnt}(s) - 1 \mapsto \mathit{Segment} \tag{22}$$
$$s\texttt{[[}i\texttt{]]} \overset{\text{def}}{=} \begin{cases} s\texttt{[}32 \cdot i\texttt{:]} & \text{if } i = \mathit{SegCnt} - 1 \\ s\texttt{[}32 \cdot i\texttt{:}32 \cdot (i+1)\texttt{]} & \text{otherwise} \end{cases} \tag{23}$$

Custom types

Definition 3 – Swarm overlay address of node n.
A Swarm node is associated a Ethereum account that the node operator must possess the private key for, called *bzz account* (K_n^{bzz}). The node's overlay address is derived as the hash of the binary serialisation of the Ethereum address of this account with the Swarm network ID and a minable nonce appended.

$$\mathit{overlay}(n) \overset{\text{def}}{=} H(\mathit{acc} \oplus \mathit{id} \oplus \nu) \tag{24}$$

where

$$\text{ETH ADDRESS } \mathit{acc} = \mathit{Account}(K_n) \tag{25}$$
$$\text{NETWORK ID } \mathit{id} = \texttt{BZZ\_NETWORK\_ID} \tag{26}$$
$$\text{OVERLAY NONCE } \nu \in \mathit{Nonce} \tag{27}$$

Definition 4 – DISC custom types .
Let us define the DISC specific custom types used in the formalisation.

$Segment \equiv byte\{32\} \equiv uint256$
  32-long slice of raw bytes
  in numerical context cast as BigEndian encoded 256-bit unsigned integer
$Address \equiv Segment$
  swarm chunk address, swarm peers' overlay address
$Nonce \equiv Segment$
  deterministically random *Segment*
$Account \equiv byte\{20\}$
  Ethereum address deriveed from EC keypair K
  $Account(K) \overset{\text{def}}{=} H(PubKey(K))\,[12:32]$
$Nodes \equiv Segment$
  swarm client node (peer)
$Sig \equiv byte\{65\}$
  $\langle r, s, v\rangle$ representation of an EC signature (32+32+1 bytes)
$Timestamp \equiv uint64$
  64-bit unsigned integer for unix time, nanosecond resolution
  big endian binary serialisation.
$H : byte* \mapsto Segment$
  the 256-bit Keccak SHA3 hash function, the base hash used in swarm.

XOR distance and proximity order

Definition 5 – XOR distance (χ).
Consider the set of bit sequences with fixed length d as points in a space. Define a distance metric χ such that the distance between two such sequences is the numerical value of their bitwise XOR ($\veebar$) using big endian (= most significant bit first) encoding.

$$\chi \quad : \quad uint[d] \times uint[d] \mapsto uint[d] \tag{28}$$

$$\chi(x, y) \quad \overset{\text{def}}{=} \quad Uint^d(BE^d(x) \veebar BE^d(x))) \tag{29}$$

Given the fixed length $d > 0$, there is a maximum distance ($2^d - 1 = \chi(0\{d\}, 1\{d\})$) in this space, and thus we can define the notion of *normalised distance*:

$$\overline{\chi} \quad : \quad uint[d] \times uint[d] \mapsto \mathbb{Q}[0, 1] \tag{30}$$

$$\overline{\chi}(x, y) \quad \overset{\text{def}}{=} \quad \frac{\chi(x, y)}{2^d - 1} \tag{31}$$

Definition 6 – Proximity order (PO).
Proximity order (PO) is a discrete logarithmic scaling of proximity.

$$PO \quad : \quad uint[d] \times uint[d] \mapsto \overline{0, d} \tag{32}$$

$$PO(x, y) \quad \overset{\text{def}}{=} \quad \begin{cases} d & \text{if } x = y \\ int(\log_2(Proximity(x, y))) & \text{otherwise} \end{cases} \tag{33}$$

where proximity is the inverse of normalised distance:

$$Proximity \quad : \quad uint[d] \times uint[d] \mapsto uint[d] \tag{34}$$

$$Proximity(x, y) \overset{\text{def}}{=} \frac{1}{\overline{\chi}(x, y)} \tag{35}$$

Given two points x and y, the order of their proximity $PO(x, y)$ equals the number of initial bits shared by their respective most significant bit first binary representations. In practice, with $d = 256$, $uint[d] \equiv Segment$, so PO also applies to a pair of slices of 32 bytes.

Binary Merkle tree hash

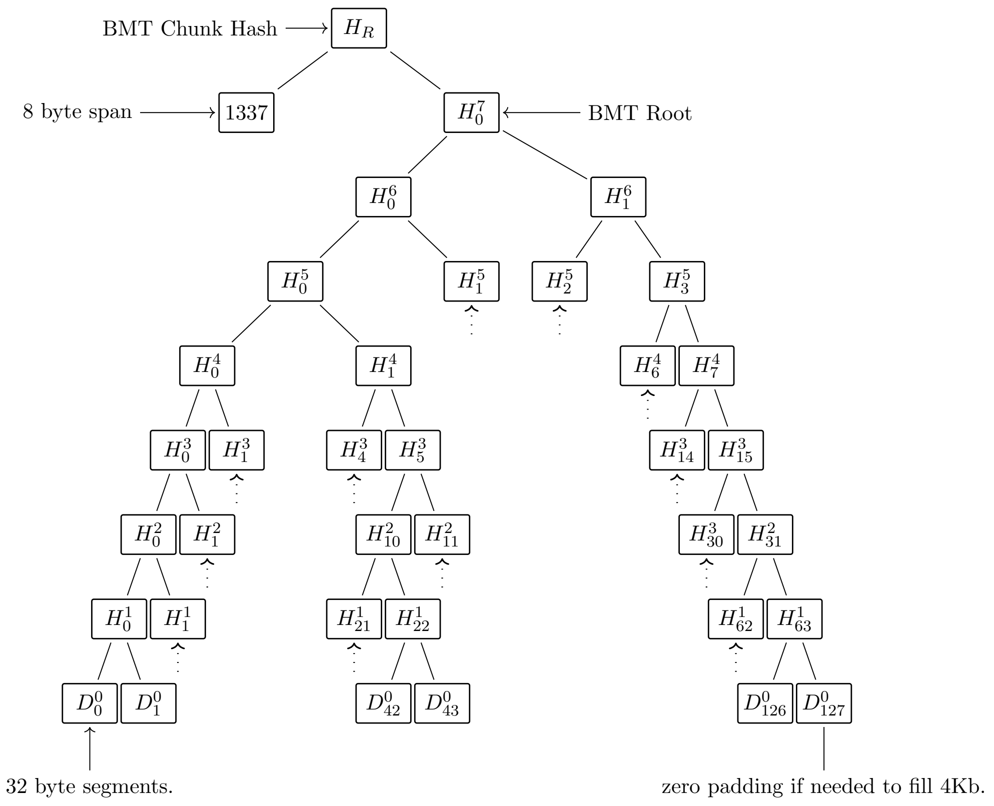

Figure 14: BMT (Binary Merkle Tree) used as the native chunk hash in Swarm. In this example, 1337 bytes of chunk data is segmented into 32 byte segments. Zero padding is used to fill up the rest up to 4 kilobytes. Pairs of segments are hashed together using the ethereum-native Keccak256 hash to build up the binary tree. On level 8, the binary Merkle root is prepended with the 8 byte span and hashed to yield the BMT chunk hash.

The BMT chunk address is the hash of the 8 byte metadata (span) and the root hash of a *binary Merkle tree* (*BMT*) built on the 32-byte segments of the underlying data (see figure 14). If the chunk content is less than 4k, the hash is calculated as if the chunk was padded with all zeros up to 4096 bytes.

Definition 7 – Binary Merkle Tree Root .
Define $\triangle[H, n](i, d)$ as the Binary Merkle Tree Root of data d fitting in at most 2^n 32-byte

segments using H as its base hash:

$$\triangle \quad : \quad (\mathit{byte}{*} \mapsto \mathit{Segment}) \times \mathit{uint8} \mapsto \mathit{uint8} \times \mathit{byte}{*} \mapsto \mathit{Segment} \tag{36}$$

$$\triangle[H,n] = \quad : \quad \overline{0,n} \times \mathit{byte}\{,2^n\} \mapsto \mathit{Segment} \tag{37}$$

$$\triangle[H,n](i,d) \overset{\text{def}}{=} \begin{cases} d & \text{if } i = 0 \\ \triangle[H,n](i, d \oplus 0\{2^{n+5} - \mathit{len}(d)\}) & \text{if } \mathit{len}(d) < 2^{i+5} \\ H(\triangle[H,n](d\texttt{[}:2^{i-1}\texttt{]}, i-1) \oplus \\ \quad \oplus \triangle[H,n](d\texttt{[}2^{i-1}:\texttt{]}, i-1)) & \text{otherwise} \end{cases} \tag{38}$$

Definition 8 – Binary Merkle Tree Hash .
Define $BMT[H](d,m)$ as the hash of Binary Merkle Tree Root of chunk length data d prepended with metadata m. H is the base hash function, by default 256-bit Keccak. Note that a chunk size blob of bytes can accommodate 128 32-byte segments, hence depth 7:

$$BMT \quad : \quad (\mathit{byte}{*} \mapsto \mathit{Segment}) \mapsto \mathit{Chunk} \times \mathit{byte}\{8\} \mapsto \mathit{Segment} \tag{39}$$

$$BMT[H] \quad : \quad \mathit{Chunk} \times \mathit{byte}\{8\} \mapsto \mathit{Segment} \tag{40}$$

$$BMT[H](d,m) \overset{\text{def}}{=} H(m \oplus \triangle[H,7](7,d)) \tag{41}$$

Chunks

Definition 9 – Content addressed chunks .
Define content address chunk c as a function from bytes data with size limit of 4096 bytes and an associated address calculated with BMT:

$$CAC \quad : \quad \mathit{Chunk} \times \mathit{byte}\{8\} \mapsto \mathit{Chunks} \tag{42}$$

$$CAC(d,m) \overset{\text{def}}{=} \langle \mathit{addr}, \mathit{cont} \rangle \tag{43}$$

such that

$$\mathit{addr} \overset{\text{def}}{=} BMT(d,m) \tag{44}$$

$$\mathit{cont} \overset{\text{def}}{=} m \oplus d \tag{45}$$

By convention the metadata prefix m encodes the *span* using 64-bit little endian. If the chunk is an intermediate chunk (see definition 11), the span is the length of the data that the subtree spans over. If the chunk is a data chunk, then span encodes the data length:

$$m \overset{\text{def}}{=} LE^{64}(\mathit{len}(d)) \tag{46}$$

We also say that for $\mathit{cac} = CAC(d,m) = \langle \mathit{addr}, \mathit{cont} \rangle$:

$$\textsc{address}(\mathit{cac}) \overset{\text{def}}{=} \mathit{addr} \tag{47}$$

$$\textsc{payload}(\mathit{cac}) \overset{\text{def}}{=} \mathit{cont} \tag{48}$$

$$\textsc{data}(\mathit{cac}) \overset{\text{def}}{=} d \tag{49}$$

$$\textsc{metadata}(\mathit{cac}) \overset{\text{def}}{=} m \tag{50}$$

For convenience *SegCnt* and the segment indexing function '`[[]]`' can be trivially extended to apply to chunks:

$$\begin{aligned} \mathit{SegCnt} &: \mathit{Chunks} \mapsto \mathit{uint64} && (51)\\ \mathit{SegCnt}(c) &\stackrel{\text{def}}{=} \mathit{SegCnt}(\textsc{data}(c)) && (52) \end{aligned}$$

$$\begin{aligned} \texttt{[[]]} &: \mathit{Chunks} \times \mathit{uint64} \mapsto \mathit{Segment} && (53)\\ c\texttt{[[}i\texttt{]]} &\stackrel{\text{def}}{=} \textsc{data}(c)\texttt{[[}i\texttt{]]} && (54) \end{aligned}$$

Single Owner Chunks

Definition 10 – Single owner chunks .
A single owner chunk is defined as a content addressed chunk associated with an ID and an ethereum address:

$$\begin{aligned} \mathit{SOC} &\stackrel{\text{def}}{=} \mathit{Account} \times \mathit{Segment} \times \mathit{CAC} \mapsto \mathit{Chunks} && (55)\\ \mathit{SOC}(\mathit{owner}, \mathit{id}, \mathit{cac}) &\stackrel{\text{def}}{=} \langle \mathit{addr}, \mathit{cont} \rangle && (56) \end{aligned}$$

where

$$\begin{aligned} \mathit{addr} &\stackrel{\text{def}}{=} H(o \oplus \mathit{id}) && (57)\\ \mathit{cont} &\stackrel{\text{def}}{=} \mathit{id} \oplus \mathit{Sig}(o, \mathit{id} \oplus \textsc{address}(\mathit{cac})) \oplus \textsc{payload}(\mathit{cac}) && (58) \end{aligned}$$

A single owner chunk's address is the Keccak256 hash of identifier prepended to owner account, while its data is serialised as follows:

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

Integrity of a *single owner chunk* is verified with the following process:

1. Deserialise the chunk content into fields for identifier, signature and payload.
2. Construct the expected plaintext composed of the identifier and the *BMT hash* of the payload.
3. Recover the owner's address from the signature using the plaintext.
4. Check the hash of the identifier and the owner (expected address) against the chunk address.

Definition 11 – Packed address chunk.
Define the packed address chunk for a sequence of chunks C as the concatenation of all the

addresses of the chunks in the sequence:

$$\begin{aligned} PAC &: Chunks* \times byte\{8\} \mapsto Chunks && (59) \\ PAC(C,m) &\overset{\text{def}}{=} CAC\left(\bigoplus_{i=0}^{len(C)-1} Address(C[i]), m\right) && (60) \end{aligned}$$

Segment inclusion proofs

Using BMT hashes allows for compact *segment inclusion proofs* (substring relationship with a 32-byte resolution).

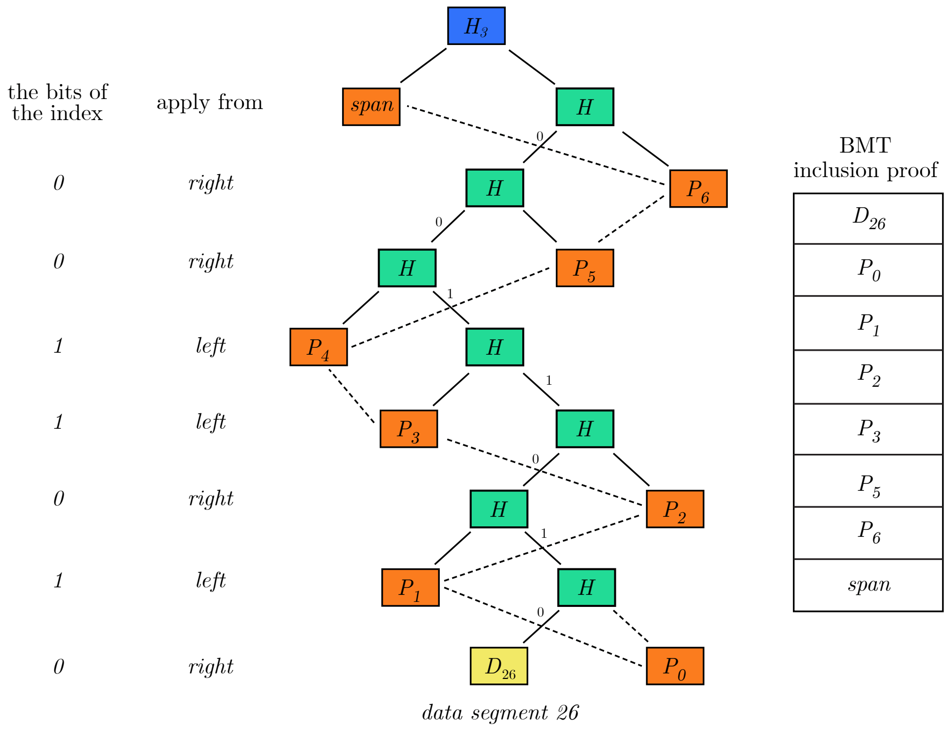

Figure 15: Compact segment inclusion proofs for chunks. Assume we need proof for segment 26 of a chunk (yellow). The orange hashes of the BMT are the sister nodes on the path from the data segment up to the root and constitute what needs to be part of a proof. When these are provided together with the root hash and the segment index, the proof can be verified. The side on which proof item i needs to be applied depends on the i-th bit (starting from least significant) of the binary representation of the index. Finally the span is prepended and the resulting hash should match the chunk root hash.

Definition 12 – BMT segment inclusion proof .
Define $\Pi^{\text{SIP}}(c,i)$ as the *BMT* inclusion proof on chunk c for segment index i:

$$\begin{aligned} \Pi^{\text{SIP}} &: Chunks \times \overline{0,127} \mapsto SIP && (61) \\ SIP &\overset{\text{def}}{=} Segment \times Segment^7 \times byte\{8\} && (62) \\ \Pi^{\text{SIP}}(c,i) &\overset{\text{def}}{=} \langle c[[i]], \langle h_0, h_1, \ldots, h_6\rangle, \text{METADATA}(c)\rangle && (63) \end{aligned}$$

where

$$\begin{aligned} h_j &\overset{\text{def}}{=} BMT(s_j, j) && (64) \\ s_j &\overset{\text{def}}{=} c\texttt{[}start(i,j) : start(i,j) + 32 \cdot 2^j\texttt{]} && (65) \end{aligned}$$

where

$$start(i,j) \overset{\text{def}}{=} \begin{cases} 0 & \text{if } j = 7 \\ start(i, j+1) & \text{if } int\left(i/2^j\right) = 0 \mod 2 \\ start(i, j+1) + 32 \cdot 2^j & \text{otherwise} \end{cases} \quad (66)$$

In order to validate segment inclusion proofs we first introduce the prover hash function H_Π.

Definition 13 – BMT prover function.

$$\begin{aligned} H_\Pi &: \overline{0,127} \times SIP \mapsto Address && (67) \\ H_\Pi(i, \langle d, sisters, m\rangle) &\overset{\text{def}}{=} H(m, H_\Pi^\triangle(7, d, sisters)) && (68) \end{aligned}$$

where

$$H_\Pi^\triangle : \overline{0,7} \times Segment \times Segment^7 \mapsto Address \quad (69)$$

$$H_\Pi^\triangle(j,d,s) \overset{\text{def}}{=} \begin{cases} d & \text{if } j = 0 \\ H(H_\Pi^\triangle(j-1,d,s) \oplus s\texttt{[}j-1\texttt{]}) & \text{if } int(i/2^{j-1}) = 0 \mod 2 \\ H(s\texttt{[}j-1\texttt{]} \oplus H_\Pi^\triangle(j-1,d,s)) & \text{otherwise} \end{cases} \quad (70)$$

Definition 14 – BMT SIP validation .
Define $\mathbb{V}^{\text{SIP}}(a,i,p)$ as the validator of a BMT segment inclusion proof p for chunk at address a on segment index i:

$$\begin{aligned} \mathbb{V}^{\text{SIP}} &: Segment \times \overline{0,127} \times SIP \mapsto \{\texttt{T}, \texttt{F}\} && (71) \\ \mathbb{V}^{\text{SIP}}(a,i,p) &\Leftrightarrow H_\Pi(i,p) = a && (72) \end{aligned}$$

Definition 15 – Single owner chunks data integrity proof .
Define a single owner chunk storage proof $\Pi^{\text{SOC}}(c,i)$ as a segment inclusion proof of the data payload of SOC c on index i together with the ID and signature of SOC c:

$$\begin{aligned} SIP_{SOC} &\overset{\text{def}}{=} SIP \times Sig \times Segment && (73) \\ \Pi^{\text{SIP[SOC]}} &: SOC \times \overline{0,127} \mapsto SIP_{SOC} && (74) \\ \Pi^{\text{SIP[SOC]}}(\langle o, id, cac\rangle, i) &\overset{\text{def}}{=} \langle p, sig, id\rangle && (75) \end{aligned}$$

where

$$\begin{aligned} p &= \Pi^{\text{SIP}}(cac, i) && (76) \\ sig &= Sig(o, id \oplus address(cac)) && (77) \end{aligned}$$

Definition 16 – Single owner chunks data integrity validation .
Define $\mathbb{V}^{\text{SIP[SOC]}}(a, i, p)$ as the validator of a single owner chunk storage proof p for chunk at address a on segment index i:

$$\begin{aligned}\mathbb{V}^{\text{SIP[SOC]}} &: \mathit{Address} \times \overline{0,127} \times \mathit{SIP}_{SOC} \mapsto \{\mathtt{T},\mathtt{F}\} && (78)\\ \mathbb{V}^{\text{SIP[SOC]}}(a, i, \langle p, sig, id\rangle) &\Leftrightarrow a = H(id \oplus o) && (79)\end{aligned}$$

such that

$$\begin{aligned}\text{OWNER } o &= \mathit{ECRecover}(sig, id \oplus a') && (80)\\ \text{PAYLOAD } a' &= H_\Pi(i, p) && (81)\end{aligned}$$

Postage stamps

Definition 17 – Postage stamps .

$$\begin{aligned}\mathit{Stamps} &\stackrel{\text{def}}{=} \mathit{Segment} \times \mathit{uint64} \times \mathit{Timestamp} \times \mathit{Address} && (82)\\ ps &= \langle b, i, ts, a\rangle \in \mathit{Stamps} && (83)\\ \text{BATCHID}(ps) &\stackrel{\text{def}}{=} b && (84)\\ \text{INDEX}(ps) &\stackrel{\text{def}}{=} i && (85)\\ \text{TIMESTAMP}(ps) &\stackrel{\text{def}}{=} ts && (86)\\ \text{ADDRESS}(ps) &\stackrel{\text{def}}{=} a && (87)\end{aligned}$$

Definition 18 – Storage slot reference .
Define the *storage slot reference* $slot(ps)$ of a postage stamp ps as the tuple of the batch identifier and the within-batch stamp counter:

$$\begin{aligned}\mathit{Slots} &\stackrel{\text{def}}{=} \mathit{Segment} \times \mathit{uint64} && (88)\\ slot &: \mathit{Stamps} \mapsto \mathit{Slots} && (89)\\ slot(ps) &\stackrel{\text{def}}{=} \langle \text{BATCHID}(ps), \text{INDEX}(ps)\rangle && (90)\end{aligned}$$

Definition 19 – Postage stamp validity .
Define $\mathbb{V}^{\text{STAMP}}(ps)$ as the validator of the proof of relevance expressed as the postage stamp ps relying on blockchain information:

$$\begin{aligned}\mathbb{V}^{\text{STAMP}} &: \mathit{Stamps} \times \Gamma \times \mathit{Nodes} \mapsto \{\mathtt{T},\mathtt{F}\} && (91)\\ \mathbb{V}^{\text{STAMP}}(ps, \gamma, n) &\Leftrightarrow && (92)\\ \text{AUTHENTIC} &\quad \text{BATCHID}(ps) \in \text{Batches}(\gamma) \wedge && (93)\\ \text{ALIVE} &\quad \text{Balance}(ps) > 0 \wedge && (94)\\ \text{AUTHORISED} &\quad \mathit{ECRecover}(\mathit{Sig}(ps), \mathit{encode}(ps)) = \text{Owner}(ps) \wedge && (95)\\ \text{AVAILABLE} &\quad 0 <= \text{INDEX}(ps) < \text{Size}(ps) \wedge && (96)\\ \text{ALIGNED} &\quad \mathit{PO}(\text{INDEX}(ps), n) \geq \text{DEPTH}(\mathit{reveal}(\gamma, n)) && (97)\end{aligned}$$

Ordering and sampling

Lemma 20 – Ordering and indexing functions.
Given an arbitrary finite set C, and another set I with a total order $<$. Any invertible function total over C, $f : C \mapsto I$ defines a total order $<_f$ over C as follows:

$$<_f \quad \subseteq \quad C \times C \tag{98}$$

$$\forall c, c' \in C, c <_f c' \quad \Leftrightarrow \quad f(c) < f(c') \tag{99}$$

Proof. f is injective wrt I, so $\mathit{Image}(f) = J \subseteq I$. Since $<$ restricted to a subset $(<_J)$ is also a total order over J. Since f is invertible, C and J are isomorphic and therefore the total order $<$ on J carries over to C.

Corollary 21 – hash orders.
Any hash function defines a total order on a finite set of byte slices.

Proof. The collision free nature of the hash function makes it practically invertible. The actual hashes when read as binary encodings of integers, offer a natural integer ordering over the values.

Example: the prefixed BMT hash transform (see 28) defines a total order over a set of chunks.

Definition 22 – Sampler function.
Using f and its derivative ordering on $C \subseteq \mathit{Dom}(f)$ we represent C as an ordered sequence:

$$\overrightarrow{\mathit{Seq}} \quad : \quad (T \mapsto T) \times \mathcal{P}(T) \mapsto T* \tag{100}$$

$$\overrightarrow{\mathit{Seq}}(f, C) \quad \stackrel{\text{def}}{=} \quad C' \tag{101}$$

$$\text{such that } f(C'\texttt{[}i\texttt{]}) < f(C'\texttt{[}j\texttt{]}) \qquad \text{for every } 0 \leq i < j < |C| \tag{102}$$

Finally, we define a sampler function for any f invertible with an image having a total order and $C \subseteq \mathit{Dom}(f)$ such that it selects a prefix slice of length l from the ordered C:

$$\mathit{Sampler} \quad : \quad (T \mapsto T) \times \mathcal{P}(T) \times \mathit{uint64} \mapsto T+ \tag{103}$$

$$\mathit{Sampler}(f, C, l) \quad \stackrel{\text{def}}{=} \quad \overrightarrow{\mathit{Seq}}(f, C)\texttt{[:}l\texttt{]} \tag{104}$$

A.2 Redistribution game

Definition 23 – Redistribution game round.
Define γ as a redistribution game round. γ is conceived of as a multidimensional index:

$$\begin{aligned}
\Gamma &\overset{\text{def}}{=} \mathit{uint64} \times \mathit{uint64} \times \mathit{uint64} && (105)\\
\gamma \in \Gamma &= \langle c, \sigma, i \rangle && (106)\\
\textsc{chain}(\gamma) \quad & c \quad \text{ID of the blockchain context} && (107)\\
\textsc{series}(\gamma) \quad & \sigma \quad \text{index of the parallel series} && (108)\\
\textsc{round}(\gamma) \quad & i \quad \text{sequential index of the round} && (109)
\end{aligned}$$

Define $\textsc{block}(\gamma)$ as the starting block height of this particular game γ:

$$\textsc{block}(\gamma) \overset{\text{def}}{=} \textsc{round}(\gamma) \cdot \texttt{ROUND\_LENGTH} + \texttt{START\_BLOCK} \tag{110}$$

Ordering by sequential index defines the chain of games, which lets us define the *Prev* function:

$$\begin{aligned}
\mathit{Prev} &: \Gamma \mapsto \Gamma && (111)\\
\mathit{Prev}(\langle c, \sigma, i \rangle) &\overset{\text{def}}{=} \langle c, \sigma, i-1 \rangle && (112)
\end{aligned}$$

Transactions and on-chain registers

The smart contract receives transactions from applicants in phases. The following virtual registers capture the information given in these transactions that are relevant for defining the winner:

– batches (see definition 19)
– stakes (see definition 24)
– commits (see definition 25)
– reveals (see definition 26)

Definition 24 – Stakes .
We define *Stakes* as the registry of stakes resulting from transactions sent to the staking contract. A record is a tuple of a node overlay, the stake balance and the committed stake and can be updated.

$$\begin{aligned}
\mathit{Stakes} &: \Gamma \mapsto \mathit{Nodes} \times \mathit{uint64} \times \mathit{uint64} && (113)\\
\langle n, s, m \rangle \in \mathit{Stakes}(\gamma) &\Leftrightarrow && (114)\\
\textsc{right age} \quad & \exists b' < b - \texttt{MIN\_STAKE\_AGE}, \tau \in \text{Transactions}(b') && (115)\\
\textsc{node overlay} \quad & n = H(\text{origin}(\tau) \oplus \texttt{BZZ\_NETWORK\_ID} \oplus \text{data}(\tau)\texttt{[0]}) && (116)\\
\textsc{stake balance} \quad & s = \text{amount}(\tau) && (117)\\
\textsc{committed stake} \quad & m = \text{data}(\tau)\texttt{[1]} && (118)
\end{aligned}$$

There is only one stake allowed per node, so we can define the staked amount belonging to a node as the minumum of the stake balance and the committed stake times the unit price of storage:

$$\begin{aligned} \mathit{Stake} &: \Gamma \times \mathit{Nodes} \mapsto \mathit{uint64} && (119)\\ \mathit{Stake}(\gamma, n) &\overset{\text{def}}{=} \mathit{min}(s, m \cdot \mathit{Price}(\gamma)) && (120) \end{aligned}$$

Definition 25 – Commits .
We define *Commits*(γ) as the registry of applications for a game γ resulting from a transaction sent to the game contract's *commit* endpoint. A tuple of the overlay of the committing node, its commitment hash and the number of the block containing the transaction is entered in the register after verifying that (i) the transaction was sent during the commit phase (right time), and (ii) that the node has enough stake and is not frozen (right amount).

$$\begin{aligned} \mathit{Commits} &: \Gamma \mapsto \mathit{Nodes} \times \mathit{Segment} \times \mathit{Blocks} && (121)\\ \langle n, h, b\rangle \in \mathit{Commits}(\gamma) &\Leftrightarrow && (122)\\ \textsc{right time} &\quad b < \texttt{PHASE\_LENGTH} \mod \texttt{ROUND\_LENGTH} && (123)\\ \textsc{right amount} &\quad \mathit{stake}(\gamma, n) \geq \texttt{MINIMUM\_STAKE} && (124)\\ (\textsc{redundancy}) & && (125) \end{aligned}$$

Definition 26 – Reveals.
We define *Reveals* as the registry of reveals resulting from a transaction sent to the game contract's reveal endpoint. The reveal record is a tuple of the node overlay, the two commitment hashes, the self-reported storage depth, a serial index used for sorting, the obfuscation key, and the block number. The record is entered in the register after it is validated that (i) it was submitted during the reveal phase (right time), (ii) the commitments when obfuscated match the commit by the same node (right reveal), and (iii) that the neighbourhood selection anchor

falls within the node's area of responsibility using the self-reported depth (right location).

$$
\begin{aligned}
\mathit{RevealEntry} &: \mathit{Nodes} \times \mathit{Address}^2 \times \mathit{uint8}^2 \times \mathit{Nonce} \times \mathit{Blocks} && (126)\\
\mathit{Reveals} &: \Gamma \mapsto \mathcal{P}(\mathit{RevealEntry}) && (127)\\
r = \langle n, chc, chs, sd, i, k, b\rangle \in \mathit{Reveals}(\gamma) &\Leftrightarrow \langle && (128)\\
\textsc{node}(r) &= n && (129)\\
\textsc{chc}(r) &= chc && (130)\\
\textsc{chs}(r) &= chs && (131)\\
\textsc{depth}(r) &= sd && (132)\\
\textsc{index}(r) &= i && (133)\\
\textsc{nonce}(r) &= k && (134)\\
\textsc{block}(r) &= b && (135)\\
& \rangle && (136)\\
&\Leftrightarrow && (137)\\
\textsc{right time} \quad & p \leq b < 2p \mod r && (138)\\
& p = \texttt{PHASE\_LENGTH}, r = \texttt{ROUND\_LENGTH} && (139)\\
\textsc{right reveal} \quad & H(n \oplus sd \oplus chc \oplus chs \oplus k) = h \text{ such that} && (140)\\
& \langle n, h, b'\rangle \in \mathit{Commits}(\gamma) \text{ for some } b' && (141)\\
\textsc{right location} \quad & \mathit{PO}(\mathit{addr}(n), \mathit{NSA}(\mathit{Prev}(\gamma))) \geq sd && (142)\\
(\textsc{responsibility}) \quad & && (143)
\end{aligned}
$$

There is only one reveal allowed per node, so we can define the reveal belonging to a node:

$$
\begin{aligned}
\mathit{Reveal} &: \Gamma \times \mathit{Nodes} \mapsto \mathit{RevealEntry} && (144)\\
\mathit{Reveal}(\gamma, n) &= r && (145)
\end{aligned}
$$

such that

$$
\begin{aligned}
n &= \mathit{Node}(r) && (146)\\
r &\in \mathit{Reveals}(\gamma) && (147)
\end{aligned}
$$

Random nonces

Definition 27 – Random nonces for the round.
From the round's random seed (see definition 45 appendix B) we can derive all the necessary random input nonces:

$$
\begin{aligned}
\text{N.hood Selection Anchor } \mathit{NSA}(\gamma) &\stackrel{\text{def}}{=} H(\mathcal{R}(\gamma) \oplus \mathit{BE}^{64}(\mathit{SK}(\gamma))) && (148)\\
\text{Truth Selection Nonces } \mathit{TSN}(\gamma) &\stackrel{\text{def}}{=} H(\mathcal{R}(\gamma) \oplus \mathit{BE}^{8}(0)) && (149)\\
\text{Winner Selection Nonces } \mathit{WSN}(\gamma, i) &\stackrel{\text{def}}{=} H(\mathcal{R}(\gamma) \oplus \mathit{BE}^{8}(1)) && (150)\\
\text{Reserve sampling salt } \mathit{RSS}(\gamma) &\stackrel{\text{def}}{=} H(\mathcal{R}(\gamma)) && (151)\\
\text{Segment Selection Nonce } \mathit{SSN}(\gamma, i) &\stackrel{\text{def}}{=} H(\mathcal{R}(\gamma) \oplus \mathit{BE}^{8}(i)) && (152)
\end{aligned}
$$

where

$$
\begin{aligned}
\mathit{SK} &: \Gamma \mapsto \mathit{uint64} && (153)\\
\mathit{SK}(\gamma) &\stackrel{\text{def}}{=} \begin{cases} 0 & \text{if } |\mathit{Reveals}(\gamma)| > 0 \\ \mathit{SK}(\mathit{Prev}(\gamma)) + 1 & \text{otherwise} \end{cases} && (154)
\end{aligned}
$$

Let us now define the witness selection function $\mathcal{W}$ that selects two random witness indexes as well as the last index of the reserve sample such that they are all distinct:

$$
\begin{aligned}
\mathcal{W} &: \Gamma \times \mathit{uint64} \times \{0,1,2\} \mapsto \mathit{uint8} && (155)\\
\mathcal{W}(\gamma, m, k) &\stackrel{\text{def}}{=} \begin{cases} \mathit{SSN}(\gamma, 0) \mod m-1 & \text{if } k = 0 \\ m-1 & \text{if } k = 2 \\ m-2 & \text{if } k = 1 \wedge \\ & \quad \mathit{SSN}(\gamma,0) = \mathit{SSN}(\gamma,1) \mod m-1 \\ \mathit{SSN}(\gamma, 1) \mod m-2 & \text{otherwise} \end{cases} && (156)
\end{aligned}
$$

Winner selection and claim validation

Definition 28 – Prefixed hash .
Define the hash prefixing function $\mathit{prefix}(H, p)$ as a function which when applied to a hash function H and a constant byte slice p outputs a hash function which for every input returns the hash of the input prefixed by p using H.

$$
\begin{aligned}
\mathit{prefix} &: (\mathit{byte}* \mapsto \mathit{Segment}) \times \mathit{byte}* \mapsto (\mathit{byte}* \mapsto \mathit{Segment}) && (157)\\
\mathit{prefix}(H,p) &: \mathit{byte}* \mapsto \mathit{Segment} && (158)\\
\mathit{prefix}(H,p)(b) &\stackrel{\text{def}}{=} H(p \oplus b) && (159)
\end{aligned}
$$

Exceptionally, we define the prefixed version of BMT hash (denoted as $\mathit{BMT}[]$) as one that uses the prefixed verson of its base hash:

$$
\begin{aligned}
\mathit{BMT}[] &: \mathit{byte}* \mapsto \mathit{Chunk} \times \mathit{byte}\{8\} \mapsto \mathit{Segment} && (160)\\
\mathit{BMT}[p] &\stackrel{\text{def}}{=} \mathit{BMT}[\mathit{prefix}(H,p)] && (161)
\end{aligned}
$$

Definition 29 – Transformed chunk reserve sample.
Let us now define the chunk transformation function $\Delta^C(p)$ for a random nonce prefix p as follows:

$$
\begin{aligned}
\Delta^C &: \mathit{Nonce} \mapsto \mathit{Chunks} \mapsto \mathit{Segment} && (162)\\
\Delta^C(p) &: \mathit{Chunks} \mapsto \mathit{Segment} && (163)\\
\Delta^C(p)(c) &\stackrel{\text{def}}{=} \mathit{BMT}[p](\textsc{data}(c), \textsc{metadata}(c)) && (164)
\end{aligned}
$$

Define the transformed reserve sample $RS^C(\gamma, n)$ for game γ and node n as the first 2^d chunks of the node's reserve at block $\textsc{block}(\gamma)$, using the ordering defined (see lemma 20 and definition 22) by the hash (see definition 28) of their data using BMT with 256-bit Keccak prefixed with random nonce p as its base hash.

$$\begin{aligned} RS^C &: \Gamma \times \mathit{Nodes} \mapsto \mathit{Chunks}* && (165)\\ RS^C(\gamma, n) &\overset{\text{def}}{=} \mathit{Sampler}(\Delta^C(p), \mathit{Reserve}(\gamma, n), 2^d) && (166)\end{aligned}$$

where

$$\begin{aligned} d &= \texttt{SAMPLE\_DEPTH} \text{ (see A.4)} && (167)\\ p &= \mathit{RSS}(\gamma) \text{ (see definition 27)} && (168)\end{aligned}$$

Definition 30 – Chunk reserve sample commitment hash .
Define $CH^C(\gamma, n,)$ for game γ and node n as the BMT chunk hash of the packed address chunk (see definition 11) packing the chunks of the transformed reserve sample (see definition 29) for game γ and node n.

$$\begin{aligned} CH^C &: \Gamma \times \mathit{Nodes} \mapsto \mathit{Address} && (169)\\ CH^C(\gamma, n) &\overset{\text{def}}{=} \mathit{Address}(PAC^C(RS^C(\gamma, n)), p) && (170)\end{aligned}$$

where

$$p = \mathit{RSS}(\gamma) \text{ (see definition 27)} \qquad (171)$$

and where

$$\begin{aligned} PAC^C &: \mathit{Chunks}* \times \mathit{Nonce} \mapsto \mathit{Chunks} && (172)\\ PAC^C(C, p) &\overset{\text{def}}{=} CAC\left(\bigoplus_{i=0}^{len(C)-1} \mathit{Seg}(C\texttt{[}i\texttt{]}, p), m\right) && (173)\end{aligned}$$

where

$$\begin{aligned} \mathit{Seg}(C\texttt{[}i\texttt{]}, p) &= \mathit{Address}(C\texttt{[}i\texttt{]}) \oplus \Delta^C(p)(C\texttt{[}i\texttt{]}) && (174)\\ m &= LE^{64}(2 \cdot 32 \cdot len(C)) && (175)\end{aligned}$$

Definition 31 – Transformed slots reserve sample.
Let us now define the slots transformation function $\Delta^S(p)$ for a random nonce p as follows:

$$\begin{aligned} \Delta^S &: \mathit{Nonce} \mapsto \mathit{Chunks} \mapsto \mathit{Segment} && (176)\\ \Delta^S(p) &: \mathit{Chunks} \mapsto \mathit{Segment} && (177)\\ \Delta^S(p)(c) &\overset{\text{def}}{=} H[p](\mathit{Slot}(\mathit{Stamp}(c))) && (178)\end{aligned}$$

Define the transformed reserve sample $RS^S(\gamma, n)$ for game γ and node n as the first 2^d chunks of the node's reserve at block $\textsc{block}(\gamma)$, using the ordering defined by the slot transformation function using prefix p:

$$\begin{aligned} RS^S &: \Gamma \times \mathit{Nodes} \mapsto \mathit{Chunks}* && (179)\\ RS^S(\gamma, n) &\overset{\text{def}}{=} \mathit{Sampler}(\Delta^S(p), \mathit{Reserve}(\gamma, n), 2^d) && (180)\end{aligned}$$

where

$$
\begin{aligned}
d &= \texttt{SAMPLE\_DEPTH} \text{ (see A.4)} && (181)\\
p &= \mathit{RSS}(\gamma) \text{ (see definition 27)} && (182)
\end{aligned}
$$

Definition 32 – Transformed slots reserve sample commitment hash .
Define $\mathit{CH}^S(\gamma, n)$ for game γ and node n as the BMT chunk hash of the packed address chunk (see definition 11) the chunks of the transformed reserve sample (see definition 29) for game γ and node n.

$$
\begin{aligned}
\mathit{CH}^S &: \Gamma \times \mathit{Nodes} \mapsto \mathit{Address} && (183)\\
\mathit{CH}^S(\gamma, n) &\stackrel{\text{def}}{=} \mathit{Address}(\mathit{PAC}(\mathit{RS}^S(\gamma, n), m)) && (184)
\end{aligned}
$$

where

$$
\begin{aligned}
m &= \mathit{LE}^{64}(32 \cdot 2^d) && (185)\\
d &= \texttt{SAMPLE\_DEPTH} \text{ (see A.4)} && (186)
\end{aligned}
$$

Definition 33 – Weighted selection.
We define $\mathit{WeightedSelect}(w, k)$ as a sampler function which selects an index $0 < i < \mathit{len}(w)$ such that the indexes have a probability of being selected according to the weights in w determined by the input nonce (pseudorandom number) k.

$$
\begin{aligned}
\mathit{WeightedSelect} &: \mathit{uint256+} \times \mathit{uint256} \mapsto \mathit{uint256+} && (187)\\
\mathit{WeightedSelect}(w, k) &\stackrel{\text{def}}{=} \begin{cases} i & \text{if } k < w\,[i] \mod W\,[i] \\ \mathit{WeightedSelect}(w\,[:i], k) & \text{otherwise} \end{cases} && (188)
\end{aligned}
$$

such that $i = \mathit{len}(w) - 1$, and where W (cumulative weights) is defined as

$$
\begin{aligned}
W &: \mathit{uint256+} \mapsto \mathit{uint256} && (189)\\
W\,[i] &\stackrel{\text{def}}{=} \begin{cases} w\,[0] & \text{if } i = 0 \\ W\,[i-1] + w\,[i] & \text{otherwise} \end{cases} && (190)
\end{aligned}
$$

Definition 34 – Truth selection.
We determine the truth from reveals through selection weighted by stake density using the truth selection nonce as random input.

$$
\begin{aligned}
\mathit{Truth} &: \Gamma \mapsto \mathit{RevealEntry} && (191)\\
\mathit{Truth}(\gamma) &\stackrel{\text{def}}{=} R(\gamma)\,[\mathit{WeightedSelect}(\mathit{weights}, \mathit{TSN}(\gamma))] && (192)
\end{aligned}
$$

where weights are stake densities such that

$$
\mathit{weights}\,[i] = \mathit{Stake}(\gamma, \textsc{node}(R\,[i])) \cdot 2^{\textsc{depth}(R\,[i])} \qquad (193)
$$

where R is the reveals of the round sorted by index:

$$R \quad = \quad \overrightarrow{Seq}(\textsc{index}, \mathit{Reveals}(\gamma)) \tag{194}$$

Definition 35 – Honest reveals.
We define honest reveals as the subset of reveals for the round agreeing with the truth in reserve commitment hashes and storage depth.

$$\begin{aligned} \mathit{HonestReveals} &: \Gamma \mapsto \mathit{RevealEntry}* & (195)\\ \mathit{HonestReveals}(\gamma) &\overset{\text{def}}{=} \overrightarrow{Seq}(\textsc{index}, \{r \in \mathit{Reveals}(\gamma) | \mathit{Honest}(r)\}) & (196)\end{aligned}$$

where

$$\begin{aligned} \mathit{Honest} &: \mathit{RevealEntry} \mapsto \{\texttt{T},\texttt{F}\} & (197)\\ \mathit{Honest}(r) &\leftrightarrow & (198)\\ & \textsc{chc}(r) = \textsc{chc}(\mathit{truth}(\gamma)) \ \wedge & (199)\\ & \textsc{chs}(r) = \textsc{chs}(\mathit{truth}(\gamma)) \ \wedge & (200)\\ & \textsc{depth}(r) = \textsc{depth}(\mathit{truth}(\gamma)) & (201)\end{aligned}$$

Definition 36 – Winner selection.
We determine the winner from honest reveals through selection weighted by stake using the winner selection nonce as random input.

$$\begin{aligned} \mathit{Winner} &: \Gamma \mapsto \mathit{RevealEntry} & (202)\\ \mathit{Winner}(\gamma) &\overset{\text{def}}{=} \mathit{HonestReveals}(\gamma)\texttt{[}\mathit{WeightedSelect}(\mathit{weights}, \mathit{WSN}(\gamma))\texttt{]} & (203)\end{aligned}$$

where weights are stakes such that

$$\mathit{weights}\texttt{[}i\texttt{]} \quad = \quad \mathit{Stake}(\gamma, \textsc{node}(\mathit{HonestReveals}\texttt{[}i\texttt{]})) \tag{204}$$

Proofs of reserve

Definition 37 – Proof of reserve .
Proof of reserve provides evidence that the reserve is replicating relevant content and shows a proof of recency of retaining chunk data in full integrity.

$$\begin{aligned} \mathit{POR} &: \mathit{SIP}^2 \times \mathit{Stamps} & (205)\\ \Pi^{\text{R}} &: \Gamma \times \mathit{Chunks} \times \mathit{Chunks}+ \times \{0,1,2\} \mapsto \mathit{POR} & (206)\\ \Pi^{\text{R}}(\gamma, c, C, k) &\overset{\text{def}}{=} \langle & (207)\\ \textsc{witness proof} \quad & \Pi^{\text{SIP}}(c, d \cdot i), & (208)\\ \textsc{retention proof} \quad & \Pi^{\text{SIP}}(C\texttt{[}i\texttt{]}, j), & (209)\\ \textsc{postage stamp} \quad & \mathit{Stamp}(C\texttt{[}i\texttt{]}) & (210)\\ & \rangle & (211)\end{aligned}$$

where

$$\begin{aligned} i &= \mathcal{W}(\gamma, len(C), k) && (212)\\ j &= SSN(\gamma, k) \mod 128 && (213)\\ d &= \frac{SegCnt(c)}{len(C)} \left(= \begin{cases} 1 & \text{if } C = RS^S \\ 2 & \text{if } C = RS^C \end{cases} \right) && (214) \end{aligned}$$

Definition 38 – Proof of reserve validation.

$$\begin{aligned} \mathbb{V}^{\text{R}} &: \Gamma \times Nodes \times Address \times \{0,1,2\} \times POR \mapsto \{\texttt{T},\texttt{F}\} && (215)\\ \mathbb{V}^{\text{R}}(\gamma, n, ch, k, \pi) &\leftrightarrow && (216)\\ \text{RELEVANCE} \quad & \mathbb{V}^{\text{SIP}}(ch, d \cdot i, p_w) \wedge && (217)\\ & \mathbb{V}^{\text{STAMP}}(ps, \gamma, n) \wedge && (218)\\ \text{RETENTION} \quad & \text{DATA}(p_w) = a \wedge && (219)\\ & \mathbb{V}^{\text{SIP}}(a, j, p_r) \wedge && (220)\\ \text{RECENCY} \quad & i = \mathcal{W}(\gamma, SegCnt(p_w), k) \wedge && (221)\\ & j = SSN(\gamma, k) \mod 128 \wedge && (222)\\ \text{RETRIEVABILITY} \quad & PO(a, n) \geq sd && (223) \end{aligned}$$

where

$$\begin{aligned} a &= \text{ADDRESS}(ps) && (224)\\ \pi &= \langle p_w, p_r, ps \rangle && (225)\\ sd &= \text{DEPTH}(Reveal(\gamma, n)) && (226)\\ d &= \frac{SegCnt(p_w)}{2^D} \left(= \begin{cases} 1 & \text{if } C = RS^S \\ 2 & \text{if } C = RS^C \end{cases} \right) && (227)\\ D &= \texttt{SAMPLE\_DEPTH} \text{ (see A.4)} && (228) \end{aligned}$$

Definition 39 – Proof of chunk density validation.
Define $\mathbb{V}^{\text{CD}}(\gamma, p_0, p_1, p_2)$ as the validation function for the proof of chunk density for round γ and proof of reserve and segment inclusion proof pairs p_0, p_1, p_2.

$$\begin{aligned} PORT &: POR \times SIP && (229)\\ \mathbb{V}^{\text{CD}} &: \Gamma \times PORT^3 \mapsto \{\texttt{T},\texttt{F}\} && (230)\\ \mathbb{V}^{\text{CD}}(\gamma, \pi_0, \pi_1, \pi_2) &\leftrightarrow && (231)\\ \text{RIGHT DATA SEGMENT} \quad & \text{DATA}(pr_k) = \text{DATA}(pt_k) \text{ for } k \in \{0,1,2\} \wedge && (232)\\ \text{RIGHT ADDRESS} \quad & \text{SISTER}(pw_k, 0) = ta_k \text{ for } k \in \{0,1,2\} \wedge && (233)\\ \text{RIGHT ORDER} \quad & ta_0 < ta_1 < ta_2 \wedge && (234)\\ \text{RIGHT SIZE} \quad & ta_2 \leq \texttt{MAX\_SAMPLE\_VALUE} && (235) \end{aligned}$$

where for $k \in \{0,1,2\}$

$$\begin{aligned} ta_k &= H_\Pi(p, j_k, pt_k) && (236)\\ j_k &= SSN(\gamma, k) \mod 128 && (237)\\ \pi_k &= \langle\langle pw_k, pr_k, \_\rangle, pt_k\rangle && (238) \end{aligned}$$

and where

$$p = RSS(Prev(\gamma)) \text{ (see definition 27)} \qquad (239)$$

Definition 40 – Proof of stamp density validation.
Define $\mathbb{V}^{\text{SD}}(\gamma, ps_0, ps_1, ps_2)$ as the validation function for the proof of stamp density for round γ and postage stamps ps_0, ps_1, ps_2.

$$\begin{aligned} \mathbb{V}^{\text{SD}} &: \Gamma \times Stamps^3 \mapsto \{\texttt{T,F}\} && (240)\\ \mathbb{V}^{\text{SD}}(\gamma, ps_0, ps_1, ps_2) &\leftrightarrow && (241)\\ \text{RIGHT ORDER} &\quad ta_0 < ta_1 < ta_2 \wedge && (242)\\ \text{RIGHT SIZE} &\quad ta_2 \leq \texttt{MAX\_SAMPLE\_VALUE} && (243) \end{aligned}$$

where for $k \in \{0,1,2\}$

$$ta_k = H(Slot(ps_k) \oplus p) \qquad (244)$$

and where

$$p = RSS(Prev(\gamma)) \text{ (see definition 27)} \qquad (245)$$

Definition 41 – Proof of entitlement.
Proof of entitlement captures all the evidence a node needs to submit with their claim transaction to valildate.

$$\begin{aligned} POE &: PORT^3 \times POR^3 && (246)\\ \Pi^{\text{ENT}} &: \Gamma \times Nodes \mapsto POE && (247)\\ \Pi^{\text{ENT}}(\gamma, n) &\stackrel{\text{def}}{=} \langle && (248)\\ &\langle \Pi^{\text{POR}}(\gamma, n, PAC^C(crs, p), crs, 0), pt_0\rangle, && (249)\\ &\langle \Pi^{\text{POR}}(\gamma, n, PAC^C(crs, p), crs, 1), pt_1\rangle, && (250)\\ &\langle \Pi^{\text{POR}}(\gamma, n, PAC^C(crs, p), crs, 2), pt_2\rangle, && (251)\\ &\Pi^{\text{POR}}(\gamma, n, PAC(srs), srs, 0), && (252)\\ &\Pi^{\text{POR}}(\gamma, n, PAC(srs), srs, 1), && (253)\\ &\Pi^{\text{POR}}(\gamma, n, PAC(srs), srs, 2), && (254)\\ &\rangle && (255) \end{aligned}$$

where for $k \in \{0,1,2\}$

$$\begin{aligned} pt_k &= \Pi^{\text{SIP}}_{prefix}(p, r\texttt{[}i_k\texttt{]}, j_k) && (256)\\ j_k &= SSN(\gamma, k) \mod 128 && (257)\\ i_k &= \mathcal{W}(\gamma, length(crs), k) && (258) \end{aligned}$$

and where

$$\begin{aligned} crs &= RS^C(\gamma, n) && (259)\\ srs &= RS^S(\gamma, n) && (260)\\ p &= RSS(Prev(\gamma)) \text{ (see definition 27)} && (261) \end{aligned}$$

Definition 42 – Winner's claim validation.
Define $\mathbb{V}^{\text{POE}}(\gamma, p)$ as the validation function for the proof of entitlement p as part of the winning claim for game γ:

$$\begin{aligned} \mathbb{V}^{\text{POE}} &: \Gamma \times POE \mapsto \{\texttt{T},\texttt{F}\} && (262)\\ \mathbb{V}^{\text{POE}}(\gamma, p) &\Leftrightarrow && (263)\\ \text{RESERVE:} & && (264)\\ \text{CHUNKS} & \quad \forall k \in \{0,1,2\}, \mathbb{V}^{\text{R}}(\gamma, n, \text{CHC}(r), k, \pi_k) \wedge && (265)\\ \text{STAMPS} & \quad \forall k \in \{0,1,2\}, \mathbb{V}^{\text{R}}(\gamma, n, \text{CHS}(r), k, \phi_k) \wedge && (266)\\ \text{RESERVE SIZE:} & && (267)\\ \text{CHUNK DENSITY} & \quad \mathbb{V}^{\text{CD}}(\gamma, \langle\pi_0, pt_0\rangle, \langle\pi_1, pt_1\rangle, \langle\pi_2, pt_2\rangle) \wedge && (268)\\ \text{STAMP DENSITY} & \quad \mathbb{V}^{\text{SD}}(\gamma, \text{PS}(\phi_0), \text{PS}(\phi_1), \text{PS}(\phi_2)) && (269) \end{aligned}$$

where

$$\begin{aligned} n &= \text{NODE}(r) && (270)\\ r &= Winner(\gamma) && (271)\\ p &= \langle\langle\pi_0, pt_0\rangle, \langle\pi_1, pt_1\rangle, \langle\pi_2, pt_2\rangle, \phi_0, \phi_1, \phi_2\rangle && (272) \end{aligned}$$

Corollary 43 – Outpayment scheme is fair.
Rewarding the pot to randomly selected neighbourhood implements a redistribution scheme that is fair across neighbourhoods.

Proof. With the consensus mechanism we can show that the Nash-optimal strategy of nodes is to follow the protocol and consent on the reserve. On the other hand, the optimal strategy for uploaders is to uniformly distribute chunks across the name space. As a consequence, nodes are expected to have identical storage depth and its variance is independent of being chosen. Long term then relative cumulative outpayments by the redistribution game converge to the fair share.

Secondly, we argue that the mode of selecting the winner is fair within neighbourhoods.

Proof.

A.3 Price oracle

The price oracle smart contract receives input from the redistribution game. Notably the input constitutes information on storage supply relative to the current demand. Since demand is maxed out with storage depth, this information is simply captured by the number of honest revealers per neighbourhood. Rounds when there are no revealers should be treated as rounds where the number of revealers is zero therefore the number of skipped rounds is passed to the price update function too.

Definition 44 – price function .
The *Price* function determines the unit price of storage (PLUR/chunk/block). It is defined in such a way that the ratio between the price of rent in consecutive rounds is an exponential function of supply. Supply is the deviation from the optimal number of peers in a neighbourhood as measured by the number of honest revealers in the previous round.

$$\mathit{Price} \quad : \quad \Gamma \mapsto \mathit{uint256} \tag{273}$$

$$\mathit{Price}(\gamma) \overset{\text{def}}{=} \mathit{Price}(\mathit{Prev}(\gamma)) \cdot 2^{\sigma \cdot d} \tag{274}$$

where

$$\text{SUPPLY} \qquad d = \texttt{NHOOD\_PEER\_COUNT} - |\mathit{Reveals}(\gamma)| \tag{275}$$

$$\text{REACTIVITY} \qquad \sigma = \frac{1}{\texttt{PRICE\_2X\_ROUNDS}} \tag{276}$$

First, note that for a consistent deviation signal d, the price change after n rounds is given by multiplying the starting price with $\left(2^{\sigma \cdot d}\right)^n$. The price has doubled if this expression is 2, i.e., $n \cdot d \cdot \sigma = 1$. With the lowest signal of undersupply ($r = 3$), $d = 1$, and therefore the reactivity parameter is expressed as $\sigma = \frac{1}{n}$. In other words $\frac{1}{\sigma}$ measures the number of rounds it takes for the price to double.

A.4 Parameter constants

Table 2 lists the constants used by Swarm with their type, default value and description.

CONSTANT NAME	TYPE	VALUE	DESCRIPTION
`BZZ_NETWORK_ID`	*uint64*	0	ID of the swarm network
`PHASE_LENGTH`	*uint64*	38	length of commit phase of the redistribution game round in number of blocks
`ROUND_LENGTH`	*uint64*	152	length of one redistribution game round in number of blocks, fixed at 4 times the `PHASE_LENGTH`
`NODE_RESERVE_DEPTH`	*uint8*	23	size requirement for client reserve capacity given in log number of chunks
`SAMPLE_DEPTH`	*uint8*	3	base 2 log number of chunks in sample
`MAX_SAMPLE_VALUE`	*uint256*	6.73e+71	maximum value for last transformed address in reserve sample, i.e., < 1% chance the sampled set size is below a fourth of the prescribed node reserve size.

`MINIMUM_STAKE`	*uint256*	10	minimum stake amount in BZZ to be redefined as minimum stake given in storage rent units
`MIN_STAKE_AGE`	*uint256*	228	minimum number of blocks stakers need to wait after update or creation for the stake to be useable. Defaults to one and a half rounds to prevent opportunistic manipulation of stake after a neighbourhood is selected.
`PRICE_2X_ROUNDS`	*uint64*	64	number of rounds it takes for the price to double in the presence of a consistent lowest degree signal of undersupply.
`NHOOD_PEER_COUNT`	*uint8*	4	minimum number of nodes required to form a fully connected neighbourhood.

Table 2: Parameter constants

B Source of randomness

As the *neighbourhood selection anchor* will directly affect which neighbourhood wins the pot, it is prudent to derive the randomness from a source of entropy that cannot be manipulated. A common solution to obtain randomness is to have independent parties committing to a random nonce with a stake. The random seed for a round is defined as the xor of all revealed nonces. Given the nonces are independent sources fixed in the commit, no individual participant has the ability to skew randomness by selecting a particular nonce. Thanks to the commutativity of xor, the order of reveals is also irrelevant. However, if the reveal transactions are sequential, committers compete at holding out since the last one to reveal can effectively choose the resulting seed to be either including its committed nonce (if they do reveal) or not (if they do not reveal). The threat to slash the stake of non-revealers serves to eliminate this degree of freedom from last revealers and thus renders this scheme a secure random oracle assuming there is at least one honest (non-colluding) party.[27]

Now note that the redistribution scheme already has a commit reveal scheme as well as stake slashed for non-revealers, so a potential random oracle is already part of the proposed scheme. Incidentally, the beginning of the claim phase is when new randomness is needed to select the truth and a winner. Importantly, these random values are only needed if there is a claim which implies that there were some commits and reveals to choose from. Or conversely, if there are no reveals in the round,[28] the random seed is undefined but is also not needed for the claim.

The random seed that transpires at the beginning of the claim phase can serve as the *reserve sample salt* (nonce input to modified hash used in the sampling) with which the nodes in the selected neighbourhood can start calculating their reserve sample.

[27]If the stake is higher than the reward pot, one cannot afford being slashed with even just one commit without a loss. If this cannot be guaranteed, slashing of the stake is not an effective deterrent.

[28]If saboteurs get slashed or frozen in the claim transaction, if there is no claim, the committers get away without being punished. This can be remedied if the staking contract keeps a flag on each overlay (set when commits, unsets when reveals in the same round) and the check and punishment happens as a result of a commit call in the case the flag is found set.

The neighbourhood for the next round is selected by the *neighbourhood selection anchor*, which is, similarly to the truth and winner selection nonces, deterministically derived from the same random seed. Unlike the nonces used to select from the reveals, neighbourhood selection should be well defined for the following round even if a round is skipped, i.e., when there is no reveals. To cover this case skipped rounds keep the random seed of the previous round. However, in order to rotate selected neighbourhoods through skipped rounds, we derive the neighbourhood selection anchor from the seed by factoring in the number of game rounds passed since the last reveal.[29]

In order to provide protection against the case when each committer in the neighbourhood is colluding, and can afford losing stake we need to make sure that the entropy is still high otherwise the nodes can influence the neighbourhood selection nonce and reselect themselves or a fixed colluding neighbourhood (or increase the chances of reselection).

Definition 45 – Random seed for the round.
Define the random seed of the round as the xor of all obfuscation keys sent as part of the reveal transaction data during the entire reveal period:

$$\mathcal{R} \quad : \quad \Gamma \mapsto \mathit{Nonce} \tag{277}$$

$$\mathcal{R}(\gamma) \quad \overset{\text{def}}{=} \quad \begin{cases} \mathcal{R}(\mathit{Prev}(\gamma)) & \text{if } |\mathit{Reveals}(\gamma)| = 0 \\ \underline{\bigvee}_{r \in \mathit{Reveals}(\gamma)} \quad \text{NONCE}(r) & \text{otherwise} \end{cases} \tag{278}$$

Lemma 46 – Round seed is a secure random oracle.
The nonce produced by xoring the revealed obfuscation keys is a correct source of entropy.

Proof. Assuming n independent parties committing, choosing any particular nonce will leave the outcome fully random.

C Density-based size estimation

Nodes in Swarm must utilise their full reserve capacity: nodes will potentially further replicate chunks in case they have unused reserve capacity beyond storing their share necessary for a system-wide level of redundancy required. To incentivise this, participating in the redistribution game involves a check called *proof of resources* which is supposed to verify the size of reserve from which the reserve samples are generated. The insight here is that the sample is the lowest range of a uniformly random variate over the entire 256-bit address space. Intuitively, the higher the original volume of the sampled set, the denser it is, the lower the expected maximum value in the sample. Conversely, a constraint on the maximum value of the last element in the sample practically puts a minimum cardinality requirement on the sampled set using a solution called *density based set size estimation.*

[29]Otherwise a selected neighbourhood could collude maliciously not to commit/reveal and have the pot roll over to the following round. By simply holding out for a number of redistribution rounds, they could unfairly multiply their reward when they eventually claim the pot.

We are given n independent, uniformly distributed values between 0 and 1.[30] Let the value of the kth smallest of these be x_k (so the smallest of the n values is x_1, the second smallest is x_2, and so on, up to x_n). What is the distribution of x_k, given n? And what is the threshold value u such that for any given probability α, the chance of obtaining an x_k lower than u is α?

Note that the distance between any two adjacent values out of n independent uniform variates follows an exponential distribution, as long as n is sufficiently large.[31] The rate parameter of this exponential distribution is $n+1$, where n is increased by one to account for the fact that the expected gap between adjacent values is $1/(n+1)$.[32]

We are after the distribution of the kth value, [33] x_k, which then can be thought of as arising from the sum of k independent exponential variables, each with a rate parameter of $n+1$. This is known to result in an Erlang distribution with shape parameter k and rate parameter $n+1$. Using $X(\lambda)$ to denote the exponential distribution with rate λ and $E(k,\lambda)$ to denote the Erlang distribution with shape k and rate λ:

$$\sum_{i=1}^{k} X_i(n+1) = E(k, n+1), \tag{279}$$

where the subscript i in $X_i(n+1)$ distinguishes between independent exponentially distributed random variables. The probability density function $E(x,k,n+1)$ of the Erlang distribution itself is given by

$$E(x,k,n+1) = \frac{(n+1)^k x^{k-1} \mathrm{e}^{-(n+1)x}}{(k-1)!}. \tag{280}$$

This distribution contains the answer to the first question: what is the distribution of the kth smallest value out of n independent uniform variates between 0 and 1? For example, if $k=16$ and n is either 500, 750, or 1000, we get the distributions shown in Figure 16.

We can now answer the second question: given n and a confidence level α, what is the threshold value u for x_k such that the probability that $x_k < u$ is equal to α? That is, we wish to know the value $x = u$ at which the probability distribution has encompassed a given area of α (see figure 17).

The area under the curve of the Erlang distribution is given by its cumulative distribution function $P(x,k,n+1)$, which is known to be

$$P(x,k,n+1) = \int_0^x E(y,k,n+1)\,\mathrm{d}y = \frac{1}{(k-1)!}\int_0^{(n+1)x} t^{k-1}\mathrm{e}^{-t}\,\mathrm{d}t. \tag{281}$$

[30]Because we work with densities, the actual integer range is not relevant and results obtained for the unit interval can simply be rescaled to the Swarm use case by multiplying with 2^{256}.

[31]This follows from the fact that n independent uniform variates can be thought of as realizing a Poisson process, whereby the timing of events is random, and it is known that the nearest-neighbour distribution (i.e., waiting time between two consecutive events) is then exponentially distributed.

[32]For $n=2$, the mean outcome is $x_1 = 1/3$ and $x_2 = 2/3$; for $n=3$, it is $x_1 = 1/4$, $x_2 = 2/4$, $x_3 = 3/4$; and so on: for arbitrary n, $x_i = i/(n+1)$, with the gap between adjacent values in this ideal case always being $1/(n+1)$.

[33]An alternative approach using order statistic expresses x_k via a beta distribution. It is very difficult to prove that the Beta distribution's quantile function is a strictly decreasing function of n, which is a key piece of the argument presented here. Although this method is exact even for small n, in our case, n always a very large number, therefore we adopted the other method.

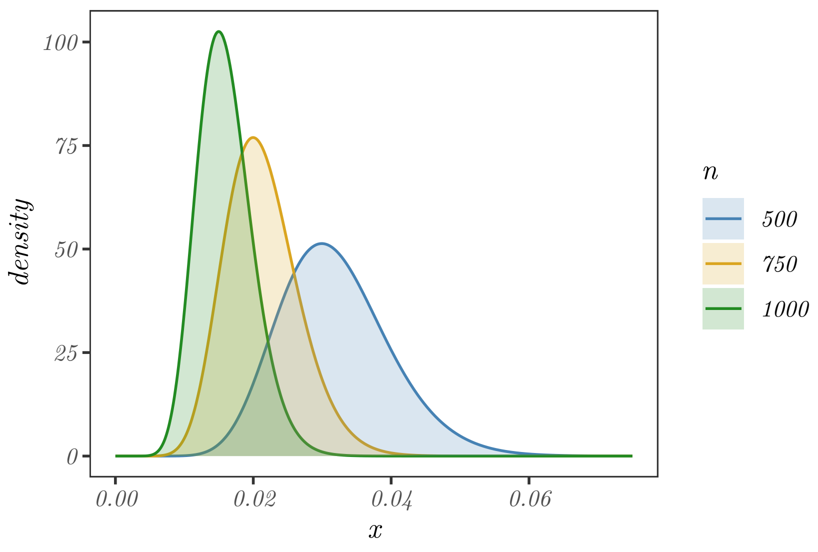

Figure 16: Probability density function $E(x, k, n+1)$ of the Erlang distribution, with $k = 16$ and n either 500, 750, or 1000.

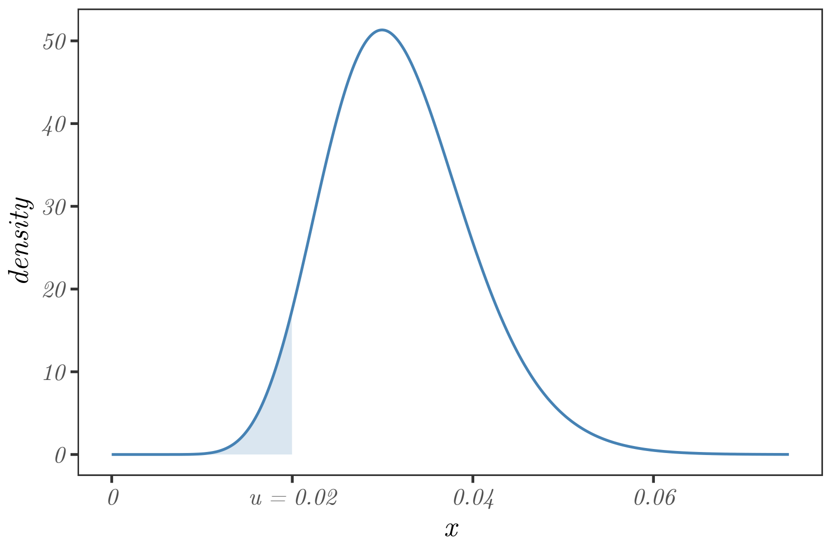

Figure 17: The distribution of x_{16}, i.e., the 16th smallest value from among $n = 500$ independently and uniformly drawn variates between 0 and 1. The area under the curve is shaded up to 5% of its area. The point at which the shading stops is therefore the value u for which there is only a 5% chance of getting an even smaller x_{16}.

The latter expression is sometimes written as $\tilde{\gamma}(k, (n+1)x)$, where

$$\tilde{\gamma}(k, x) = \frac{1}{\Gamma(k)} \int_0^x t^{k-1} \mathrm{e}^{-t} \, \mathrm{d}t \tag{282}$$

is the regularized lower incomplete gamma function. We therefore want to solve the equation

$$\alpha = \int_0^u E(y, k, n+1) \, \mathrm{d}y = P(u, k, n+1) \tag{283}$$

for u.

Inverting this expression in u (since the cumulative distribution function increases monotonically in u, the inverse exists) leads to the quantile function $Q(\alpha, k, n+1)$ of the Erlang distribution: $u = Q(\alpha, k, n+1)$. The quantile function is known to be expressible as

$$Q(\alpha, k, n+1) = \frac{\tilde{\gamma}^{-1}(k, x)}{n+1}, \tag{284}$$

where $\tilde{\gamma}^{-1}(k, x)$ is the inverse regularized lower incomplete gamma function. Its particular form is of no interest to us, except for two properties. First, it is positive for all x.[34] Second, it is independent of n. Instead, the entire dependence of $Q(\alpha, k, n+1)$ on n is given by the $n+1$ term in the denominator of Equation 284. From this, we conclude that $Q(\alpha, k, n+1)$ is a strictly decreasing function of n.

These two points lead to an important consequence. Say we compute the threshold u for a given α and n in order to have an upper bound on a lower quantile. Now, if we were to decrease n but hold all other things equal, the threshold will always get higher than what it was before. The threshold obtained for higher values of n may therefore serve as a conservative estimate of the threshold for lower values: if u is a threshold such that the kth smallest out of n uniform variates is only smaller than u in α of cases, then for any amount $m < n$, the chance of the kth variate conforming to the same constraint (i.e., $x_k < u$) is now even smaller than α.

Conversely, if we were to constrain x_k so that the probability of not getting a value smaller than u is lower than β (minimising a higher quantile), we find that the constraint remains true as n is increased.

Armed with these results, let us see how Equation 284 can be used for the estimation procedure. There are two problems to tackle, ultimately relating to the two aspects of a test's accuracy. First, we want to catch inadequate storers slacking on volume. In other words, we want to constrain the x_k values so that we can safely say that any attacker with a stored volume below an acceptable size n has a probability less than α to obtain such a small x_k by pure chance. Construing the condition for $x_k < u$ as a test to filter honest players (just based on the size of their reserve), $1-\alpha$ expresses the *sensitivity* of the test. From the previous argument on the monotonic dependence of α on n, it is safe to use a condition that requires x_k to stay below a threshold obtained for n.

Second, we want to avoid situations when honest participants end up not satisfying the above constraint even though they sampled from a set larger than the required minimum. Given a target volume $m > n$, the error rate of false negatives is guaranteed to be less than β obtained from the quantile function with parameters $m, k,$ and u. The quantity $1-\beta$ is the *specificity* or *precision* of the test.

Figure 18 illustrates this idea, for two different distributions in both the lower- and upper-end estimation. What we want is to choose u to simultaneously make sure that dishonest players do not sneak through the system *and* also that honest players do not get excluded too often. This translates to make both α and β as small as possible.

One way to try and find the best compromise is by minimising $\alpha + \beta$ (the *accuracy* of the test) and pick the u value at the optimum to be used in the proof of resources test. To this end, one

[34]This stands to reason: the quantile function of a distribution on $x \in [0, \infty)$ is itself between 0 and ∞, and $\tilde{\gamma}^{-1}(k, x)$ is just the quantile function of the Erlang distribution times the positive constant $n+1$.

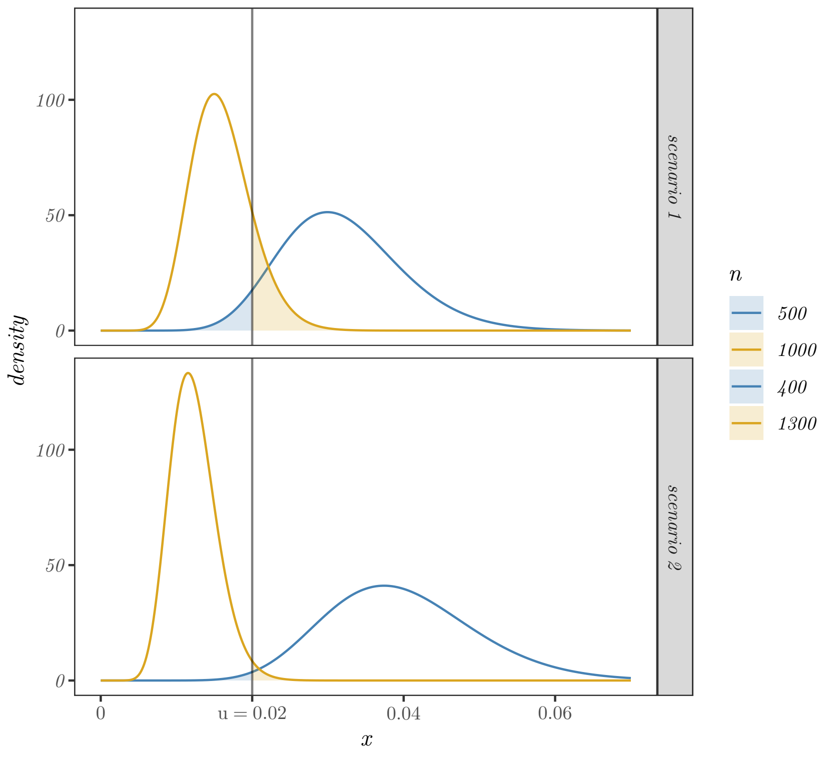

Figure 18: Recall and precision of proof of reserve size validation: Any chosen u will lead to different α and β values, depending on n. Here u is fixed at 0.02. The top panel shows distributions for $x \equiv x_{16}$ with $n = 500$ (blue) and $n = 1000$ (yellow). The area left under the blue curve to the left of u is equal to α (blue shade); the area under the yellow curve right of u is equal to β (yellow shade). If the curves overlap considerably (top), it is impossible to choose an u such that α and β are simultaneously small.

can vary α between 0 and 1 and, for each of its values, solve the equation $\alpha = Q(1-\beta, k, n+1)$ (where n is the larger value, used for estimating β). This way, we get a β value for every possible α. Then, we can find the combination which minimises $\alpha + \beta$, and determine the value of u that leads to this optimum. As illustrated in figure 19, larger values of k yield a trade-off curve along which better accuracies can be achieved.

Table 3 summarizes the important numerical results in Figure 19.[35]

k	α	β	u	$u \cdot 2^{256}$
8	0.196216	0.1373622	$5.54589 \cdot 10^{-6}$	$6.421705 \cdot 10^{71}$
16	0.097612	0.0716570	$1.10923 \cdot 10^{-5}$	$1.284401 \cdot 10^{72}$
32	0.029386	0.0219151	$2.21962 \cdot 10^{-5}$	$2.570140 \cdot 10^{72}$

Table 3: Proof of density parameter calibration. Assuming $n = 10^6$ and $m = 2 \cdot 10^6$ to calculate recall and precision error rates α and β, respectively, the cutoff value for the proof is calibrated by optimizing on acccuracy using sample sizes $8, 16, 32$.

[35]Since the hash function used to generate random variates does so in the range $[0,\ 2^{256}-1]$ instead of $[0,\ 1]$, the calculated thresholds are scaled with 2^{256} to show where they would fall in their actual range.

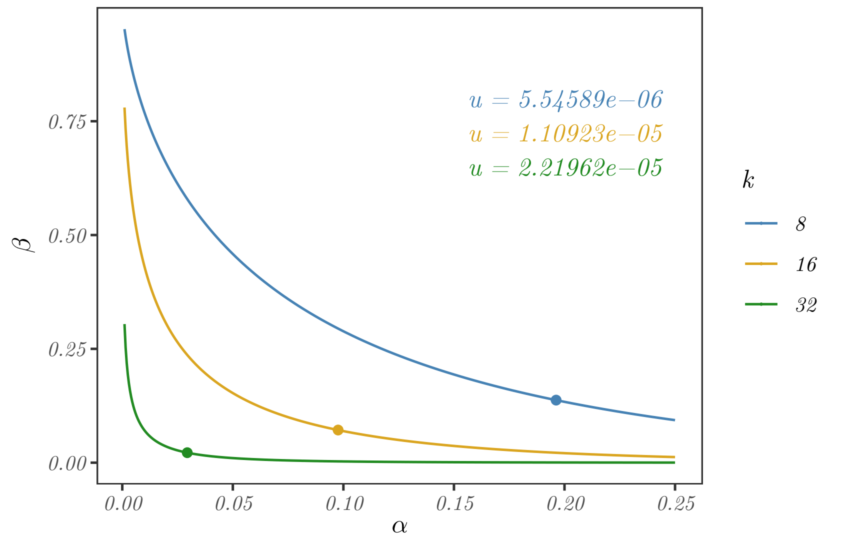

Figure 19: Optimal accuracy of reserve size probe By increasing k, one can get better optima for minimising $\alpha + \beta$. Here $n = 10^6$ for estimating α and $2 \cdot 10^6$ for estimating β, and k is either 8, 16, or 32 (colors). The values of u associated with the optima are also shown.

D Batch utilisation

When pushing content to the Swarm network, uploaders are required to attach an attestation of storage rent prepayment to each chunk they post. The latter is essentially a wallet registered on the postage contract seeded with a balance from which storage rent is deduced by the network-wide incentive system. Since this is reminiscent of buying a batch of postage stamps and attaching one to each envelope to be posted, the attestations are called *postage stamps* and the registered wallet a *postage batch.* One can think of batches as collections of *storage slots.* The size of a batch is the number of storage slots and is always specified as a power of 2, with the exponent called *batch depth.* Each slot can hold at most one chunk. Putting a chunk into a slot is like issuing a postage stamp.

In practice, the attached postage stamps are digital signatures which associate the address of a chunk with a *storage slot reference.* This in turn is composed of 1) a reference to the wallet through its *batch ID,* and 2) a *within-batch index.* The fact that each slot can hold at most one chunk ensures that batches cannot issue more stamps than the volume registered with them. However, for an *overissuance* incident to be detectable locally by storer nodes, the within-batch indices are arranged such that the highest n bits match the prefix of the chunk they are assigned to. These n bits define 2^n *buckets*, the other half of the index is essentially a counter within the buckets, which is sequentially assigned to chunks. If the batch depth is d and there are 2^n buckets, then each bucket will hold a maximum of 2^{d-n} chunks. $k = d - n$, the log size of a bucket is called *bucket depth.* The bucket size (2^k) provides an exclusive upper bound to within-bucket indices. This enforces a uniformity of stamp issuance across the 2^n buckets, therefore n is called *uniformity depth.*

Overusing a batch is now easily detected by any storer node as long as their storage depth is shallower than the batch's uniformity depth. In that case, each bucket of a batch is entirely within the node's reserve. Overissuance is therefore immediately caught, since the multiple chunks assigned to the same slot index are seen by any node in that neighbourhood.

When all slots are filled, we say the batch is *fully utilised.* Although the *a priori* distribution of chunks is uniform (and therefore the expected number of chunks falling into each bucket is the same), their stochastic assignment means that there is necessarily some variance. It is practically impossible to fully fill all buckets before eventually attempting to add to one that is already full. We assume that the uploader is unable to affect the address of the chunks (unencrypted fixed content) so in this scenario they are unable to continue uploading. Because of this, one may legitimately consider the batch to be no longer usable. The number of stamps hitherto issued by the batch is called its *effective batch size,* and its ratio to the batch size its *batch utilization rate.*

Below we explore the effect of batch parameters on their utilization rate. With an insight into utilization rates as a function of the number of buckets 2^n and the size of buckets 2^k, we will have a way to calibrate the expected effective batch size to be presented to users in the context of a batch purchase user experience.

The problem of assigning chunks to storage slots is analogous with the process of throwing marbles, one after the other, in boxes which are initially empty. Each throw may end up in any of the boxes with equal probability, and thus the marbles get distributed across the boxes more or less evenly through time. However, the time will come when the boxes start filling up. At that point, a marble may by chance end up being thrown into a box that is already full, and thus get rejected. Substituting chunks for marbles, buckets for boxes, and the act of signing a stamp for throwing a marble, we recover our original scenario. Marbles ending up in a box with equal probability corresponds to the fact that a random chunk has equal chance of being assigned to a bucket since the hash function has uniform distribution. Repeated rounds of marble throwing correspond to consecutive stamping of multiple chunks of the uploaded content; rounds constitute repeated independent trials.

Taking a particular bucket, each round of stamping is a "success" if the stamp falls into the bucket, and a "failure" otherwise. Due to the fact that a marble may end up in each box with equal chance, the probability of success is $1/n$ and the probability of failure is $1 - 1/n$. The number of stamps issued to the bucket after a given number of rounds is described by the negative binomial distribution $\mathcal{B}(k, 1/n)$, where the first parameter k is the number of failed rounds before we stop counting, and the second parameter is the probability of success.

The negative binomial distribution $\mathcal{B}(k, 1/n)$ is equivalent to the sum of k independent geometrically distributed variables with parameter $1/n$. Unless k is very small, the central limit theorem ensures that this sum converges to a normal distribution. Formally, if $\mathcal{Y}_i(1/n)$ are independent geometrically distributed variables with parameter $1/n$, and $\mathcal{N}(\mu, \sigma^2)$ is the normal distribution with mean μ and variance σ^2, then for large k we have

$$\mathcal{B}(k, 1/n) = \sum_{i=1}^{k} \mathcal{Y}_i(1/n) \approx \mathcal{N}(kn, kn(n-1)) \tag{285}$$

The reason for the above form of the mean and variance in the normal distribution is as follows: given a single geometrically distributed variable with parameter $1/n$, its mean is n and its variance is $n(n-1)$. When adding these up over k independent variables, we arrive at $\mu = kn$ and $\sigma^2 = kn(n-1)$ in the limiting normal distribution.

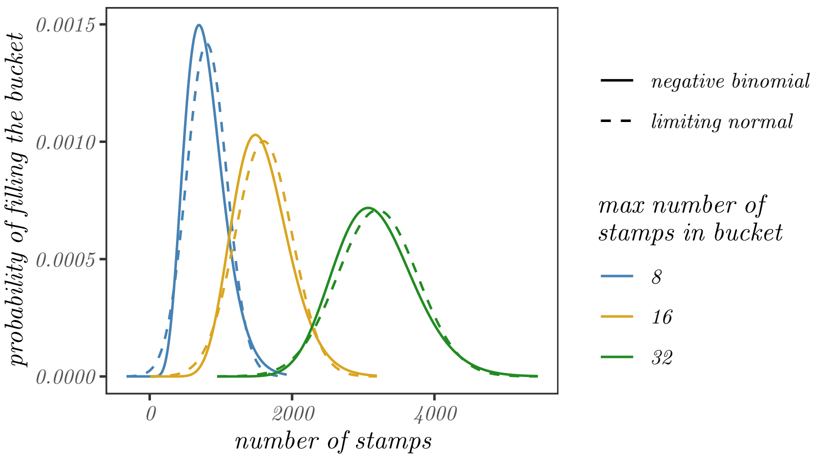

Figure 20: Approximating negative binomial distributions with limiting normal ones. The number of buckets n is fixed at 100.

The fit of the normal distribution with the binomial is robust even for low values of k, as depicted in Figure 20.

The negative binomial distribution estimates the number of rounds a particular bucket fills up, given its size k and the number of buckets n. The rounds of stamping can be conceived of as parallel independent attempts at filling up buckets. Stamps issued for failed rounds still end up in one of the other $n - 1$ buckets, and therefore the same probability variable counts the total stamps issued by the batch in all buckets at the time the first one fills up. Thus, we want to know how the *minimum* of n independent normal variates, $X = \min_{i \in \{1 \dots n\}} \mathcal{N}_i(\mu, \sigma^2)$, is distributed.

This so-called *extreme value distribution* would give us the distribution of the absolute number of total rounds needed until the first event when one of the buckets fills up. Therefore its mean is exactly the expected value of the number of rounds until the first bucket-filling event. Since this is when the batch is considered effectively used up, the mean of the distribution measures the effective utilization of the batch. Dividing by n gives the expected number of stamps per bucket making the rate comparable across any parameter of bucket size. Further dividing by k in turn gives the *expected normalized utilization rate*, which is now comparable across all k and n values.

The extreme value distribution for the minimum of n independent variates drawn from the standard normal distribution (with zero mean and unit variance) is known. It is the Gumbel distribution, which reads

$$\mathcal{G}(x; \alpha, \beta) = \frac{1}{\beta} \exp\left[\frac{x + \alpha}{\beta} - \exp\left(\frac{x + \alpha}{\beta}\right)\right]. \tag{286}$$

Here x is the independent variable (in our case, the number of rounds of stamping), and α and

β are the *location* and *scale* parameters, respectively.[36] They are in turn given by

$$\begin{aligned} \alpha &= \rho\left(1 - \frac{1}{n}\right), \\ \beta &= \rho\left(1 - \frac{\mathrm{e}^{-1}}{n}\right) - \alpha, \end{aligned} \tag{287}$$

where $\mathrm{e}^{-1} = \exp(-1) \approx 0.368$, n is the number of normal variates whose minimum we are looking for, and $\rho(x)$ is the quantile function of the normal distribution (or the inverse of the error function). The mean, standard deviation, and quantile function of the Gumbel distribution are

$$\begin{aligned} \mathbb{E}(\mathcal{G}) &= \alpha + \gamma\beta, \\ \mathbb{S}(\mathcal{G}) &= \beta\,\frac{\pi}{\sqrt{6}}, \\ Q(p;\mathcal{G}) &= \beta\,\log(-\log(1-p)) - \alpha, \end{aligned} \tag{288}$$

where $\gamma \approx 0.5772$ is the Euler–Mascheroni constant, and $0 < p < 1$ in $Q(p;\mathcal{G})$ is a probability quantile.

The normal distribution whose variates' minimum values we are looking for is not standard, but instead has mean $\mu = kn$ (instead of zero), and variance $\sigma^2 = kn(n-1)$ (instead of one). Therefore, the Gumbel distribution needs to be appropriately rescaled. Denoting this scaled probability distribution by $\mathcal{X}$, we have

$$\mathcal{X}(x;\mu,\sigma,\alpha,\beta) = \frac{1}{\sigma}\mathcal{G}\left(\frac{x-\mu}{\sigma};\alpha,\beta\right) \tag{289}$$

(the overall factor of $1/\sigma$ restores the proper normalization of the scaled function). Consequently, the mean, standard deviation, and quantile function should also be rescaled:

$$\begin{aligned} \mathbb{E}(\mathcal{X}) &= \mu - \sigma(\alpha+\gamma\beta) = kn - \sqrt{kn(n-1)}\left[(1-\gamma)\rho\left(1-\frac{1}{n}\right) + \gamma\rho\left(1-\frac{\mathrm{e}^{-1}}{n}\right)\right], \\ \mathbb{S}(\mathcal{X}) &= \sigma\beta\frac{\pi}{\sqrt{6}} = \sqrt{kn(n-1)}\left[\rho\left(1-\frac{\mathrm{e}^{-1}}{n}\right) - \rho\left(1-\frac{1}{n}\right)\right]\frac{\pi}{\sqrt{6}}, \\ Q(p;\mathcal{X}) &= \mu - \sigma\left[\alpha - \beta\,\log(-\log(1-p))\right] \\ &= kn - \sqrt{kn(n-1)}\left[\rho\left(1-\frac{1}{n}\right) - \rho\left(1-\frac{\mathrm{e}^{-1}}{n}\right)\log\left(-\log(1-p)\right)\right]. \end{aligned} \tag{290}$$

Normalizing the mean and the quantile function by the product kn:

$$\begin{aligned} \frac{\mathbb{E}(\mathcal{X})}{kn} &= 1 - \sqrt{\frac{(n-1)}{kn}}\left[(1-\gamma)\rho\left(1-\frac{1}{n}\right) + \gamma\rho\left(1-\frac{\mathrm{e}^{-1}}{n}\right)\right], \\ \frac{Q(p;\mathcal{X})}{kn} &= 1 - \sqrt{\frac{(n-1)}{kn}}\left[\rho\left(1-\frac{1}{n}\right) - \rho\left(1-\frac{\mathrm{e}^{-1}}{n}\right)\log\left(-\log(1-p)\right)\right]. \end{aligned} \tag{291}$$

Taking into account that k and n must both be powers of two, we can write $k = 2^\kappa$ and $n = 2^\nu$, where κ and ν are positive integers. We then have

$$\frac{\mathbb{E}(\mathcal{X})}{kn} = \frac{\mathbb{E}(\mathcal{X})}{2^{\kappa+\nu}} = 1 - \sqrt{\frac{2^\nu - 1}{2^{\kappa+\nu}}}\left[(1-\gamma)\rho\left(1-\frac{1}{2^\nu}\right) + \gamma\rho\left(1-\frac{\mathrm{e}^{-1}}{2^\nu}\right)\right]. \tag{292}$$

[36] The Gumbel distribution is often given using the convention that one is looking for the maximum of n normal variates, instead of their minimum. One can change between the two by simply flipping the sign of x.

In order to explore this solution, we consider the average expected utilization rate as a function of uniformity depth, for various log bucket sizes (see Figure 21). As long as bucket size is around 2^8 or greater, the dependence of the utilization rate on n is milder and milder, and at 2^{16}, it is virtually a flat line at 100%. In general, we see that for the same bucket size, the miss rate (one minus the utilization rate) increases with the number of buckets. Using small bucket sizes (up to 2^{10}), we get unacceptably high miss rates even with few buckets.[37] On the other hand, for larger bucket sizes (over 2^{10}), the miss rate is contained at a tolerable $< 10\%$ (on the right in Figure 21).

On the right, we plot the miss rate on a logarithmic scale as a function of log bucket size, for various numbers of buckets. We find that doubling the batch size 6 times decreases the miss rate tenfold, regardless the actual size or the number of buckets. In fact, as the close parallel lines show (on the right in Figure 21), there is no interaction: going from one bucket to a billion increases the miss rate only by tenfold, independently of bucket size.

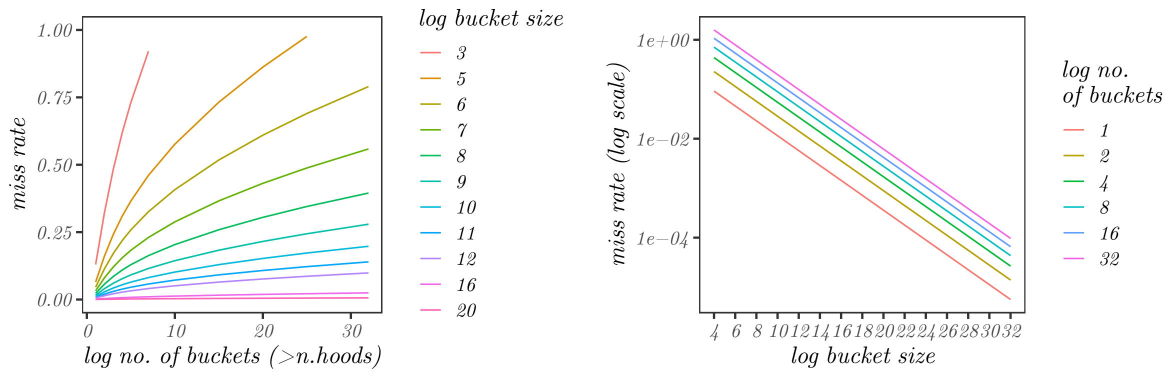

Figure 21: Expected miss rate, for various parameters. Left: miss rate against the number of buckets. Right: miss rate against bucket size.

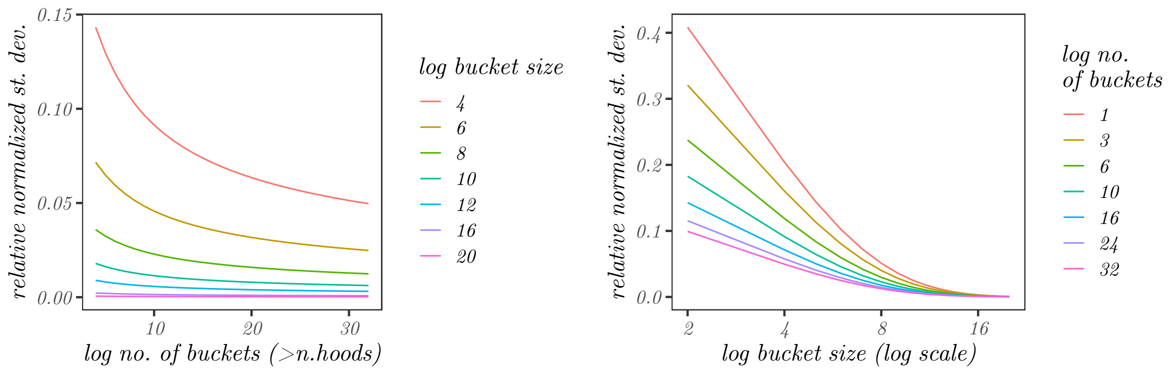

Figure 22: Normalized standard deviation of utilization rate, for various parameters. Left: relative standard deviation against the number of buckets. Right: relative standard deviation against bucket size.

Looking at relative standard deviation of the distribution for various values of k and n (see Figure 22), we find not much variance whenever k and n are large. Nevertheless, for the sake of correctness, we check the quantile function of our extreme value distribution (see Figures 23

[37]Note that for low k but high n, the solution is unreliable, predicting a negative number of stamps per bucket. The prediction will generally not work well if k is too small, because the normal approximation to the negative binomial distribution starts to break down.

and 24).

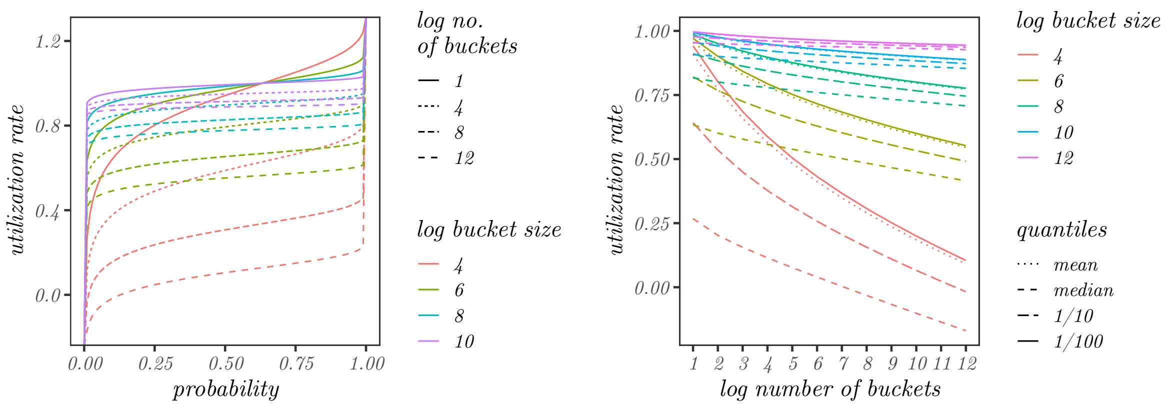

Figure 23: Quantiles of the utilization rate, for various parameters. Left: the quantile function of the utilization rate. Right: mean, median, 10%, and 1% percentiles of utilization against the number of buckets.

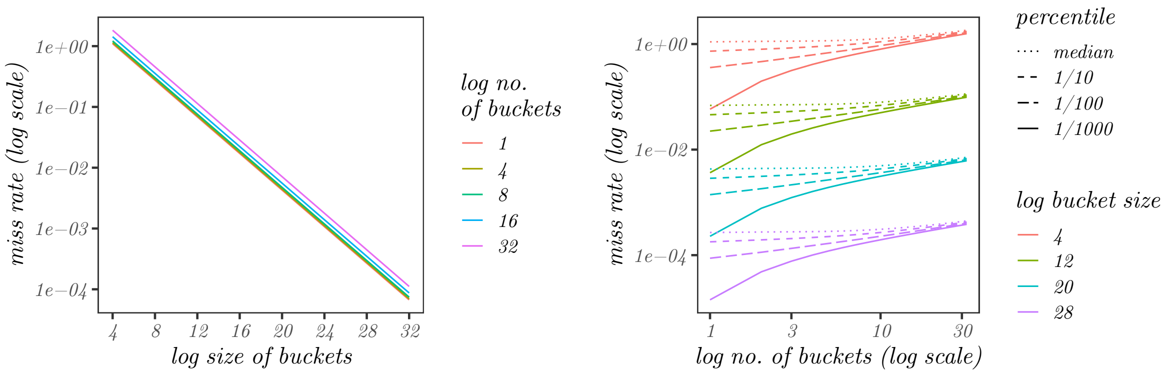

Figure 24: Quantiles of the miss rate for various parameters. Left: one-in-a-thousand quantile for miss rate against bucket size. Right: miss rate in terms of percentiles, against number of buckets.

Eventually, we aim to use the information on utilization rates to protect the user from being surprised at the effective utilization of their postage batch. To this end, we apply the volume calibration using a reasonable worst-case scenario, i.e., define the effective utilization rate as the one-in-a-thousand low-tail quantile of the utilization rate. To obtain the “effective” volumes, we simply multiply the theoretical purchased volume (given by 2^d where d is the batch depth) by the effective utilization rate.

Finally, in table 4, we provide these volume calibrations for 2^{12} and 2^{16} buckets and log bucket sizes ranging from 4 to 25. Above this bucket size the required quantile already shows a miss rate below 0.1%. That is, there is only a one-in-a-thousand chance that the difference between the effectively utilized volume and the theoretically purchased one exceeds one tenth of a percent which we consider insignificant enough to justify no calibration on the theoretical volume.

log2 no. of buckets	log2 size of buckets	utilisation rate	theoretical volume	effective volume
12	4	0.00000	64.00 K	0.00 B
12	5	0.06747	128.00 K	8.64 K
12	6	0.34060	256.00 K	87.19 K
12	7	0.53374	512.00 K	273.27 K
12	8	0.67030	1.00 M	686.39 K
12	9	0.76687	2.00 M	1.53 M
12	10	0.83515	4.00 M	3.34 M
12	11	0.88343	8.00 M	7.07 M
12	12	0.91758	16.00 M	14.68 M
12	13	0.94172	32.00 M	30.13 M
12	14	0.95879	64.00 M	61.36 M
12	15	0.97086	128.00 M	124.27 M
12	16	0.97939	256.00 M	250.72 M
12	17	0.98543	512.00 M	504.54 M
12	18	0.98970	1.00 G	1013.45 M
12	19	0.99271	2.00 G	1.99 G
12	20	0.99485	4.00 G	3.98 G
12	21	0.99636	8.00 G	7.97 G
12	22	0.99742	16.00 G	15.96 G
12	23	0.99818	32.00 G	31.94 G
12	24	0.99871	64.00 G	63.92 G
12	25	0.99909	128.00 G	127.88 G
16	4	0.00000	1.00 M	0.00 B
16	5	0.00000	2.00 M	0.00 B
16	6	0.28669	4.00 M	1.15 M
16	7	0.49561	8.00 M	3.96 M
16	8	0.64334	16.00 M	10.29 M
16	9	0.74781	32.00 M	23.93 M
16	10	0.82167	64.00 M	52.59 M
16	11	0.87390	128.00 M	111.86 M
16	12	0.91084	256.00 M	233.17 M
16	13	0.93695	512.00 M	479.72 M
16	14	0.95542	1.00 G	978.35 M
16	15	0.96848	2.00 G	1.94 G
16	16	0.97771	4.00 G	3.91 G
16	17	0.98424	8.00 G	7.87 G
16	18	0.98885	16.00 G	15.82 G
16	19	0.99212	32.00 G	31.75 G
16	20	0.99443	64.00 G	63.64 G
16	21	0.99606	128.00 G	127.50 G
16	22	0.99721	256.00 G	255.29 G
16	23	0.99803	512.00 G	510.99 G
16	24	0.99861	1.00 T	1022.57 G
16	25	0.99901	2.00 T	2.00 T

Table 4: Theoretical and effective volumes of postage batches