WATERFALL: A SCALABLE DISTRIBUTED LEDGER CONSENSUS PROTOCOL

DMYTRYSHYN D., GRYBNIAK S., NASHYVAN O., AND SHANIN R.

Abstract. Waterfall is a highly scalable cross-platform transaction protocol and a smart contract platform based on Directed Acyclic Graphs (DAG). The Waterfall Project plans to implement the DAG-based Proof-of-Stake (PoS) consensus algorithm and to release a protocol for efficient processing of both Fungible and Non-Fungible tokens (NFTs).

Contents

1. Introduction

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

1.1. Waterfall Project Development.

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 [1–3]. However, our implementation has higher transaction throughput, and as a result, increased system scalability parameters.

Our block referencing mechanism is similar to [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]. We suggest the implementation of shards [6]. This approach may allow us to further improve the scalability of the system.

Implementation of a beacon chain [7] 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 are based on [8]. Greedy implementation allows for high processing speeds. 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.

2. Current Industry Problems and Proposed Solutions

— *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.

— *High transaction fees.* As long as the entire system's performance is expandable, transaction fees are not expected to rise as the system scales. 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.

— *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 [11].

— $Scalability = Centralization$. 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.

3. Waterfall Protocol Mechanisms

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

3.1. **Messaging Between 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)

version: P, // current protocol version
networkid: P, // network number 0 — main, 1 — testnet
height: P, // height of the most well-known chain
blockhashes: [blockhash$_1$: B_32, blockhash$_2$: B_32, ...], // blocks without child hashes
genesis: B_32 // genesis hash block

3.2. **Full Synchronization.** After node status has been exchanged, the nodes switch to synchronization mode if they have missing blocks. Blocks and transactions are uploaded recursively until all blocks and transactions are uploaded. The following messages are used for this:

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

request-id: P,
blockhashes: [blockhash$_1$: B_32, blockhash$_2$: B_32, ...] // block hashes used to request the headers

BlockHeaders (0x04) — response for the previous request

request-id: P,
[header$_1$, header$_2$, ...] // headers

Where header:

coinbase: B_20, // address of the validator that created the block
parent-hashes: [blockhash$_1$: B_32, blockhash$_2$: B_32, ...], // hashes of parent blocks
state-root: B_32, // hash of root node of prefix tree state
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,
[blockhash$_1$: B_32, blockhash$_2$: B_32, ...] // block hashes used to request the content

BlockBodies (0x06) — response to the previous request

request-id: P, [block-body$_1$, block-body$_2$, ...] // block content

where block-body:

transaction-hashes: [txhash$_1$: B_32, txhash$_2$: B_32, ...] // hashes of transactions that joined the blocks

GetTransactions (0x09) — request for transactions

request-id: P, [txhash$_1$: B_32, txhash$_2$: B_32, ...] // transaction hashes used to request the information

Transactions (0x0a) — response to the previous request

request-id: P, [tx$_1$, tx$_2$, ...] // transaction content

where tx:

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

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.

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

(1) 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 message state.

(2) 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 parent-hashes: [blockhash$_1$: B_32, blockhash$_2$: B_32, ...], // parent block hashes transaction-hashes: [txhash$_1$: B_32, txhash$_2$: B_32, ...] // hashes of the transactions entered to the block

NewBlockHashes — new blocks hashes information

blockhash$_1$: B_32, blockhash$_2$: B_32, ... // new block hashes

3.4. **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

txhash$_1$: B_32, txhash$_2$: 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, [txhash$_1$: B_32, txhash$_2$: B_32, ...] // hashes of the requested transactions

PooledTransactions

request-id: P, [tx$_1$, tx$_2$, ...] // 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

$txhash_1$: B_32, $txhash_2$: 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.

4. Accounts and System

4.1. **Account State.** The state of each account, regardless of its type, can have one of four values:

(1) *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.
(2) *Balance.* The total amount of wei purchased by the account.
(3) *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.
(4) *Code hash.* Hash of the account's Waterfall Virtual Machine (WVM) code. For contract accounts, this field is a code that is hashed and stored as a code hash.

4.2. **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 2 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.

5. Main Data Structure

5.1. **Blocks.** The block structure consists of two main parts: Block and Header. The Header contains brief information about the block without providing detailed information about the transactions it contains.

```
type Header struct {
    ParentHashes common.HashArray 'json:"ParentHashes" gencodec:"required"'
    Epoch uint64 'json:"epoch" gencodec:"required"'
    Slot uint64 'json:"slot" gencodec:"required"'
    Height uint64 'json:"height" gencodec:"required"'
    Coinbase common.Address 'json:"miner" gencodec:"required"'
    Root common.Hash 'json:"stateRoot" gencodec:"required"'
    TxHash common.Hash 'json:"transactionsRoot" gencodec:"required"'
    ReceiptHash common.Hash 'json:"receiptsRoot" gencodec:"required"'
    Bloom Bloom 'json:"logsBloom" gencodec:"required"'
    GasLimit uint64 'json:"gasLimit" gencodec:"required"'
    GasUsed uint64 'json:"gasUsed" gencodec:"required"'
    Time uint64 'json:"timestamp" gencodec:"required"'
    Extra []byte 'json:"extraData" gencodec:"required"'
    BaseFee *big.Int 'json:"baseFeePerGas" rlp:"optional"'
}
type Block struct {
    header *Header
    parents []common.Hash
    transactions Transactions
    // caches
    hash atomic.Value
    size atomic.Value
    // These fields are used by package eth to track
    // inter-peer block relay.
    ReceivedAt time.Time
    ReceivedFrom interface
}
```

5.2. **Transactions.** type Tx struct {

```
    Nonce uint64 // nonce of sender account
    GasPrice *big.Int // wei per gas
```

```
    Gas uint64 // gas limit
    To *common.Address ‘rlp:"nil"‘ // nil means contract creation
    Value *big.Int // wei amount
    Data []byte // contract invocation input data
    V, R, S *big.Int // signature values
    //Token values
    TokenOp token.TokenOp
    TokenAddr common.Address
    TokenStandard token.TokenStd
    TokenName string
    TokenSymbol string
    TokenVersion string
    // ERC-20 spec
    TokenTotalSupply uint64
    TokenDecimals uint
    // ERC-721 spec
    TokenBaseUrl string
    TokenTo common.Address
    TokenFrom common.Address
    TokenApproved bool
    TokenValue *big.Int
    TokenTokenId uint64
    TokenMetadata string
    TokenData []byte
}
```

5.3. **Tips.** Until the blocks are finalized, they remain sorted from the last finalized block chain for each block.

```
// BlockDAG represents a currently no descendants block
// of directed acyclic graph and related data.
type BlockDAG struct {
    Hash common.Hash
    LastFinalizedHash common.Hash
    LastFinalizedHeight uint64
    // ordered non-finalized ancestors hashes
    DagChainHashes common.HashArray
}

type HeaderChain struct {
    config *params.ChainConfig
    chainDb ethdb.Database
    genesisHeader *types.Header
    tips atomic.Value
    lastFinalisedHeader atomic.Value
```

```
    lastFinalisedHash common.Hash
    headerCache *lru.Cache // Cache for the most recent block headers
    numberCache *lru.Cache // Cache for the most recent finalized block numbers
    procInterrupt func() bool
    rand *mrand.Rand
    engine consensus.Engine
}
```

5.4. **Submitting Blocks in Parallel.** When the validator creates a block, it takes not only the transactions from the pool, but also adds all block hashes to parent-hashes, which are kept in DAG state tips. Also, the root node hash of the prefix tree state is added to the block, based on the block's past history (this means that the DAG has been sorted and transactions have been performed sequentially).

After the block is created, the node sends it to the network as soon as possible so that other nodes can add this block to tips and remove the block specified in parent-hashes.

Since the pools for all nodes are almost synchronized, if we create several blocks simultaneously, the transactions in all blocks will be the same. In order to create different blocks in parallel, two tasks must be solved:

(1) Selecting who can create blocks
(2) Preventing the same transaction from getting to multiple blocks

In Waterfall, only validators coordinated by the Beacon chain can create blocks. When validator distribution by committees for a slot in epoch occurs, the algorithm calculates how many validators can create blocks at a given moment, and in which order, depending on the load on the network.

The node creating the block knows its number and is unique within the slot. Each transaction has its own unique hash, which is a number. These numbers can be divided into groups and each group assigned to a specific validator. For the first step we will use the following algorithm:

(1) If the number of validators that can create a block in a given slot is a power of 2, we simply take the last bits of the transaction hash in that power, which gives us the number of the validator that will take a particular transaction.
(2) Otherwise, we take the last 8 bits of the hash, get a number, and divide it modulo by the number of validators that can create a block at a given moment. That number will be the number of the validator.

This functionality will be adjusted in upcoming versions of the protocol.

5.5. **Ordering and Finality.** When creating a block, we must order its past from the most recent final block and add the block itself, thus obtaining a chain of blocks.

Ordering algorithm:

Here we will use the following terminology.

Denote by $G = (V, E)$ a directed acyclic graph (DAG) with a set V of blocks and a set E of directed edges (representing the hash references to previous blocks). We write $v \in G$ for a block $v \in V$. Recall some concepts related to a graph G:

(1) $past(v, G)$ is the subset of blocks reachable from v;
(2) $future(v, G)$ is the subset of blocks from which v is reachable;
(3) $cone(v, G) = past(v, G) \cup \{v\} \cup future(v, G)$;
(4) $anticone(v, G) = V \setminus cone(v, G)$;
(5) $tips(G)$ is the set of all blocks with in-degree 0;
(6) $virtual(G)$ is a hypothetical block satisfying $past(virtual(G)) = G$.

The graph G has a unique block which is called genesis. This block is created at the inception of the system, and every valid block must have it in its past set.

Definition 5.1. Let $G = (V, E)$ be a blockDAG and let $k \in \mathbb{N}$. A subset $S \subseteq V$ is called a k-kluster, if, for every $v \in S$, we have $|anticone(v, G) \cap S| \leqslant k$.

Algorithm 1: GHOSTDAG

Input: G — a block DAG, k — the propagation parameter
Output: $BLUE_k(G)$ — the Blue set of G; ord — an ordered list containing all blocks in G

function $OrderDAG(G, k)$
  if $G == \{genesis\}$ **then**
    return $[\{genesis\}, \{genesis\}]$
  end
  for $v \in tips(G)$ **do**
    $[BlueSet_v, OrderedList_v] \leftarrow OrderDAG(past(v), k)$
  end
  $v_{max} \leftarrow \arg\max\{|BlueSet_v| : v \in tips(G)\}$ (break ties according to lowest hash)
  $BlueSet_G \leftarrow BlueSet_{v_{max}}$
  $OrderedList_G \leftarrow OrderedList_{v_{max}}$
  add v_{max} to $BlueSet_G$
  add v_{max} to the end of $OrderedList_G$
  for $v \in antocine(v_{max}, G)$ **do** in some topological ordering
    if $BlueSet_G \cup \{v\}$ *is a k-klaster* **then**
      add v to $BlueSet_G$
    end
    add v to the end of $OrderedList_G$
  end
  return $[BlueSet_G, OrderedList_G]$

Once the block is created or received from other nodes, the following algorithm works:

(1) If the node does not contain blocks specified in parent-hashes, unknown blocks are requested and uploaded until the node knows all blocks.
(2) If the node knows all BlockDAG blocks that are in parent-hashes of the current block, new blocks from "OrderBlocks" are delivered in this order:
    - Those with the greatest height in BlockDAG

- If the height is the same, BlockDAG is sorted by BlockHash

(3) If the node contains a BlockDAG of some current block parent-hash, then it is necessary to return to point (2) for the blocks that are not unknown by the BlockDAG and repeat recursively.
(4) The block that was created or received from other nodes is added to the end.
(5) In DAG state in Tips, all the BlockDAGs, including the current chain, are removed and the newly created one is added. We still keep the BlockDag for each block in the database.
(6) The created block contains a root node hash of the prefix tree that was previously created.

Once the nodes are well connected, it is rare to have to execute point (3).

Transactions are not executed until the block is finalized.

Waterfall uses the state of the network, which is why we need to calculate precisely when to execute the transactions and update the state. We will lean on the following points:

(1) Each node within a 2-slot timeframe receives information about all blocks created within the previous 2 slots
(2) Information about known blocks and their sorting is saved in BeaconChain
(3) The longest chain in Tips is finalized, except for the last 2 layers

Thus, after ordering another block, the following happens:

(1) The longest chain in Tips is taken.
(2) The blocks that were created for the last 2 slots are removed from the finalization chain, to be added in the future.
(3) The transactions are executed consistently, starting with the youngest block that was not added to the finalizing chain. The Merkle tree of states is used to speed up this process.
(4) The last finalized block is put in DAG State.
(5) The global state is updated and, as a result, the hash of the prefix state matches the hash of the last finalizing block.

6. Future Plans

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].

6.1. **Future Work.**

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. We may expand and/or change our approach to solving particular problems described in the sections of this paper.

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].

According to the most recent lab tests of our protocol, 1000 transactions per second will cause the transaction history to grow by ∼9,5 GB per day. By comparison, Visa processes approximately 1,700 transactions per second [15, 16]. Ethereum is planning to address this issue using Shard Chains [17]. 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 [18, 19]. 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 [20]. This and the previous method could both be implemented with sharding and combined with approaches that would be similar in part to [21].

Whilst reasonable efforts have been taken by The Waterfall to seek to ensure that the contents of the light paper are accurate and not misleading (at the date of writing), there can be no guarantee that the information set out in the light paper will not be out-dated or modified in the future. All statements of opinion or belief and all forward-looking statements (including statements of intent, opinions, projections, forecasts and estimates) relating to future events or performance contained in the light paper reflect The Waterfall's current intent, assessment and expectation and speak only as of the date specified in respect of such statements or otherwise the date of the light paper. These statements should not be read as commitments or as accurate indicators of future events or performance. No representation or assurance is given that any such intentions will not change, that any such statements are correct, or that any such events or performance will occur.

References

[1] Kiayias, Aggelos, et al. "Ouroboros: A provably secure proof-of-stake blockchain protocol". Annual International Cryptology Conference. Springer, Cham, 2017.

[2] Badertscher, Christian, et al. "Ouroboros genesis: Composable proof-of-stake blockchains with dynamic availability". Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security. 2018.

[3] David, Bernardo Machado, et al. "Ouroboros Praos: An adaptively-secure, semi-synchronous proof-of-stake protocol". IACR Cryptol. ePrint Arch. 2017 (2017): 573.

[4] Sompolinsky, Yonatan, Yoad Lewenberg, and Aviv Zohar. "SPECTRE: Serialization of proof-of-work events: confirming transactions via recursive elections". (2016).

[5] Ethereum. (2021, July 29). Devp2P/Rlpx.Md at master · ethereum/devp2p. GitHub. Retrieved September 24, 2021, from https://github.com/ethereum/devp2p/blob/master/rlpx.md.

[6] Buterin, Vitalik, et al. "Combining GHOST and Casper". arXiv preprint arXiv:2003.03052 (2020).

[7] Alurki, ML et al. "An executable model of Ethereum 2.0 Beacon Chain Phase 0 Specification". GitHub Report (2019).

[8] Sompolinsky, Yonatan, Shai Wyborski, and Aviv Zohar. "PHANTOM and GHOSTDAG: A scalable generalization of nakamoto consensus". Cryptol. ePrint Arch., Tech. Rep 104 (2018).

[9] Johnson, Sandra, Peter Robinson, and John Brainard. "Sidechains and interoperability". arXiv preprint arXiv:1903.04077 (2019).

[10] Wood, Gavin. "PolkaDot: vision for a heterogeneous multi-chain framework, draft 1". 2019-05-25. https://polkadot. network/PolkaDotPaper. pdf (2016).

[11] Markets Insider. "Ethereum transaction fees skyrocketed in the first QUARTER, ramping up costs for users, report finds". April 6, 2021. https://markets.businessinsider.com/currencies/news/ethereum-ether-transactions-fees-skyrocketed-q1-network-changes-2021-4-1030276867.

[12] Churyumov, Anton. "Byteball: A decentralized system for storage and transfer of value". URL https://byteball. org/Byteball. pdf (2016).

[13] Popov, Serguei, et al. "The coordicide". Accessed Jan (2020): 1–30.

[14] YouTube. "SPECTRE BPASE 18". Yonatan Sompolinsky, January 24–26th 2018, Stanford University.

[15] "Ethereum Transaction Fees 'Skyrocketed' in the the First QUARTER, Ramping up Costs for Users, Report Finds". Business Insider, Business Insider, https://markets.businessinsider.com/news/currencies/ethereum-ether-transactions-fees-skyrocketed-q1-network-changes-2021-4.

[16] Johnsen, Jahn Arne, Lars Erik Karlsen, and Sebjørn Sæther Birkeland. "Peer-to-peer networking with BitTorrent". Department of Telematics, NTNU (2005).

[17] Ethereum.org. “Shard Chains”. https://ethereum.org/en/eth2/shard-chains/.

[18] Benet, Juan. “IPFS-content addressed, versioned, P2P file system (DRAFT 3)”. arXiv preprint arXiv:1407.3561 (2014).

[19] Hedera Hashgraph. “HBAR”. https://hedera.com/hbar.

[20] R. Kumar and R. Tripathi, “Implementation of Distributed File Storage and Access Framework using IPFS and Blockchain”, 2019 Fifth International Conference on Image Information Processing (ICIIP), 2019, pp. 246–251, doi: 10.1109/ICIIP47207.2019.8985677.

[21] Hedera Hashgraph. “Hedera: A Public Hashgraph Network & Governing Council”. August 15, 2020. https://hedera.com/hh_whitepaper_v2.1-20200815.pdf