

Waterfall: A Scalable Distributed Ledger Technology

Sergii Grybniak
Institute of Computer Systems
Odesa Polytechnic State University
Odesa, Ukraine
s.s.grybniak@op.edu.ua

Dmytro Dmytryshyn
Institute of Computer Systems
Odesa Polytechnic State University
Odesa, Ukraine
dmitrishin@op.edu.ua

Yevhen Leonchyk
Faculty of Mathematics, Physics and
Information Technologies
Odesa I.I. Mechnikov National University
Odesa, Ukraine
leonchyk@onu.edu.ua

Igor Mazurok
Faculty of Mathematics, Physics and
Information Technologies
Odesa I.I. Mechnikov National University
Odesa, Ukraine
mazurok@onu.edu.ua

Oleksandr Nashyvan
Institute of Computer Systems
Odesa Polytechnic State University
Odesa, Ukraine
o.nashyvan@op.edu.ua

Ruslan Shanin
Faculty of Mathematics, Physics and
Information Technologies
Odesa I.I. Mechnikov National University
Odesa, Ukraine
ruslanshanin@onu.edu.ua

Abstract — Waterfall is a high-scalable smart contract platform for the development of decentralized applications and financial services. The distributed protocol is based on Directed Acyclic Graphs (DAGs) with fast finality Proof-of-Stake (PoS) consensus. The Waterfall platform consists of the Coordinating and Sharding networks achieving high transaction throughput due to the parallelized block production since the DAG structure facilitates scalability which is one of the main challenges of decentralized technologies. The Coordinating network maintains the register of validators as well as assigns block producers, committee members, and leaders in each time slot. In addition, the linearization and finalization of the distributed ledger are performed in the Coordinating network increasing overall security and synchronization.

Keywords — *blockchain, blockdag, directed acyclic graph, distributed protocol, consensus.*

I. INTRODUCTION

This paper provides an overview of the Waterfall Protocol, the problems it addresses, its mechanisms of operation, and future plans for the project.

Our PoS model is based on epochs and committees to the proposed DAG-based protocol. The ideology and rationale behind the rewards and security assumptions of the system intersect with committee based consensus [1-3]. However, our implementation has higher transaction throughput, and as a result, increased system scalability parameters.

Our block referencing mechanism is similar to confirming transactions via recursive elections [4]. Our approach to transaction processing, block production and data propagation across the network allows us to reduce the equipment and network requirements for node clients. Also, the Waterfall protocol introduces finality for transactions. We utilize the RLPx transport protocol to communicate encrypted messages among nodes [5]. In addition, we suggest the implementation of shards and subnetworks [3]. This approach may allow us to further improve the scalability of the system.

Implementation of a Coordinating Network [6] allows for increased deterministic parameters for block ordering.

We propose the inclusion of certain *standard operations in the core of the proposed system.* Such implementation may

lower the entry barriers for users of standard procedures for particular use cases, thereby increasing the system's usability parameters

Ordering and conflict resolution solution was inspired by PHANTOM GHOSTDAG [7] but is a completely original development from our team.. We plan to implement a multiple tier node system that allows various devices to join the system. This will positively influence the system's decentralization characteristics.

II. CURRENT INDUSTRY PROBLEMS AND PROPOSED SOLUTIONS

A. Low performance

System-scalable DAG-based block structures enable the simultaneous publication of multiple blocks. This forms a DAG and achieves finality for all transactions, provided the blocks do not conflict with one another. This approach increases the system's performance.

B. High transaction fees

As long as the entire system's performance is expandable, transaction fees are not expected to rise as the system scales [8]. More blocks will be published simultaneously, and transaction fees are not expected to grow significantly for transactions to be included in blocks from the pool.

C. Cross chain interoperability

Eventually, widely adopted blockchain protocols will be required to achieve cross chain interoperability [9, 10]. The introduction of a scalable cross chain protocol will allow for the moving of assets along chains, performing some additional computational operations in the process.

At the time of writing, there exist solutions on the market that allow for the creation of NFTs, but there is no adopted solution that allows them to be created with reasonable creation fees and to be simultaneously transferable to other NFT networks. Current networks take a maximalist approach, striving to create NFTs that stay only within their own network, and bridges are not a priority.

Waterfall could be connected to popular protocols via two-way bridges, allowing for creation and movement between protocols. This can be achieved with lower cost and

higher speed due to the protocol's scalability compared to currently available industry protocols [8, 11].

D. Scalability = Centralization

Many current systems that are proposing solutions to overcome the scalability limitations of distributed systems suggest a tradeoff with decentralization. Our approach intends to overcome the limitations of scalability, preserving decentralization.

III. WATERFALL PROTOCOL MECHANISMS

This section provides an overview of the mechanisms by which Waterfall delivers superior scalability without compromising decentralization.

A. Functional Subsystems

The main technical structural element of the Waterfall network is a Node – a server registered in the network, which stores all the relevant records – Ledgers. On each Node we can deploy a certain number of logical structural elements – Workers, whose accounts have the necessary credentials to participate in the PoS protocol Stake. Each Worker consists of two components with independent addresses – Validator and Coordinator.

The timeline of the network is divided into separate slots - time slots during which actions are considered simultaneous. Validators selected in each subnet of each shard must create and distribute their block during the slot. Slots are combined into epochs. Epochs are intended to summarize the intermediate results of the network, assigning committees of validators to each subsequent slot of the next epoch. Several epochs can be combined into epochs, within which the most coordinated changes of the network's settings values take place democratically. Changes in the settings are aimed at increasing stability, increasing productivity and increasing economic efficiency of the network.

There are two functional subsystems implemented for the system functioning (Fig. 1). Independent shard networks accept transactions, combine them into blocks and make reference DAG. The blockchain-based coordinating network is responsible for DAG linearization, block finalization, and selection of validators creating blocks in a particular time slot.

Coordinating Network functions:

- unite all the shards informationally and provide unified management;
- synchronize shards by time (eras, epochs and slots);
- maintain registers of validators;
- assign roles to workers;
- receive information from shards about skeleton blocks and vote for them, while creating a block in your network;
- pass consensus information to the shards for finalization.

Shard Network features:

- retrieve information about an era, era, slot and worker roles from the Coordinating Network;
- distribute transactions;
- create and distribute new blocks;

- store the state of the network;
- transfer skeleton blocks to the Coordinating Network;
- receive instructions from the Coordinating Network to finalize blocks.

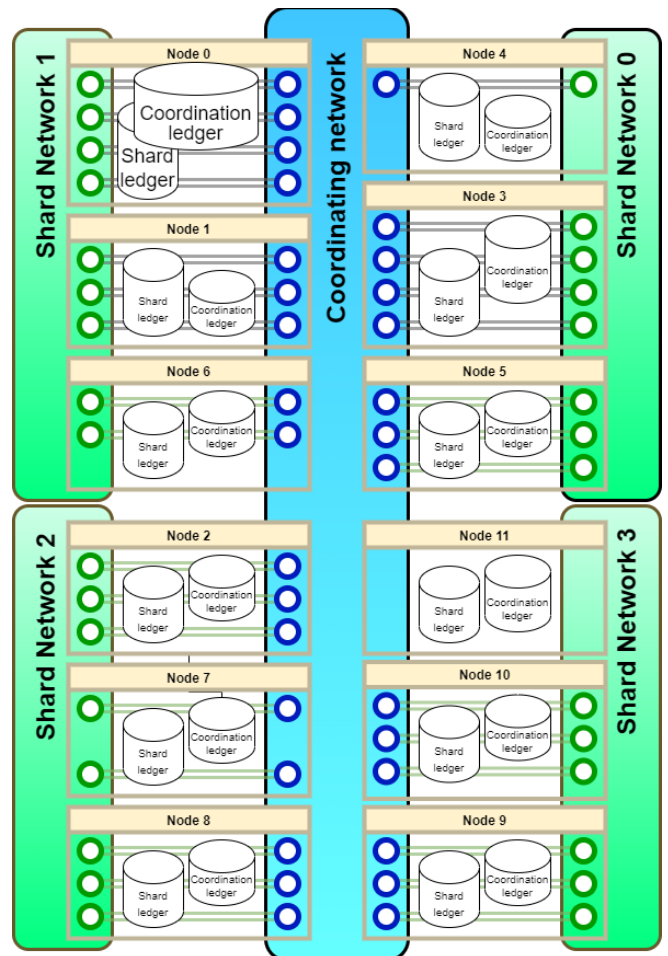


Fig. 1. The Coordinating and Shard networks.

As a result of the networks working together, the created blocks of transactions line up in a DAG, part of which is topologically sorted and finalized (Stream). The second part, consisting of not yet ordered blocks, forms Spray (Fig. 2).

The width of the Spray area is determined by the number of subnets currently working in the shard. Within each fixed time slot, called a slot, a subnet has the right to create one block. The block is created by a single validator node in each subnet selected in the coordination network.

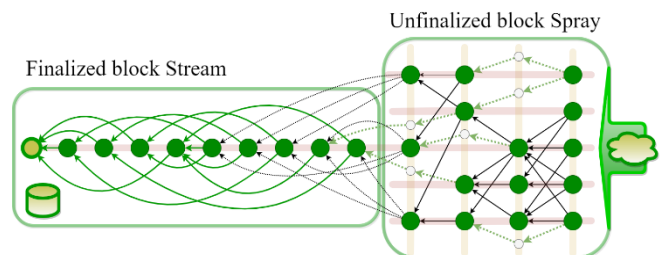


Fig. 2. The linearization of the distributed ledger.

Two strategies for creating subnets are currently considered. The probabilistic strategy provides better security features and is based on the distribution of transactions between nodes, based on the sender's public key hashcode.

The second strategy provides a higher speed of transaction processing and a smaller load on the data network by combining nodes with better message transmission characteristics between them (lower latency and higher speed of information transmission) into a subnet.

The allowed depth of links (a system calculation parameter) sets the depth of the Spray area (Fig. 3). When a new block is created, the block signer writes into it references to some blocks from previous slots, called tips. Tips-blocks are the blocks that are not referenced by any other blocks known to the block signer at the moment.

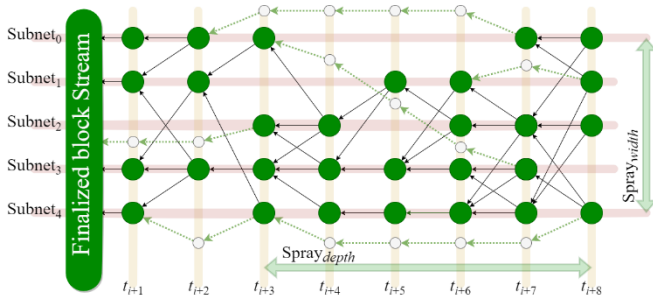


Fig. 3. The Spray area.

B. Messaging Between BlockDAG Nodes

Waterfall uses a protocol based on the RLPx transport protocol for information exchange between nodes. Once the connection is established, nodes are exchanged with their status, and only after that are the nodes able to receive and send additional messages. Message status has the following structure:

Status (0x00)

```
[
  ProtocolVersion P // current protocol version
  NetworkID P // network id
  LastFinNr P // last finalized number
  // array of hashes of not finalized dag-chain blocks
  Dag [hash1: B_32, hash2: B_32, ...]
  Genesis B_32 // hash of genesis
]
```

C. Full Synchronization

After the exchange of node statuses, the nodes go into sync mode if they do not match:

- LastFinNr;
- Dag.

A node with a smaller LastFinNr starts synchronizing all blocks up to the larger LastFinNr, after the LastFinNr of the nodes matches, the nodes start exchanging blocks whose hashes are listed in Dag. Synchronization will occur until the node synchronizes with all the nodes to which it has a connection. The following messages are used to organize the synchronization:

GetBlockHeaders (0x03) - request header for the specified hashes

```
[
  request-id: P,
  [
    startblock: {P, B_32},
    limit: P,
    skip: P,
```

```
reverse: {0, 1}]
```

```
]
```

BlockHeaders (0x04) - response for the previous request

```
[
  request-id: P,
  [header1, header2, ...] // headers
]
```

where header:

```
[
  // address of the validator that created the block
  coinbase: B_20,
  // hashes of parent blocks
  parent-hashes: [blockhash1: B_32, blockhash2: B_32, ...],
  state-root: B_32, // hash of root node of prefix tree state
  state-block-hash: B_32, // final block hash for which state
  root is calculated
  txs-root: B_32, // hash of root node of prefix tree,
  containing all transactions named in this block
  receipts-root: B_32, // hash of root node of prefix tree,
  which contains information regarding payment for all
  transactions named in this block
  bloom: B_256, // Bloom filter (data structure), containing
  information from journals
  height: P, // chain length
  gas-limit: P, // gas limit for block
  gas-used: P, // how much gas was used (transaction sum)
  time: P, // block creation time
  extradata: B // additional information about the block e.g.
  information about the client that created the block
  epoch P, // epoch when the block was created
  slot B_8 // slot number where the block was created
]
```

GetBlockBodies (0x05) – block content request

```
[
  request-id: P,
  [blockhash1: B_32, blockhash2: B_32, ...] // block hashes
  used to request the content
]
```

BlockBodies (0x06) – response to the previous request

```
[
  request-id: P,
  [block-body1, block-body2, ...] // block content
]
```

where block-body:

```
[
  // transactions that joined the blocks
  transactions: [txhash1: B_32, txhash2: B_32, ...]
]
```

GetTransactions (0x09) - request for transactions

```
[
  request-id: P,
  // transaction hashes used to request the information
  [txhash1: B_32, txhash2: B_32, ...]
]
```

Transactions (0x0a) - response to the previous request

```
[
  request-id: P,
  [tx1, tx2...] // transaction content
]
```

where tx:

```
[
  nonce: P, // number of transactions sent by a sender
  // amount of Wei the sender is ready to pay for
  // the gas unit required to complete the deal:
  gas-price: P,
  // maximum amount of gas the sender is ready
  // to pay for the transaction:
  gas-limit: P,
  recipient: {B_0, B_20}, // recipient address
  // amount of Wei that will be transferred from
  // sender to recipient:
  value: P,
  // input data (parameters) for calling a message
  // (used for smart contracts):
  data: B,
  // designation data, used to create a signature
  // that identifies the transaction sender:
  V: P,
  // designation data, used to create a signature
  // that identifies the transaction sender:
  R: P,
  // designation data, used to create a signature create
  // that identifies the transaction sender:
  S: P,
]
```

GetDag (0x11) - request for DAG part

```
[
  request-id: P,
  from-fin-nr P // last finalizer block number
]
```

Dag (0x12)

```
[
  request-id: P,
  [daghash1: B_32, daghash2: B_32, ...] // block hashes used to
  request the content
]
```

After all the blocks and transactions are uploaded using the algorithm described below, blocks create a chain, and transactions are executed consistently along this chain.

D. Block Propagation

Recently created blocks must be retranslated to all nodes immediately. This is done by a 2-step block expanding process:

- When the message “NewBlock” is received from a peer-to-peer node, the client validates the basic block header, signature, epoch and slot affiliation. It then sends the block from the smallest part of connected peer-to-peer nodes (usually the square root of the total number of peer-to-peer nodes) using the message “NewBlock.” After header validation, the client imports the block to its local chain, executing a merge operation (described in detail below). The root hash of the block state should match the calculated root of the state of the finalizing block specified in that block.

- After the block is completely processed and considered to be valid, the client sends the message “NewBlockHashes” to all peer-to-peer nodes, notifying them of the new block. These peer-to-peer nodes can request the entire block later via the message “NewBlock” if they are unable to get it elsewhere.

The node should not send the message about the new block back to the peer-to-peer node that first introduced it. This is usually accomplished by remembering the large number of block hashes that were recently transferred to each peer-to-peer node. Acceptance of block messages can also run a chain synchronization, provided the block is not a direct successor of the client’s most recent block.

NewBlock – new block information

```
[
  block, // block information
]
```

where block

```
[
  header, // block header, see above
  transactions: // transactions entered to the block
]
```

NewBlockHashes – new blocks hashes information

```
[
  blockhash1: B_32, blockhash2: B_32, ... // new block hashes
]
```

E. Transaction Exchange

All nodes exchange pending transactions, to be transferred to validators who will select them for block inclusion. The clients’ realizations track the list of pending transactions in a “transaction pool.”

When a new peer-to-peer connection is set up, transaction pools should be synchronized on both ends. To begin the exchange, both ends should send the message “NewPooledTransactionHashes,” which contains all the transaction hashes in a local transaction pool.

NewPooledTransactionHashes – current transaction pool

```
[
  txhash1: B_32, txhash2: B_32,
  ...
]
```

When the client receives the message “NewPooledTransactionHashes”, it filters the received set, collecting any transaction hashes it does not yet have in its local pool. It can then request the transactions using the message, “GetPooledTransactions”.

GetPooledTransactions – missing transactions information request

```
[
  request-id: P,
  // hashes of the requested transactions:
  [txhash1: B_32, txhash2: B_32, ...]
]
```

PooledTransactions

```
[
  request-id: P,
```

```
[tx1, tx2...] // transaction information, the format is described above  
]
```

When new transactions appear in a client's pool, it should distribute them to the network with the transaction messages described above, and "NewPooledTransactionHashes." This sends information that a transaction has been added to the pool.

NewPooledTransactionHashes – information that a transaction was added to the pool

```
[txhash1: B_32, txhash2: B_32, ...]
```

The transaction message retranslates the objects of the entire transaction and is usually sent to a small random part of connected peer-to-peer nodes. All other peer-to-peer nodes receive a notification about the transaction hash and can request the entire transaction object.

Whole transaction distribution among peer-to-peer nodes usually guarantees that all nodes receive the transaction and will not have to request it. The node should never send a transaction to a partner that is already aware of it (either because it was previously sent or was initially informed by the partner). This is usually achieved by remembering the set of transaction hashes that were most recently transferred by a peer-to-peer node.

IV. ACCOUNTS AND SYSTEM

A. Account State

The state of each account, regardless of its type, can have one of four values:

- *Nonce*. If the real account matches an external account, then the received number is the number of transactions that were sent from this account address. If it is a contract account, then the nonce element is the number of contracts created in this account.
- *Balance*. The total amount purchased by the account.
- *Storage root*. Hash of the root node of the prefix Merkle tree (information about Merkle tree provided below). The Merkle tree codes the hash of the account content and is empty by default.
- *Code hash*. Hash of the account's Ethereum Virtual Machine (EVM) code. For contract accounts, this field is a code that is hashed and stored as a code hash.

B. General System State

The global Waterfall state is a match between an account address and the account state. This match is stored in the prefix Merkle tree data structure.

Each block has a header where the root node hash of three different Merkle tree structures are stored, including:

- *prefix tree state* – state-root;
- *prefix tree transactions* – txs-root;
- *payment acceptance pages for a prefix tree* – receipts-root.

The possibility of storing data effectively in a Waterfall prefix Merkle tree is a practical solution for "thin" clients or nodes. BlockDAG support is done with the help of a set of nodes. There are only two types of nodes - thin and full.

The advantage of using the prefix Merkle tree is that the structure's root node depends cryptographically on the data stored in the tree. Therefore, the root node hash can be used as a safe data identifier. The root hash of the trees is included in the block header, along with the block's state, transactions, and payment receipt information. Every node can check any other part of the Waterfall state without having to store all the states, whose size can be unlimited.

V. LIFECYCLES

In the previous sections, we described the main operating parts of our system. In this section, we will describe the main life cycles.

As described above, there are 2 types of nodes that connect to their networks: Coordinating Network and Shard network. To start a node, you need to raise these two types at the same time.

A. Validator life cycle

Sending a Deposit. The Validator makes a deposit by sending a transaction that triggers the deposit contract function in the Shard Network.

During the deposit, the validator transmits:

- *amount* – amount, must be greater than MIN_DEPOSIT_AMOUNT;
- *pubkey* – BLS public key corresponding to the private key that will be used to sign messages in the Coordinating Network;
- *withdrawal_credentials* – information for deposit withdrawal and earnings;
- *signature* – BLS message signature;
- *deposit_data_root* – tree hash to protect against error.

The purpose of having separate signature and withdrawal keys is to ensure that the more vulnerable withdrawal key is stored securely. The signature key is used to actively sign the message for each epoch. The deposit contract stores the Merkle tree root for all deposits made previously.

Deposit processing. Deposits cannot be processed in the Coordinating Network until the work proof block in which they were deposited, or any of their descendants, are added to the Coordinating Network state.shard1_data. Once the necessary BlockDAG block data is added, the deposit is usually added to the Coordinating Network block, and to state.validators for one or two epochs. The validator is then in the activation queue.

Validator index. After the validator is processed and added to the validators Coordinating Network, the validator_index of the validator is determined by the index in the registry in which the ValidatorRecord contains the pubkey specified in the validator deposit. The validator_index of the validator is guaranteed not to change from the initial deposit until the validator exits and withdraws completely. This validator_index is used throughout the specification process to define the roles and responsibilities of the validator at any time, and must be stored locally.

Activation. Once a Sufficient Deposit has been made for a validator and the Coordinating Network has processed it, after about 4 epochs the validator is activated and added to the mix, and begins to receive its roles.

Exit. After a worker sends a request to leave the network, a delay of about 27 hours is introduced before he can withdraw his funds. This period is required in order to:

- ensure that in the case of incorrect worker behavior, there is a period of time in which the error can be detected and the worker can be reduced, even if the exit queue is almost empty;
- it takes time for all the shards to turn on the reward;
- it takes time to regroup the data if the voter is a keeper of sparse data.

If the validator is shortened, an additional delay of ~36 days is imposed on the withdrawal.

Roles. At the beginning of each epoch, the validator must:

- check his/her assignment to the next epoch using the mixing algorithm;
- check the number of validators in the committee;
- calculate the subnet to which it should connect;
- find the peers and connect to the topic.

The paired validator coordinator has the following responsibilities in the Coordinating Network:

- propose a block if it is chosen as a committee leader (happens very rarely);
- sign the block (to participate in the consensus), that is, create an attestation (to be done once per epoch);
- aggregate signatures if selected as an aggregator (so as not to spam the whole network);
- create a block in the Shard Network if given the opportunity.

B. Transaction lifecycle

Any user who has a wallet with a balance on it can make a transfer to another user. To do this through an application (e.g. MetaMask), the user generates and signs a transaction and sends it to their own or a public node of the BlockDAG Network. As described above, the transaction is distributed to all nodes and added to the pools, until the transaction hits the block. Once the block is finalized, the result of the transaction will change the Account State.

C. Life cycle of a block in the Sharding Network

Submit block. As described above, the validator can obtain the right to create a block in the Sharding Network. For this, you must specify in the block itself:

- hash of blocks which have not yet been referenced (tips);
- block height – the number of blocks with their roots in the given one;
- the current Epoch and Slot;
- add transactions;
- calculate the State of the last finalized block and write its (block) hash and Merkle Root State hash.

The following approach is used to prevent duplicates. Each validator has its own position in the list of block creators of the current slot. The transaction enters the block if the remainder after dividing the sender address by the number of

creators equals the position of the validator. The second way to solve the problem of re-inclusion of transactions in blocks is to use subnets.

Block Propagation. Other nodes must check if the block was created correctly. For this, there are the following rules for checking the block:

- correctness of the slot and creation node (could the validator create a block in this slot?);
- whether the contents of the block are correct;
- whether references to previous blocks are correct (no references to multiple blocks that were created by the same node in the same slot; no references to incorrectly created blocks).

After creating a block, a node must distribute this block in the network as quickly as possible so it can be referenced in the next slot, in order to finalize the block faster.

Ordering and Consensus. At the beginning of each slot the Coordinating Network receives from the Sharding Network a list of non-finalized skeleton blocks. A skeleton block is a block that has the same block height as the ordering position. After receiving the list of skeleton blocks, members of the Coordinating Network come to a consensus and share the result with the Sharding Network nodes.

Ordering algorithm. Here we describe the ordering algorithm:

1. The node takes all the blocks that have not yet been referenced (Tips).
2. It takes the block with the highest Height among them. If there are several such blocks, then it takes the block with the smallest Hash; this block will be the latest skeleton block at the moment.
3. We take the parents of the block and check if there is a finalized block. If such a block exists then it is taken. Otherwise a block with the largest Height is taken. If there are several such blocks then the one with the smallest Hash is taken. This block will be the previous skeletal block.
4. The unfinalized past of the remaining parents of the block are ordered according to heights and hashes and are added to the list to the right of the skeletal block. Thus, they will never match the Height with the position in the sorted list.
5. Items 3-4 are repeated until the algorithm reaches the finalized blocks.
6. The result of the algorithm is a sorted list from the finalized blocks part, to the last visible block.

Finality and transactions complete. At the beginning of each slot, a Sharding Network node can receive from the Coordinating Network a list of skeleton blocks to finalize. The node arranges the past of these skeleton blocks with an algorithm similar to the previous one. It performs sequential transactions, updates the State of the system. The node that creates the block in the given slot specifies in the new block the hash of the last finalized block, and the Merkle Root stack that corresponds to the executed transactions, up to and including the given block.

VI. FUTURE WORK

We are considering the release of an intermediary version with a set of known validators who will vote on behalf of the total number of stakeholders. On our way to complete decentralization, we may also consider introducing additional

security measures, similar in part to the Witnesses in [12] and/or the Coordinator in [13].

Future work will include research on privacy implementation, work to optimize the size and distribution of the transaction history, further increasing speed and scalability parameters, post-quantum cryptography, and (most likely) further modification of the consensus algorithm.

One arising threat in the later stages of protocol adoption is the growing size of the transaction history. The faster new blocks are produced, the larger the size of the transaction history. For instance, the proposed implementation of Spectre for Bitcoin at the speed of 1000 transactions per second will require the blockchain to grow by ~100 GB per day [14].

1,000 transactions per second will cause the transaction history to grow by ~9,5 GB per day in our protocol's most recent implementation. According to the most recent intermediary lab tests our protocol was able to process 3,600 transactions per second. Test was conducted on Amazon EC2 (t3a.small). By comparison, Visa processes approximately 1,700 transactions per second [15]. Ethereum is planning to address this issue using Shard Chains [16]. Sharding could be a solution for Waterfall also. Applying blockDAG with sharding could further improve performance by increasing the scalability of each shard, and thus the entire system.

One possible solution is to implement distributed storage for the transaction history. Basically, each node will store a portion of the transaction history instead of storing it in full. We can think of transaction history as a single larger file, or multiple smaller files, distributed across multiple computers.

Peer-to-Peer file sharing systems have been available for 20 years already [17, 18]. We believe that the transaction history could be distributed across nodes, with a group of nodes storing certain parts of the transaction history, and the ability to reconstruct the entire transaction history with a probability of close to 100%, without storing the entire file or groups of multiple files in one place.

In the same way, BitTorrent technology allows for the reconstruction of an entire large file by collecting fragments from multiple sources, without downloading them from a single place. For such implementation, we may need to introduce Archive nodes and switch the roles of full nodes.

Full Nodes will hold portions of the transaction history instead of holding the entire history. A group of full nodes holding different parts of the transaction history will be able to reconstruct the entire transaction history without holding it in one place.

Archive Nodes will be set up on powerful servers in different parts of the world and hold the entire history of transactions. They will serve as a backup source in cases where the entire history takes too long to reconstruct from the full nodes.

Machine-learning-enabled Controller nodes will monitor the nodes' online time and the proper distribution of portions of the transaction history, to achieve a maximum level of availability. This role could be merged with Archive nodes.

With a growing network achieving higher decentralization, if we choose this approach, we anticipate that

the number of requests to the Archive Nodes will diminish over time.

Another way to approach this is to store transactions in IPFS and record the returned. IPFS hash transactions to the block [19]. This and the previous method could both be implemented with sharding and combined with approaches that would be similar in part to [20].

REFERENCES

- [1] A. Kiayias, A. Russell, B. David, and R. Oliynykov, "Ouroboros: A Provably Secure Proof-of-Stake Blockchain Protocol," *Lecture Notes in Computer Science*, pp. 357–388, 2017, doi: 10.1007/978-3-319-63688-7_12.
- [2] H. K. Alper, "BABE," Available: <https://research.web3.foundation/en/latest/polkadot/block-production/Babe.html>.
- [3] V. Buterin, et al., "Combining GHOST and Casper," arXiv:2003.03052 [cs.CR], 2020, Available: <https://arxiv.org/abs/2003.03052>.
- [4] Y. Sompolinsky, Y. Lewenberg, and A. Zohar, "SPECTRE: Serialization of proof-of-work events: confirming transactions via recursive elections," The Hebrew University of Jerusalem, 2017.
- [5] Ethereum.org, "The RLPx Transport Protocol," Available: <https://github.com/ethereum/devp2p/blob/master/rlpx.md>.
- [6] Ethereum.org, "Phase 0 - The Beacon Chain," Available: <https://github.com/ethereum/consensus-specs/blob/dev/specs/phase0/beacon-chain.md>.
- [7] Y. Sompolinsky, S. Wyborski, and A. Zohar. "PHANTOM and GHOSTDAG: A scalable generalization of nakamoto consensus," *Cryptol. ePrint Arch.*, rep. 104, 2018, Available: <https://ia.cr/2018/104>.
- [8] G. A. Pierro and H. Rocha, "The Influence Factors on Ethereum Transaction Fees," 2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB), 2019, pp. 24–31, doi: 10.1109/WETSEB.2019.00010.
- [9] S. Johnson, P. Robinson, and J. Brainard, "Sidechains and interoperability," arXiv:1903.04077 [cs.CR], 2019, Available: <https://arxiv.org/abs/1903.04077>.
- [10] G. Wood, "PolkaDot: vision for a heterogeneous multi-chain framework. Draft 1," 2016, Available: <https://polkadot.network/PolkaDotPaper.pdf>.
- [11] Y. Liu, et al., "Empirical Analysis of EIP-1559: Transaction Fees, Waiting Time, and Consensus Security," arXiv:2201.05574 [econ.GN], 2022, Available: <https://arxiv.org/abs/2201.05574>.
- [12] A. Churyumov, "Byteball: A decentralized system for storage and transfer of value," 2016, Available: <https://byteball.org/Byteball.pdf>.
- [13] S. Popov, et al., "The coordicide," IOTA Foundation, Jan. 2020, Available: https://files.iota.org/papers/20200120_Coordicide_WP.pdf.
- [14] Y. Sompolinsky, "SPECTRE," BPASE'18, Stanford University, Jan. 2018, Available: <https://youtu.be/57DCYtk0IWI>.
- [15] J. A. Johnsen, L. E. Karlsen, and S. S. Birkeland. "Peer-to-peer networking with BitTorrent," Department of Telematics, NTNU, 2005, Available: <https://web.cs.ucla.edu/classes/cs217/05BitTorrent.pdf>.
- [16] Ethereum.org, "Shard Chains," 2022, Available: <https://ethereum.org/en/eth2/shard-chains/>.
- [17] J. Benet, "IPFS – Content Addressed, Versioned, P2P File System," arXiv:1407.3561 [cs.NI], 2014, Available: <https://arxiv.org/abs/1407.3561>.
- [18] Hedera Hashgraph, "HBAR," Available: <https://hedera.com/hbar>. R. Kumar and R. Tripathi, "Implementation of Distributed File Storage and Access Framework using IPFS and Blockchain," Fifth International Conference on Image Information Processing (ICIIP), Nov. 2019, doi: 10.1109/iciip47207.2019.8985677.
- [20] L. Baird, M. Harmon, and P. Madsen, "Hedera: A Public Hashgraph Network & Governing Council," Hedera Hashgraph, Aug. 2020, Available: https://hedera.com/hh_whitepaper_v2.1-20200815.pdf.