A Consensus Protocol for Decentralized Data Delivery and Storage

Siddarth Banerjee
Kandola Network
sb@kandola.network

Narayanan Ramanathan
Kandola Network
nara@kandola.network

Ragul Kumar
Kandola Network
gul@kandola.network

***Abstract*—The biggest challenge faced in building decentralized platforms for general purpose data delivery and storage is scale. The platform needs to support very high throughput and support low latency, while not compromising on the security and promises of decentralization. PoRT protocol comprises a light weight data transaction verification function, adopts cryptographic primitives and game theoretic constructs in securing the protocol from adaptive attacks and deploys subnets created from verifiable randomness in data verification and decentralization storage. Parallel chains, epoch driven leader identification for chains, batch derivatives, two-way cryptographic sortition, dynamic subnets for batch verification, static subnets for decentralized storage are some of the key innovations made in building the protocol. Guaranteeing safety and liveness being the core requirement of any consensus protocol, PoRT protocol makes every attempt within its purview to guarantee the two, with the assumption of partial synchrony in network settings. A testnet platform that runs PoRT protocol serving decentralized data delivery and storage is currently under development and will soon be released.**

***Index Terms*—Blockchain, decentralized storage, parallel chains, cryptographic sortition,**

I. Introduction and Motivation

The Web3 vision encompasses building a truly democratized internet that is built on decentralized infrastructure and decision making, that empowers individuals by giving them greater control over their data, identities and digital assets and imparts full transparency and accountability to all aspects of service rendering. Realization of such a vision involves developing holistic protocols that simultaneously encompass decentralized compute, decentralized storage and consensus gathering.

The impracticality behind on-chain data storage with state machine replication across the entire network for general purpose data has resulted in decentralized storage platforms storing transaction hashes on chain and offloading the actual payload to centralized platforms. Such solutions open the doors to the same set of problems that are witnessed in centralized storage such as data censorship, lack of data privacy and security breaches. Platforms built for fully decentralized data storage and delivery can unlock the hidden potentials behind decentralized data marketplaces and give rise to far more decentralized data driven applications.

This paper introduces Proof of Real-time Transfer (PoRT), a consensus protocol that facilitates the decentralized delivery and storage of general purpose data. With data encryption at source, end-point authentication, standardized (and yet customizable) application-driven message schemas and decentralized verification of data compliance, PoRT protocol guarantees the security, privacy and integrity of schema-compliant data all through its life-cycle on the platform.

The security guarantees of the protocol stem from cryptographic primitives and game theoretic constructs that are designed to defend the protocol from various Byzantine attacks. To facilitate high throughput, the protocol dedicates subnets that are identified using verifiable randomness to the tasks of batch verification and data storage and thereby placing implicit bounds on the network's finite resources (compute, memory, network bandwidth) that are dedicated for the tasks. A significant proponent of the protocol's high throughput is a *no-fork* design that PoRT adopts. Through parallel chains that are each dedicated for an application and deterministic leader assignments per chain for preset epoch periods, PoRT achieves deterministic finality. The design choices highlighted above have been carefully made such that the trustless nature of the decentralized platform powered by the proposed consensus is upheld.

There are two key properties of the proposed protocol, namely (a) *Safety*: The nodes of the network agree upon the sequence of verified blocks. It is possible that at some point in time, nodes do not have identical states to their respective chains, but the chain maintained by the lagging node is a subset of that maintained by the more current node (b) *Liveness*: Newer valid transactions will continue to be added to chain within a finite amount of time.

II. Prior Art

Designing decentralized consensus protocols has entailed developing scalable, Byzantine Fault Tolerant protocols that guarantee safety and liveness at all times, while keeping centralization at bay. Network timing assumptions play a critical role in shaping of such protocols. HoneyBadger BFT [2], HotStuff [1], and ICCP [3] are some of the popular consensus protocols designed to achieve high security and performance in distributed systems. While HoneyBadger BFT is suited for asynchronous networks, HotStuff and ICCP are designed for partially synchronous network settings. [4] proposes two atomic broadcast protocols named *Dumbo1* and *Dumbo2* that have asymptotically and practically better efficiency than HoneyBadger BFT.

State Machine Replication in the presence of Byzantine systems has generally entailed full replication of data across the entire network, thereby offering no scaling efficiency. While sharding solutions have been adopted for improved scale, they have been susceptible to adaptive adversarial attack. Free2Shard [5] proposes an adversary-resistant solution architecture for sharding based consensus protocols. Instachain [6] adopts stateless blockchain model in sharding and achieves high scalability while preserving safety, but trading off liveness momentarily.

Filecoin [11], Storj [13], Sia [14], Arweave [12] are some of popularly received decentralized storage solutions. [10] provides a very interesting overview on the state of popular decentralized storage systems today and evaluates them taking into consideration a set of important service requirements for such systems.

III. Proof of Real-time Transfer Consensus

A. Formalism

We will begin with a handful of definitions, relevant notations, introducing the cryptographic primitives that PoRT adopts, ones that are referred to often across this paper.

1) Let N be a set of n nodes $\{N_1, N_2, ..., N_n\}$. Nodes can be of type *full-node* or *storage-node*. A full-node runs the consensus protocol and plays a role in decentralized storage. The storage node plays a role only in the latter.
2) Let $S = \{s_1, s_2, ..., s_n\}$ correspond to the stakes of the respective nodes, the amount of coins or tokens that each node has invested as a collateral to participate in the network.
3) Let the number of Byzantine nodes in the network be f. PoRT protocol is designed for a composition of honest nodes and Byzantine nodes under which $n \geq 3f + 1$.
4) Let A be a set of a applications $\{A_1, A_2, ..., A_a\}$. Applications are data producers and / or consumers of data.
5) Let notation $[x]$ denote the set $\{1, 2, ..., x\}$.
6) Let H be a collision resistant hash function (SHA256).
7) Let DSA_KeyGen, DSA_Sign and DSA_Verify be the digital signature algorithms for the generation of key pairs, signing of messages and verification of signatures respectively.
8) Nodes and applications (henceforth referred to as entities) generate their own public-private key (pk_i, sk_i) using KeyGen, where $i \in [n + a]$. Their decentralized identity (DID) $id_i = \mathrm{H}(pk_i)$.
9) Let R be the Bootstrap registry. Every entity registers itself with R by submitting a DID document that comprises their public key(s), DID(s), service endpoints and other DID related metadata.
10) Given a message m and N_i's private key sk_i, signature S is generated as $S = \text{DSA\_Sign}(m, sk_i)$ and the signature is verified as $\text{DSA\_Verify}(S, pk_i)$.
11) Let Dx and Dh correspond to the XOR distance and hamming distance functions respectively. Let a and b be two binary strings. The XOR distance between them is $Dx(a, b) = a \oplus b$, where $\oplus$ is the XOR operator. The hamming distance between them is $Dh(a, b) = a \sqcup b$, where $\sqcup$ is the hamming distance operator.
12) Let TS_KeyGen be the distributed key generation function under the context of threshold signature. Let DSA_Sign and DSA_Verify, from above, serve their respective roles under the new context. Let t correspond to the threshold value pertaining to the minimum number of signatures required.
13) Verifiable Random Functions (VRF) are realized in PoRT as follows: Given a public input x, a node N_i generates a random number $y = \mathrm{H}(\text{DSA\_Sign}(x, sk_i))$ and generates a proof π $=\text{DSA\_Sign}(x, sk_i)$. y is a number that only N_i gets to generate from x. If supplied with (x, y, π), other nodes can corroborate that y was derived from x using $\text{DSA\_Verify}(x, y, pk_i)$ and verifying if $y = \mathrm{H}(\pi)$.
14) Let P be a function that takes a 256 bit number r as input and generates a number $\hat{s}$ between (0, 1): $\hat{s} = P(r)$. Let $r_1, r_2, ..., r_{32}$ be ordered set of bytes in r. A byte s is generated from r : $s = r_1 \oplus r_2 \oplus ... \oplus r_{32}$ and is normalized as $\hat{s} = s/256$
15) Let Q be a function that takes three inputs: a 256 bit number r, positive integers p and q and generates p numbers in the range $(0, q)$ by applying bitwise right shift operation (the operator being >>). on r followed by a *modulo* q. Q generates an array of positive numbers $\{m_1, m_2, ..., m_p\}$ where $m_i = (r{>>}i) \bmod q, \; \forall i \in [p]$

B. Decentralized Identity Management

Decentralized Identity (DID) is a globally unique, cryptographically verifiable identifier that is resolvable with high availability. DID provides a framework for auto authentication, supports interoperability and enables entities to manage their own identity and provides a framework for auto authentication. PoRT furnishes the decentralized public key infrastructure (DPKI) and other relevant DID SDKs that enable them to create and manage their own decentralized identities.

The bootstrap registry R is maintained at an identity management blockchain (such as Sovrin, uPort) and a set of nodes from the network are identified to serve as DID nodes serving as the network's liaisons for state maintenance of R and for data retrieval. PoRT protocol identifies the DID nodes using cryptographic sortition (the mechanics of which is explained later). Per protocol, nodes can either connect with the DID nodes or with their peers or correspond directly with the external identity management blockchain in requesting DID data and maintaining local repositories of the R.

C. Network Model

PoRT protocol adopts a scale-free, semi-structured, self-organizing peer-to-peer (P2P) overlay network that is designed for low propagation latency, efficiency in query-routing for resource discovery and resource sharing, resilience to high network churn, while being highly scalable and remaining free from hierarchical organization. The topology of the proposed network and the underlying mechanisms for data storage and retrieval, both draw inspirations from constructs adopted by structured and unstructured P2P overlay networks and optimizations proposed for each (Kademlia [9], Perigee [8]).

1) Network Topology: In structured P2P overlay networks [7], nodes and data objects are organized onto a keyspace, as (key, $value$) pairs. Nodes identify their peers through a notion of *proximity* based on their respective keys. Further, there is a deterministic mapping between data objects and nodes identified based on key proximity. The network adopts Distributed Hash Tables (DHT) as a substrate that holds information on data object location and facilitates efficient resource sharing. Unstructured P2P overlay networks, on the other hand, organize nodes in a flat or hierarchical random graphs. Nodes identity their peers through randomized connectivity, or through a mechanism that serves their mutual interests. Protocols such as flooding, random walks and expanding-ring time-to-live search are typically used for querying content. While the former network's key-based routing is scalable and locates rare data items efficiently, the latter is better suited for networks with high churn and locates highly replicated content with much lesser overhead.

Let the proposed network be represented as an undirected graph $G = (V, E)$ with n nodes $N_i \in V, i \in [n]$, each assigned a unique 256 bit identity id_i (also the key) obtained from hashing their respective public keys. The set of edges $e_{ij} \in E$, where $i \in [n]$, $j \in [n]$, $i \neq j$ is set to 1 when node N_i and N_j are connected and set to 0 otherwise. They are organized into binary-tree with their positions identified with the unique prefixes to their respective identities. Nodes connect with a selected set of peers and maintain a routing table facilitates query-routing and resource localization as illustrated below.

- For each node, the tree is partitioned into successively lower sub-trees based on binary prefixes of identities that did not contain the node. Nodes look for at least one peer from within each sub-tree.
- Distance between two nodes n_i and n_j is defined by the XOR metric [9], $d(n_i, n_j) = id_i \bigoplus id_j$. For each $0 \leq m \leq 255$, a node looks towards maintaining a list of nodes (called $k-buckets$) whose distance is in the range $(2^m, 2^{m+1})$ from itself.
- In addition to keyspace proximity, nodes factor in network heterogeneities and node configurations (geographic distance, bandwidth, compute, memory) that directly impact broadcast latency in identifying an optimal set of peers [8].
- From within each sub-tree the $k - bucket$ lists comprise (node id, port number, IP address, latency, node up-time). The lists within each bucket is sorted based on a weighted combination of latency, node up-time and a recency of interaction factors. Nodes that responded to a request recently are weighed more.

PoRT protocol adopts the remote procedure calls namely, {PING, STORE, FIND_NODE, FIND_VALUE} as defined in Kademlia.

2) Communication assumptions: We assume partially synchronous operational settings, whereby messages sent by honest nodes will get delivered within a certain time δ after the passage of an unknown *global stabilization time*. The protocol handles delays in message transmission by incorporating timeouts and requests for re-transmissions. Time-out events can induce nodes to temporarily update their respective k-buckets until normal operational settings are observed. During instances of unpredictability in network conditions, when time-out events are unusually frequent, nodes resort to temporarily updating their k-buckets until normal operational conditions resume or establish new connections with peers using randomized connectivity and adopt practices from unstructured network settings for resource discovery.

Nodes communicate with one another generally through direct port-based communications and resort to peer-to-peer gossip sublayer for network wide dissemination of messages. PoRT protocol adopts HTTP/3 QUIC protocol as the underlying transport protocol.

D. Standardized Data Schema and Transaction Verification

A data transaction can correspond to any one of (a) transfer of data between two or more applications (type: create or update) (b) transfer of data to self (type: create) (c) data query (type: read) (d) data deletion (type: delete). The services requested can fall under one of three types: (a) delivery and storage (b) storage only (c) data retrieval.

Let D be a set of standardized data schemas $\{d_1, d_2, ..., d_k\}$, where each schema defines an application specific structure of the data payload, delineating the fields that make up the data header and body. For every d_i there exists a unique URI u_i that points to an entry in the bootstrap registry R. While applications are free to adopt an existing data schema or use their own customized data schema, the data fields and formats identified for data conformity should be adhered to for the platform to accept the transactions. Let (pk_p, sk_p) and (pk_c, sk_c) be the key pairs of the producer and consumer respectively. Let $[h||b]$ correspond to the two header and body of the data transaction dt. The hash signature (dt_sign) field corresponds to the following: DSA_Sign(H([h || DSA_Sign(b, pk_c)]), sk_p). Table I illustrates a sample data transaction that adopts the standardized schema for IoT.

1) Data Transaction Verification Function: Let $DT = \{dt_1, dt_2, ...dt_k\}$ be a set of data transactions. The data transaction verification function be V: DT $\rightarrow \{0, 1\}$, with $V(dt) = 1$ for valid transactions and $V(dt) = 0$ otherwise. Source authenticity, signature validity, data integrity and schema compliance are primarily established in data transaction validation. Data transaction validation entails the following: [Verify if H(dt) = DSA_Verify(dt_sign, pk_p) $\wedge$ verify if schema URI is valid $\wedge$ verify raw data schema compliance $\wedge$ verify {nonce, command type} validity $\wedge$ verify producer authentication].

TABLE I
STANDARDIZED DATA SCHEMA

Header	
Application DID	Hash of the application's public key
Transaction ID	A GUID
Producer address	DID URI of the producer
Consumer address	DID URI of the consumer or group or self
Command type	An enum pointing to one of 4 commands: Create (insert), Read (query), Update and Delete
Nonce	An incremental index that signifies the order of data transactions from an application
Schema URI	URI adopted for the data payload
Timestamp	Dispatch timestamp
Hash Signature	Signed hash of the full data transaction
Body	
Raw Data	Actual payload, with the field values encrypted.

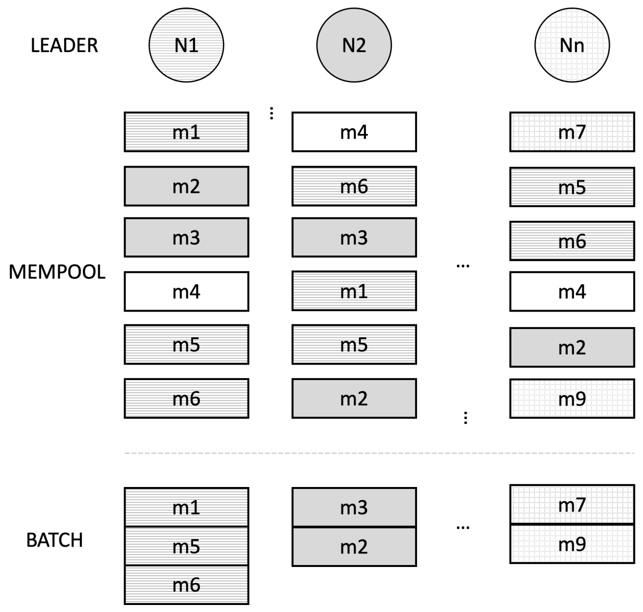

Fig. 1. Messages dispatched by applications making it to mempools across the network. The node designated as leader for a chain, prepares batches comprising of messages sent by the application that is tied to the chain.

E. Mempool data structure and Entropy Source

Mempool is a data structure (Figure 1) that nodes of the network use to store unconfirmed transactions that have been broadcast to the network by applications. Applications send their transactions to one or more nodes of the network and nodes in turn disseminate the transactions across the network use gossip protocol. Network heterogeneities generally result in the transactions making it to the individual mempools in different orders. Transactions reside on mempools until they are batched up for decentralized verification. PoRT protocol provides Network Discovery Service APIs that lets any entity (node, application) to query the status of mempools across the network, to track transaction propagation across mempools and monitor transaction handling latency.

The standardized schemas for transactions implicitly introduces entropy in every transaction that is sent to the network. With fields such as *timestamp*, *nonce*, *transaction ID* being unique and non-repetitive for a given application, a set of transactions sent by an application act as stores of entropy that the network can tap into to derive randomness in the system. The network dissuades duplicate transactions from being sent by levying penalties on the producers for such actions. The network checks for duplicate transactions being sent to the network by periodically comparing the transaction hashes with that of prior transactions. Producers who are repeatedly generating non-compliant transactions face rate limits and face the risk of eventually getting phased out.

Our solution uses batches of transactions as a source of entropy. While at the outset the above design may appear to invite *grinding attacks*, a two-way cryptographic sortition algorithm (explained subsequently) that PoRT incorporates will render such attacks unrewarding.

F. Parallel chains and Leader Election Function

PoRT protocol assigns all validated transactions sent to the network by an application to a single chain that is solely dedicated to that application. This design helps better management of application data ranging from faster responses to data query, efficient audits on transaction handling, until the eventual chain retirement when the application exits the platform. The network shall comprise as many independent chains as there are applications onboarded and hence the name, parallel chains. When an application a_i (with DID: id_i) is first onboarded onto the platform, the following steps are adopted:

- A dedicated chain whose DID is H(id_i) is created across all nodes of the network.
- The application specifies the size of storage required (in GB), region(s) of storage, billing frequency (monthly, bi-monthly, semi-annual, annual) and a replication factor (rf) for its transactions namely, $\{3, 5, 7\}$.

A leader is elected for the newly created chain and is deemed as the only entity who is allowed to propose blocks until a certain *inter-epoch block height* is reached for that chain, after which, a new leader is elected and process continues. An *epoch* is the time duration between two leader elections and *inter-epoch block height* is the number of blocks added to chain during that epoch. Both the leader election and the subsequent block height determination use Verifiable Random Functions. The above design prevents forks and facilitates instant finality, thereby helping the platform's throughput.

Whenever a chain c_x comes up for election, whose last block was b_x a node N_i with DID id_i generates a 256 bit random number r_i as r_i =H(DSA_Sign(H(b_x), sk_i)) which is mapped to number between $(0, 1)$, as $\hat{r_i}$ = P(r_i) (from (14)). A score w_i is computed as the weighted combination of the node's stake s_i and $\hat{r_i}$ in the form $w_i = a * s_i + b * \hat{r_i}$, where a and b are parameters pre-identified by the network, $0 < a < 1$, $0 < b < 1$ and $a + b = 1$. The inter-epoch block height is determined as $bh_i = Q(r_i, 1, max\_blocks)$ (from 15), where max_blocks is the network wide parameter that indicates the maximum number of blocks a leader can propose during one epoch on a chain.

The 256 bit random number r_i is a number that only N_i can generate and is not decipherable to the rest of the network until explicitly shared by the node. N_i can prove to the rest of the network that r_i was generated from H(c_x) by submitting proof π_i, where π_i = DSA_Sign(H($id_i \oplus$ H(c_x)), sk_i), that can be verified by the network upon unsigning and verifying that the message signed was derived from H(c_x). N_i disseminates its scores to the network (as illustrated in Table II through gossip sublayer.

TABLE II
LEADER ELECTION FUNCTION (LEF) REULTS SUBMITTED BY NODE N_i

LEF Header	
Signature	Signed hash of [Header | Result Body]
Chain ID	H(id_i): DID of application a_i's chain
From address	DID URI of Node N_x
LEF Body	
Random number generated	r_i
Proof	DSA_Sign(H($id_i \oplus$ H(c_i))
Inter-epoch block height	bh_i
Stake held	s_i
Cumulative score	w_i

The node that generated the highest w_i will be deemed as the leader for chain c_x, until bh_i new blocks were added to the chain. The index z of the leader is chosen as

$$z = \arg \max_{i \in (1..N)} a * s_i + b * \hat{r}_i \tag{1}$$

But the challenge is that in a partially synchronous network, there is a possibility that some nodes experience longer delays or even fail completely and as a result identifying the absolute maximum score as detailed in the equation above may be infeasible. PoRT protocol adopts a *time-out* period that is long enough to accommodate unforeseen network delays, but short enough to keep the protocol from not stalling, after the passage of which, nodes that deem that they have the highest overall scores declare themselves as the leader. If such a declaration is contested, then the leader is elected with a second round of consensus gathering.

The above design is verifiable. It gives nodes that held higher stakes a better chance of winning the leader election for the given chain, but also introduces uncertainty into leader election by introducing the weighted random number $\hat{r_i}$ in the equation. At a high level, the leader N_z serves the following roles for chain c_x until bh_z blocks are added to chain, namely:

- Aggregating transactions sent by application a_x from its mempool and creating batches of transactions that will be dispatched for verification
- Identifying subnets using *two-way cryptographic sortition* for (i) consensus gathering on the set of valid transactions (ii) subnet driven state machine replication for on-chain storage and CRUD friendly local data stores.

G. Two-way Cryptographic Sortition

Cryptographic sortition is a technique used in distributed systems to randomly select one or more participants from a group of potential candidates in a way that is both unbiased and verifiable. Consider the case when a leader node uses a source of randomness from within the network, and generates a random number that only they have the capacity to generate and repurposes the same to identify a subnet to execute one or more tasks. While it can be proven that the subnet was indeed created from the given random number, there isn't a way to rule out the scenario when the leader employed a grinding attack in identifying the optimal random number that identifies their fellow Byzantine nodes for the subnet.

PoRT protocol requires the individual nodes identified to serve the task(s) to run cryptographic sortition themselves, by generating their own random number that was derived from the *same source of randomness* as that of the leader, and determining for themselves if they are allowed to serve the assigned role or otherwise. In this design, the task assigner's grinding attack could be rendered ineffective as they cannot predict the task receiver's allowed role to serve. The application of cryptographic sortition both by the task assigner and the task receiver is what we call as the *two-way cryptographic sortition*.

H. Batch creation and Verification

Node N_z aggregates the unverified data transactions that were sent by application a_x (whose chain in c_x), from its mempool and creates a batch $B = \{dt_1, dt_2, ...dt_m\}$ comprising an ordered set of m transactions. Batch creation is governed by the most impinging of the three criteria namely: (a) maximum number of data transactions (b) maximum size of the batch (c) maximum wait-time allowed for data transactions.

Using B as a source of entropy, N_z generates a random number r_z =H(DSA_Sign(H(B), sk_z)) and the VRF proof π_z =DSA_Sign(H(B), sk_z). Next, it identifies indices of p nodes who shall be the recipients of batch B by invoking Q(r_z, p, n) (from 15), where n corresponds to the number of nodes in the network. The leader shares (r_z, π_z) with the rest of the network to supply evidence that the indices in p were obtained from r_z.

Batch verification primarily entails running data transaction verification detailed in section III-D1 on the ordered set of transactions in a batch. It is important to establish that a node which, as per protocol, was supposed to run batch verification on a given batch B, performed the job independently and thoroughly without gathering the verification results through out-of-band communications with other fellow nodes who were tasked at verifying the same batch. If such maliciousness was left unchecked, then the Byzantine nodes may gain unfair computational advantage over honest nodes. PoRT protocol adopts a game theoretic construct named *batch derivatives* to such *freeloader attacks* in batch verification.

1) Batch Derivatives: A *batch derivative* B' is obtained by inserting one or more non-conforming transactions called the *control transactions* at random indices onto B. A control transaction is similar to a regular transaction in structure, but is

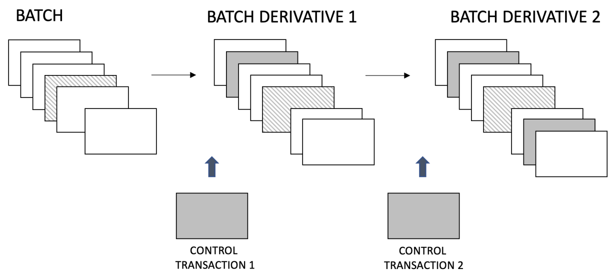

Fig. 2. Control transactions that resemble data transactions sent by applications but are designed to fail data transaction verification are inserted at random indices. The original batch had contained one transaction that would not have passed verification.

designed to fail verification either due to source authentication problems or due to data integrity issues. Let $C = \{c_1, c_2, ...c_y\}$ be the set of y control transactions. $B' = B \odot C$ is a batch derivative constructed upon inserting them at random locations onto B, such that the data transactions in B still appear in the same relative order. The operator $\odot$ signifies control transaction ingestion. Figure 2 illustrates the creation of B'.

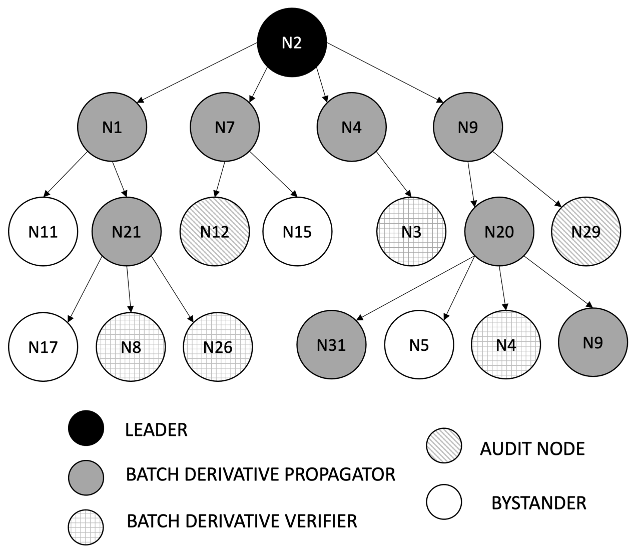

Fig. 3. An illustration of the dynamic subnets for batch verification

2) Two-way Sortition driven Dynamic Subnets: Batch verification is a multi-stage process, the evolution dynamics of which is unpredictable. Each node that received a batch (or a batch derivative) can perform any one of 4 roles in batch verification namely, (a) Verify the received batch and provide verification results to the leader (through direct port-based communication) and the rest of the network (through gossip) (b) Create a batch derivative and propagate the same (c) Perform behavioral audits, monitoring the VRF proofs submitted and the overall adherence to protocol (d) Assume the role of a by-stander, with no real role to play in verifying the batch.

Stage #1 of batch verification proceeds as follows: Every one of the p nodes that leader N_z had identified to take part in verifying batch B, perform the following tasks:

- Step #1: Each node takes (r_z, p_z) submitted by N_z and verifies N_z's source of randomness and if their node index was indeed derived from r_z as prescribed by the protocol. If incongruencies are detected, then the node reports the non-conformity observed.
- Step #2: Nodes in this stage of batch verification assume the role of batch derivative creation and propagation. Each node uses batch B as a source of entropy and generates its own random number $r\prime$ (signing with its own private key) and the associated proof of randomness. It invokes Q($r\prime$, 1, p) and determines the the number of nodes q that is should dispatch batch derivatives to. Next, it invokes Q($r\prime$, q, n), identifies the q recipient node indices, creates different batch derivatives $\{B\prime_1, B\prime_2, ..., B\prime_q\}$ for each recipient by inserting a control message at a random index onto B and dispatches them their respective batch derivatives.

Stage #2 of batch verification proceeds as follows:

- The recipient nodes in the second stage of batch verification, again establish protocol conformity as detailed in Step #1 above.
- Nodes decipher their respective roles based on the random number $r\prime$ that they generate using the same entropy source as the leader did, B. Each node invokes P($r\prime$), which maps $r\prime$ to a probability score $\hat{s}$. Based on $\hat{s}$, the node identifies the role it is allowed to serve:

$$\begin{aligned} 0 \leq \hat{s} < 0.2 &\implies \text{dispatch batch derivatives} \\ 0.2 \leq \hat{s} < 0.6 &\implies \text{verify batch derivative} \\ 0.6 \leq \hat{s} \leq 0.8 &\implies \text{serve as a bystander} \\ 0.8 \leq \hat{s} < 1 &\implies \text{run behavioral audits} \end{aligned}$$

- Batch derivative dispatch is detailed in Step #2 above.
- As a batch derivative verifier, nodes run the data transaction verification function on every transaction in the batch derivative and submit their results (Table III to the network). Nodes deploy compute sharding, thereby identifying segments of the batch that they will run batch verifications for. These segment indices are also determined from the random number that they generated.
- As a bystander, the node serves no role on the received batch derivative.
- As an audit node, the node corroborates source of randomness, VRF proofs and overall adherence to protocol.

Figure 3 illustrates dynamic subnet creation for batch verification. Batch verification runs for one additional (and final) stage. The leader N_z gathers verification results submitted and gathers the list of verified messages in the batch and creates a *block* of verified transactions that shall be added to chain c_x. The above design renders adaptive attacks ineffective and disincentivizes collusion.

TABLE III
SUMMARY FROM BATCH DERIVATIVE VERIFIER

Result Header	
Signature	Signed hash of [Header | Result Body]
Batch ID	Hash(B): Hash of the original message
From address	DID URI of Node N_j
Result Body	
Role	Batch Verifier
Proof π_x	$< h(B), PvK_x >$
Assigner	DID URI of node N_i that assigned the batch derivative
Batch derivative	B' : Batch derivative that was verified
Segments reviewed	1, 3, 4
Verification result	$h(M_1'), h(M_3'), h(M_4')$
Indices of failed messages	Indices where message verification failed

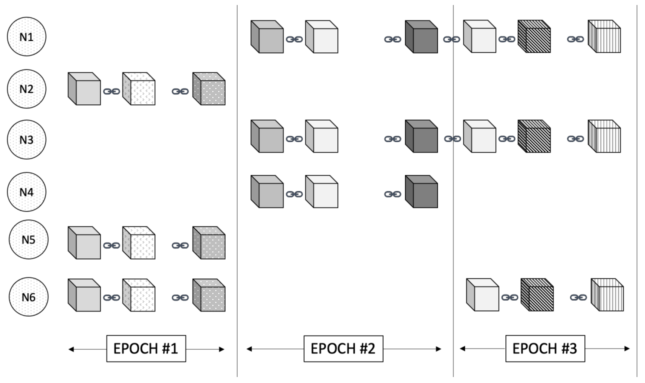

Fig. 4. An illustration of blocks stored in epoch based static storage subnets

I. Decentralized Storage: LogChains and Local Data Stores

A block $\hat{B}$ comprises an ordered set of verified data transactions. The header of the block comprises a Merkle Root, derived from a hash tree (Merkle Tree) of all the ordered data transactions contained in the block. All the transactions in $\hat{B}$ originated from application a_x, are destined to be stored on the application-specific chain c_x, and will have to be replicated a minimum of rf_x times to meet the application's requirements. PoRT's solution towards decentralized storage encompasses the following:

1) State Machine Replication using Subnets: State Machine Replication is executed in a subnet context where two subnets are created, one for chain-based storage (called Log-Chain store (Figure 4) and the other for a CRUD friendly storage (called Data store). The subnets are created using two-way cryptographic sortition with full verifiability. The size of the Log-Chain store subnet is $f+1$ (where in a n node network we have assumed that the number of Byzantines f are such that n 3f + 1). The size of the data store subnet is the same as the replication factor rf_x chosen by the application. The subnet's term period is the same as that of the leader for the chain. After bh_z blocks (called inter-epoch block height) have been added to chain, both the subnets are retired. These are called *epoch based static subnets* for decentralized storage, ones that are commissioned into existence at the beginning of a leader's term and decommissioned from chain storage responsibilities at the end of the leader's term.

2) Epoch based Static Subnets for Storage: The entropy contained in the first block $\hat{B}$ proposed by the leader for the chain during the epoch is as the source of entropy. N_z generates a random number $\hat{r}_z$ =H(DSA_Sign(H($\hat{B}$), sk_z)) and the VRF proof $\hat{\pi}_z$ =DSA_Sign(H($\hat{B}$), sk_z). The leader executes the first step in the two-way cryptographic sortition step. Next, it identifies the indices of $f+1+rf_x$ nodes by invoking Q($\hat{r}_z$, $f+1+rf_x$, n), the first f+1 nodes getting invited to serve Log-Chain storage for chain c_x and the remaining rf_x to serve as the local data store for the same chain for the epoch. The leader supplies $\hat{r}_z$ and $\hat{\pi}_z$ along with the respective invitations.

The recipient nodes establish protocol conformity (as detailed in Step #1 in the previous subsection) whereby they establish that there was no malicious behavior or protocol non-conformity in them getting the invitations to join the epoch based static subnet. The nodes apply cryptographic sortition themselves, generate their own random number $\hat{r}\prime_z$ (derived from $\hat{B}$). If P($\hat{r}\prime_z$) ≥ 0.5, then they accept the invitation. If P($\hat{r}\prime_z$) < 0.5, then the node invokes Q($\hat{r}\prime_z$, 2, n) and identifies new recipient nodes for the invitation. The nodes submit their random numbers and the corresponding proofs to the leader and to the rest of the network.

Once the subnets have been formed, every block dispatched by the leader gets replicated within the LogChain subnet on the respective chains dedicated for application a_x. As a measure of *safety*, a Merkle root is derived, whereby the hash of the blocks added to chain in the current epoch form the leaf nodes of the Merkle tree, every time a new block is added to chain and stored on the respective block headers. The transactions added to the block are simultaneously maintained in local data store preserving the order of the transactions. A data transaction of type *create* results in a new record added to the local data store. Transactions of type *update* result in records getting updated as requested. Transactions of type *read* cause no change to the state of the data store and those of type *delete* get expunged from the data store. Apart from the two forms of storage detailed above, a third network wide chain that leader N_z presides for the epoch duration is maintained that logs the hash of the chain at the end of every epoch, yeilding network-wide *checkpointing* capabilities.

J. Safety, Liveness and Byzantine Defense Mechanisms

Table IV discusses the different Byzantine attack vectors that PoRT protocol defends against. The carefully laid out design choices starting from standardized message schema, network discovery service APIs, batch derivatives, two-way cryptographic sortition and dynamic subnets contribute towards keeping adaptive attacks generally ineffective. The *safety* guarantees of the protocol stem from the *no fork* parallel chain construct that has a leader elected for an epoch duration. Further, dynamic audits and protocol governed checkpointing in the form of Merkle roots derived for every newly added block to the respective chains further strengthen the safety claims of PoRT protocol.

For *liveness* guarantees, PoRT protocol adopts the following measures: if a node that was inducted into either the dynamic subnets for batch verification or the static subnets for decentralized storage crashes or simply becomes unresponsive, to keep the protocol moving, the node is replaced by its nearest node (applying XOR distance on the hash of the node's DID) in the Kademlia keyspace. The same applies to the leader. If the leader goes down or becomes unresponsive or displays malicious behavior, nodes agree to initiate a lighter version of the *view change* protocol. As against the three phases of the protocol (preparation phase, proposal phase and acceptance phase), just the preparation phase is initiated and nodes need to come to a consensus that the primary leader is non-reachable. Then, in deterministic fashion, the node that was closest to the leader node in the keyspace is appointed to take up the leader's responsibilities.

There could be instances when the two-way sortition driven static subnet creation process could run into far more rounds than usual, either due to the distribution of random numbers generated by the potential candidates or due to unfavorable network conditions. PoRT employs a *time-out* duration after which the candidates for unfilled spots in the subnet are identified using deterministic methods such as keyspace proximity to leader. These measures are integrated to preserve *liveness*.

IV. Discussions and Future Work

Service guarantees required off of decentralized storage platforms span persistence in data replication to handle instances of high network churn, efficient storage tracking, generating proofs of data storage and guaranteeing data availability across time and efficient data retrieval upon query [10]. PoRT protocol is built to deliver on each of the aforementioned operational requirements for decentralized data storage. Developing a tokenomics model that results in a positive sum game for every actor in the platform, building TestNet and running performance benchmarks are the key next steps in furthering PoRT protocol. We envision the protocol powering decentralized platforms that will support decentralized data marketplaces, one that can unlock the immense potentials hidden in scattered, thus far untapped data streams.

TABLE IV
PoRT protocol's defense against attack vectors

Attack vector	**PoRT Defense**
Replay attack: Producer burdens the network with duplicate, messages undermining the source of entropy and stressing the network	Nodes run hash-collision test on periodically, on incoming messages by comparing their hash with that of mempool messages
Sybil attacks: A malicious node operates under multiple identities simultaneously	In a stake based network with role fluidity, high collateral needed for Sybil attack is less rewarding
Censorship attack: The leader for the chain withholds messages from certain producers, not including them in a batch	Network Discovery Service API's let any node run queries off of leader's mempool and detect instances of withheld messages.
Batch incongruence: The leader or batch derivative propagators may add or remove or shuffle native messages and delay consensus	Audit nodes check for the hashes of batch derivatives by expunging control messages and trace origins of batch incongruence
Freeloadership: Fellow Byzantine nodes may establish out-of-band communication and share answer keys saving on compute	Dynamic subnets introduce role unpredictability rendering such collusion strategies less viable and less effective.
Grinding attacks: A leader could manipulate blocks such that their re-election and the selection of their fellow Byzantines is favored	Two-way sortition in conjunction with batch derivatives makes network processes less predictable and hence more secure
Adaptive attacks: Malicious nodes may stage coordinated attacks (DDoS) on certain nodes affecting their ability to serve a role and delaying consensus gathering	Dynamically evolving subnets diminish success of coordinated attacks as allowed roles to serve are non-decipherable to anyone but the nodes themselves
Non-conformity in roles: Malicious nodes may perform operations that are beyond their scope	Audit nodes periodically verify role conformity. Non-conforming nodes could loose stake.

References

[1] Yin, Maofan, Dahlia Malkhi, Michael K. Reiter, Guy Golan-Gueta and Ittai Abraham. “HotStuff: BFT Consensus with Linearity and Responsiveness.” Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing (2019): n. pag.

[2] Miller, Andrew K., Yuchong Xia, Kyle Croman, Elaine Shi and Dawn Xiaodong Song. “The Honey Badger of BFT Protocols.” Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (2016): n. pag.

[3] Camenisch, Jan, Manu Drijvers and Timo Hanke. “Internet Computer Consensus.” Proceedings of the 2022 ACM Symposium on Principles of Distributed Computing (2022): n. pag.

[4] Guo, Bingyong, Zhenliang Lu, Qiang Tang, Jing Xu and Zhenfeng Zhang. “Dumbo: Faster Asynchronous BFT Protocols.” Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security (2020): n. pag.

[5] Rana, Ranvir, Sreeram Kannan, DavidN C. Tse and Pramod Viswanath. “Free2Shard.” Proceedings of the ACM on Measurement and Analysis of Computing Systems 6 (2022): 1 - 38.

[6] Ozdayi, Mustafa Safa, Yue Guo and Mahdi Zamani. “Instachain: Breaking the Sharding Limits via Adjustable Quorums.” IACR Cryptol. ePrint Arch. 2022 (2022): 413.

[7] Lua, Eng Keong, Jon A. Crowcroft, Marcelo Rita Pias, Ravi Sharma and Steven Lim. “A survey and comparison of peer-to-peer overlay network schemes.” IEEE Communications Surveys & Tutorials 7 (2005): 72-93.

[8] Mao, Yifan, Soubhik Deb, Shaileshh Bojja Venkatakrishnan, Sreeram Kannan and Kannan Srinivasan. “Perigee: Efficient Peer-to-Peer Network Design for Blockchains.” Proceedings of the 39th Symposium on Principles of Distributed Computing (2020):

[9] Maymounkov, Petar and David Mazières. “Kademlia: A Peer-to-Peer Information System Based on the XOR Metric.” International Workshop on Peer-to-Peer Systems (2002).

[10] @0xPhilillan and @FundamentalLabs, “Decentralized Storage: A Pillar of Web3”, June 2022.

[11] Protocol Labs, ”Filecoin: A Decentralized Storage Network”, 2017.

[12] Williams, Sam A., Viktor Diordiiev and Lev Berman. “Arweave: A Protocol for Economically Sustainable Information Permanence.” (2019).

[13] Wilkinson, Shawn. “Storj A Peer-to-Peer Cloud Storage Network.” (2014).

[14] Vorick, David. “Simple Decentralized Storage.” (2014).