Exonum: Byzantine fault tolerant protocol for blockchains

Yury Yanovich, Ivan Ivashchenko, Alex Ostrovsky, Aleksandr Shevchenko, Aleksei Sidorov

Abstract

The need for consensus in distributed systems is an active area of research. There are many mathematical models with known theoretical limitations, as well as many correct and fast protocols. This research paper posits that the competitive blockchain industry requires foremost that consensus algorithms be fast regarding processed transactions per second. The new blockchain-specific consensus algorithm named Exonum is proposed in the paper. Its theoretical properties are proved and performance tests are provided.

Keywords: consensus, Byzantine fault tolerance, blockchain, peer-to-peer, performance evaluation

1. Introduction

A fundamental problem in distributed computing is achieving system reliability in the presence of faulty processes – to get consensus [1, 2]. The consensus problem requires agreement among a number of processes for a single data value, for example, a new bit value. The correct consensus classically means that three conditions hold for every execution of the algorithm

- *termination*: eventually each correct process sets its decision variable
- *agreement*: the decision value of all correct processes is the same
- *integrity*: if the correct processes all propose the same value, then any correct process in the decided state has chosen that value.

There are popular models for faulty processes [3, 4]

- *fail-stop*: process stops and cannot resume

- *omission*: process cannot send some messages, and it can delete some received messages without reading or even information about such received messages
- *authorized Byzantine failures* stand for failures in which absolutely no conditions are imposed, but all the messages supposed to be cryptographically authenticated by author processes
- *Byzantine failures* stand for failures in which absolutely no conditions are imposed.

For more examples and practical aspects of failures see [5].

The possibility of a consensus algorithm with specified properties depends on processes and their communication (message delivery) assumptions. The most common message delivery models are [3]

- *asynchronous system*: any message can be delivered at any time after sending, or fail to be delivered at all
- *partially synchronous system*: some unknown time T exists, such that each message is delivered faster than T.
- *synchronous system*: each message is delivered faster than a known finite latency T.

The processes also can be distinguished into three categories: *asynchronous*, *partially synchronous* or *synchronous* based on their relative computation speed.

The key results about Byzantine fault tolerant consensus are

- impossibility of distributed consensus with one faulty process (FLP impossibility) [6]: the correct consensus in the asynchronous system is impossible if a least one Byzantine process is presented
- the Byzantine fault tolerant (BFT) consensus algorithm exists in partially synchronous and synchronous systems if and only if the total number of processes N is greater than or equal to $3f + 1$, where f is a number of faulty processes [3, 1, 2]. The number of BFT protocols are developed [7, 8, 9, 10].

The blockchain technology [11] introduced new challenges and needs of the consensus problem solution. Originally, the economic Proof-of-Work [12, 13] solution was used in Bitcoin and other cryptocurrencies, which paradigm is one-CPU-one-vote. Such an approach is very energy consuming [14], has monopolization risks due to hardware and energy production limitations and guarantees only eventual consistency [15], which is not bad for permissionless blockchains and is a reasonable price for availability regarding CAP-theorem [16, 17, 18]. There are several Proof-of-X alternatives to overcome Proof-of-Work drawbacks for cryptocurrencies: stake [19, 20], activity [21], publication [22], storage [23, 24]. Cryptocurrencies are the most popular but not the only application of the blockchain technology [25, 26, 27, 28, 29, 30] and can be not only fully public [31, 18]. If they have a limited (or even known) list of entities having write access, then the Byzantine fault tolerant consensus algorithms can be a good solution for them [32].

In case of blockchain, the processes have to get an agreement for a new block of transactions. This introduces specific limitations [31, 33, 34]

- high performance regarding transactions per second (TPS)
- *censorship resistance* which can be formulated as *chain quality*
- *common prefix* which is specific for economic consensus
- *chain growth* property for fraction of blocks by honest authors.

There is a number of blockchain specific BFT protocols [35, 36] for partially synchronous systems and papers with nondeterministic [37] and specific asynchronous [38, 39, 40] models.

2. Assumptions

We propose a consensus algorithm for a blockchain network. We refer to processes as validators because there are also read-access-only processes – auditors – in blockchains. Each validator is either written to the genesis block or chosen through the consensus. The consensus is about the next block (at a known height H) to commit it to the blockchain. Blocks contain transactions – indivisible and irreducible series of database operations such that either all or none occur. Transactions either contain business operations (for example, tokens transfer in case of cryptocurrency) or change blockchain design (for example, validators list or anchoring frequency [41, 18]). Validators receive transactions from external clients to add them to the blockchain and exchange them through requests algorithm. The transactions which are included in the blockchain are called committed transactions.

Each validator has a set of transactions that have not been committed. This set is called a *pool of unconfirmed transactions*. In general, the pools of unconfirmed transactions are different for different validators. Hereafter we refer to validators and auditors as nodes. If necessary, the validators can request unknown transactions from other nodes.

Assumption 1. The validators (processes) are partially synchronous and the network is partially synchronous.

Under Assumption 1 we can assume that messages are processed instantly.

Let δt be an unknown upper bound for message delivery time.

Assumption 2. Each validator has infinite memory.

This assumption is a simplification to avoid dealing with DoS with consensus messages by Byzantine validators.

At each height H the consensus contains several rounds starting with 1. Rounds start according to some schedule, but stops only when consensus about a block at height H is achieved. If the validator is in round R, then he can process all the messages from round R and previous rounds.

Each validator uses its own timer for time estimation, and they do not send it or compare with each other. Each time the validator comes to a new height, its timer resets to zero. The time between round beginnings is positive and known by design. It grows to infinity with round number growth, for example, round R starts in $R^2 - 1$ seconds.

There is a commonly available validators enumeration at each height, which starts with 1 and finishes with N, where N is the number of validators at current height. Let f be the number of Byzantine processes. Let $a+$ be the abbreviation for “more than a”, where $a \in (0, 1)$. And let $a-$ be the abbreviation for “less than a”, where $a \in (0, 1)$.

Assumption 3. $N \geq 3f + 1$. Other words, the fraction of Byzantine validators is $1/3-$.

The set of validators and their order could be different for different heights. For example, the enumeration can be according to the lexicographical order of their public keys. There is a leader in each round, which is defined by the leader election algorithm (see Subsection 4.3) and could be calculated from the blockchain data. The leader proposes a new block for the current hight. The block contains the limited number of transactions from the set of unconfirmed ones. The size and number of transactions per block is limited to limit the block size.

Let (H, R) be the validator position, which means that the round R at height H has started for it, but the round $R + 1$ has not yet.

3. Algorithm Overview

The consensus algorithm overview is proposed in this Section.

3.1. Consensus Messages and Their Fields

The consensus algorithm uses the following types of messages

- `Propose`, `Prevote` and `Precommit` messages that correspond to three phases of the consensus algorithm
- `Request` messages used by full nodes to request missing data from peers
- `Block` messages used to transmit an entire block of transactions to a lagging full node
- Auxiliary messages, such as `Status` and `Connect`.

Only a part of their fields are described here

- `validator_id`: index of specific validator in `validators` list of configuration. This field is common to all types of messages
- `height`: height to which the message is related. This field is common to all types of messages
- `round`: round number to which the message is related. This field is common to all types of messages
- `hash`: hash of the message. This field is common to all types of messages
- `Propose.prev_hash`: hash of the previous block
- `Prevote.propose_hash`: hash of the `Propose` message to which `Prevote` belongs.

See source code at `https://github.com/exonum/exonum` for more details.

3.2. Parameters

The algorithm has a set of global configuration parameters

- `propose_timeout`: proposal timeout after the beginning of a new height
- `first_round_timeout`: time before the second round starts. This interval increases by 1.1 each round.
- `status_timeout`: period of sending the 'Status' message with a current height information.

and a set of node state variables

- `current_height`: current blockchain height
- `queued`: queue of consensus messages (`Propose`, `Prevote`, `Precommit`) from the future height or round
- `Proposes`: a set of known block proposals
- `locked_round`: round in which the node has locked on a proposal
- `current_round`: number of the current round
- `locked_propose`: `Propose` on which the node is locked
- `Proof-of-Lock`: a set of messages to prove the lock (see 3.3)
- `state_hash`: hash of the blockchain state
- `transactions_pool`: a set of transactions that have not yet been added to the blockchain. This set is called a pool of unconfirmed transactions. In general, the pools of unconfirmed transactions are different for different nodes.

3.3. Proof-of-Lock

A set of 2/3+ `Prevote` messages for the same propose from the nodes at the current round and blockchain height is called *Proof-of-Lock* (PoL). Nodes store PoL as a part of node state. A node can have no more than one stored PoL. We say that PoL is greater than the recorded one (has a higher priority), in cases when

1. there is no PoL recorded
2. the recorded PoL corresponds to a proposal with a smaller round number.

So PoLs are partially ordered. A node must replace the stored PoL with a greater PoL if it is obtained by the node during message processing.

3.4. Message Processing

Validators use a queue for message processing. Incoming request and consensus messages are placed in this queue when they are received. The same queue is used for timeouts processing. Timeouts are implemented as messages to this validator itself.

Messages from the next height (i.e., `current_height` + 1) or future round are placed into the separate queue (`queued`).

4. Algorithm Description

The algorithm description is provided in the current Section (see Figures 1 and 2). It includes the list of consensus stages, message processing and round leader election. The requests algorithm for messages is formulated in Appendix A.

4.1. Consensus Algorithm Stages

- *Full proposal* (availability of full proposal) occurs when the node gets complete info about some proposal and all the transactions from that proposal

- *Availability of* 2/3+ *Prevotes* occurs when the node collects 2/3+ `Prevote` messages from the same round for the same known proposal.

- *Lock* occurs when the node replaces the stored PoL (or collects its first PoL)

- *Commit* occurs when the node collects 2/3+ `Precommit` messages for the same round for the same known proposal. Corresponds to the Commit Node State.

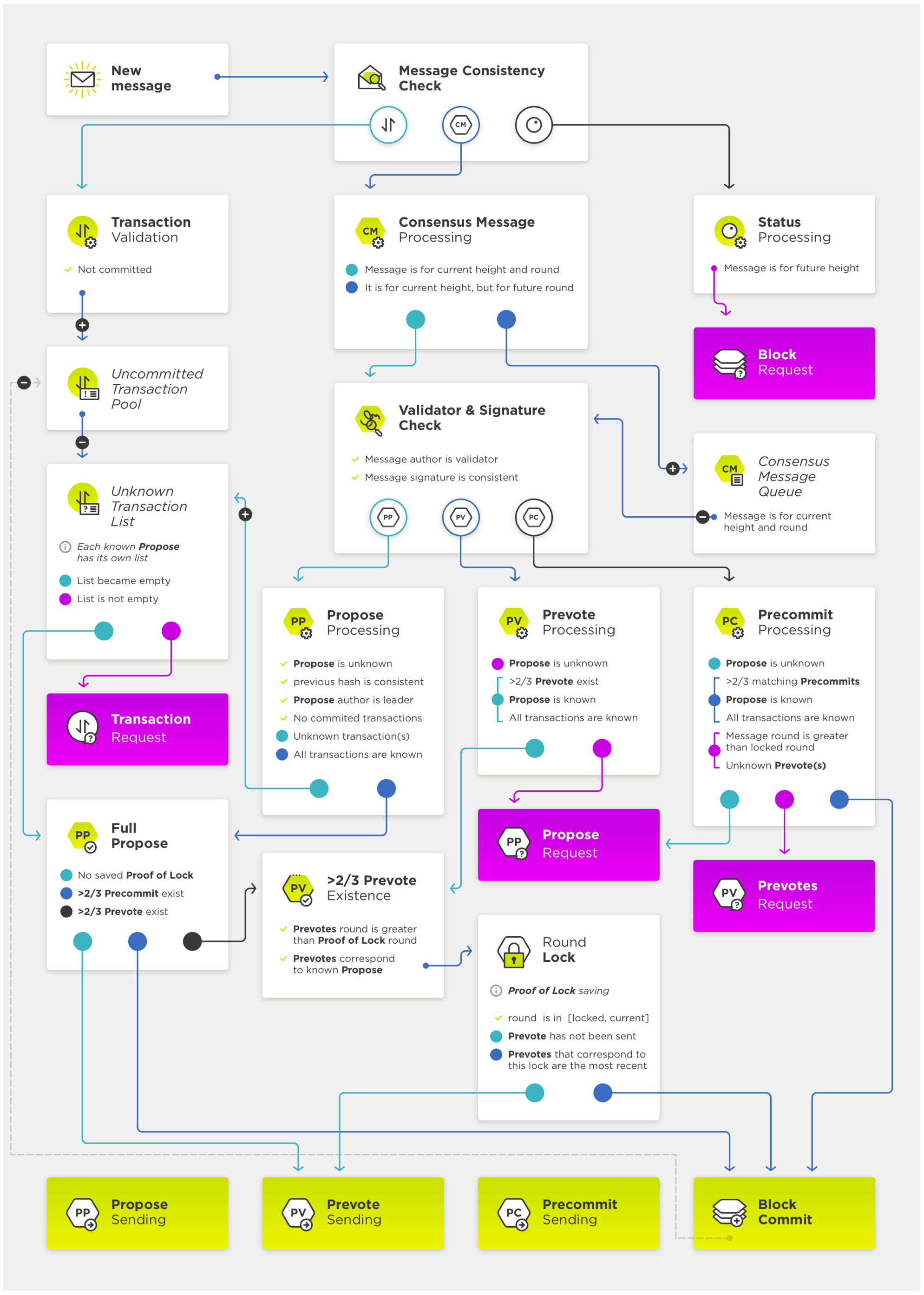
New message
Message Consistency Check
Transaction Validation
✓ Not committed
Consensus Message Processing
● Message is for current height and round
● It is for current height, but for future round
Status Processing
Message is for future height
Block Request
Uncommitted Transaction Pool
Validator & Signature Check
✓ Message author is validator
✓ Message signature is consistent
Consensus Message Queue
Message is for current height and round
Unknown Transaction List
Each known Propose has its own list
● List became empty
● List is not empty
Propose Processing
✓ Propose is unknown
✓ previous hash is consistent
✓ Propose author is leader
✓ No commited transactions
● Unknown transaction(s)
● All transactions are known
Prevote Processing
● Propose is unknown
>2/3 Prevote exist
● Propose is known
All transactions are known
Precommit Processing
● Propose is unknown
>2/3 matching Precommits
● Propose is known
All transactions are known
Message round is greater than locked round
● Unknown Prevote(s)
Transaction Request
Full Propose
● No saved Proof of Lock
● >2/3 Precommit exist
● >2/3 Prevote exist
Propose Request
Prevotes Request
>2/3 Prevote Existence
✓ Prevotes round is greater than Proof of Lock round
✓ Prevotes correspond to known Propose
Round Lock
Proof of Lock saving
✓ round is in [locked, current]
● Prevote has not been sent
● Prevotes that correspond to this lock are the most recent
Propose Sending
Prevote Sending
Precommit Sending
Block Commit

Figure 1: Exonum consensus algorithm schema

4.2. Receiving an Incoming Message

At the very beginning, the message is checked against the serialization format.

If any problems during deserialization are detected, such a message is ignored as something that the node cannot correctly interpret. If verification is successful, proceeds to consensus messages processing or transaction processing.

4.2.1. Consensus Messages Processing

- Do not process the message if it belongs to a future round or height. In this case, if the message refers to the height `current_height` + 1, it is added to the `queued` queue. If the message is related to the future height and updates the knowledge of the node, this information is saved according to the `requests algorithm`.
- If the message refers to a past height, it should be ignored.
- If the message refers to the current height and any round not higher than the current one, then the node
  - checks that the `validator_id` specified in the message is less than the total number of validators
  - check the message signature against the public key of the validator with index `validator_id` index
- If verification is successful, proceed to the message processing according to its type.

4.2.2. `Propose` Message Processing

Arguments: `Propose`.

- If `Propose` is known (its hash is already in the `Proposes` HashMap), ignore the message.
- Check `Propose.prev_hash` correctness.
- Check that the specified validator is the leader for the given round.
- Check that the proposal does not contain any previously committed transactions (`Propose` message contains only hashes of transactions, so the absence of hashes in the table of committed transactions is checked).

- Add the proposal to the `Proposes` HashMap.
- Form a list of transactions the node does not know from the `Propose`. Request transactions from this list.
- If all transactions are known, go to the full proposal.

4.2.3. Transaction Processing

- If the transaction is already committed, ignore the message.
- If such a transaction is already in the pool of unconfirmed transactions, ignore the message.
- If absent, add the transaction to the unconfirmed transaction pool.
- For all known proposals where this transaction is included, exclude the hash of this transaction from the list of unknown transactions. If the number of unknown transactions becomes zero, proceed to the full proposal for the current proposal.

4.2.4. Full Proposal

Arguments: `Propose`.

- If the node does not have a saved PoL, send `Prevote` message in the round to which the proposal belongs.
- For each round r in the interval $[\max\{\texttt{locked\_round} + 1, \texttt{propose.round}\}, \texttt{current\_round}]$
  - If the node has 2/3+ `Prevotes` for `Propose` in r, then proceed to "Availability of 2/3+ `Prevotes`" for the `Propose` in r.
- For each round r in the interval $[\texttt{propose.round}, \texttt{current\_round}]$
  - If 2/3+ `Precommits` r available for the `Propose` in r and with the same `state_hash`, then the node
    - Executes the proposal, if it has not yet been executed.
    - Checks that the resulting `state_hash` of the node coincides with the `state_hash` of the majority (if not, the node must stop working and signalize error).
    - Proceeds to `COMMIT` for this block.

4.2.5. Availability of 2/3+ ***Prevotes***

- Cancel all requests for `Prevotes` that share 'round' and the `propose_hash` fields with the collected `Prevotes`.
- If the `locked_round` of the node is less than `Prevote.round` and the hash of the locked `Propose` message corresponding to this `Prevote` is the same as the `Prevote.propose_hash`, then proceed to `Lock` for this very proposal.

4.2.6. ***Prevote*** *Message Processing*

Arguments: `Prevote`.

- Add `Prevote` to the list of known `Prevote` messages for its proposal in the `Prevote.round` round
- If the node has formed 2/3+ `Prevote` messages for the same round and `propose_hash`
  - `locked_round` < `prevote.round`
  - the node knows `Propose` corresponding to this `Prevote`
  - the node knows all of its transactions
- Then proceed to availability of 2/3+ Prevotes for `Propose` in the `Prevote.round` round
- If the node does not know `propose` or any transactions, request them.

4.2.7. ***Precommit*** *Message Processing*

- Add the message to the list of known `Precommits` for the `Propose` in this round with the given `state_hash`.
- If
  - the node has formed 2/3+ `Precommits` for the same round and `propose_hash`
  - the node knows the `Propose`
  - the node knows all of its transactions
- Then

  - Execute the proposal, if it has not yet been executed.
  - Check that the nodes `state_hash` coincides with the `state_hash` of the majority (if not, the node must stop working and signalize error).
  - Proceed to `Commit` for this block.

- Else
  - Request `Propose`, if it is not known.
  - If the message round is bigger than the `locked_round`, request `Prevotes` from the message round.

4.2.8. Lock

- For each round r in the interval [`locked_round`, `current_round`]
  - If the node has not sent `Prevote` in r, send it for the `locked_propose`.
  - If the node has formed 2/3+ `Prevotes` in r, then change `locked_round` to `current_round`, `locked_propose` to `Propose.hash` (`Propose` corresponds to 2/3+ `Prevotes` in r).
  - If the node does not send `Prevote` for any other proposal except `locked_propose` in subsequent rounds after `locked_round`, then
    - Execute the proposal, if it has not yet been executed.
    - Send `Precommit` for the `locked_propose` in the `current_round`.
    - If the node has 2/3+ `Precommits` for the same round with the same `block_hash`, then proceed to `Commit`.

4.2.9. Commit

- Add a block to the blockchain.
- Push all the transactions from the block to the table of committed transactions.
- Update current height.
- Set the value of the `locked_round` variable to 0 at a new height.
- Delete all the transactions of the committed block from the pool of unconfirmed transactions.

- If the node is the leader, form and send `Propose` and `Prevote` messages after `propose_timeout` expiration.
- Process all messages from the `queued`, if they become relevant (their round and height coincide with the current ones).
- Add a timeout for the next round of the new height.

4.2.10. `Block` Message Processing

Arguments: `Propose`.

`Block` messages are requested by the validators if they see that some consensus messages belong to a future height.

- Check the block message
  - The key in the `to` field must match the key of the node.
  - `Propose.prev_hash` of the correspondent `Propose` matches the hash of the last committed block.
- If the message structure is correct, proceed to check the block contents.
  - The block height should be equal to the current height of the node.
  - The number of `Precommit` messages should be sufficient to reach consensus.
  - All `Precommit` messages must be correct.
- If the check is successful, then check all transactions for correctness and if they are all correct, then proceed to their execution to check the resulting block hash. If this is not the case, then a critical error has occurred: either the majority of the network is Byzantine or the nodes' software is corrupted.
- Add the block to the blockchain and move to a new height. Set to 0 the value of the `locked_round` variable at the new height.
- If there are validators who claim that they are at a bigger height, then turn to the request of the block from the higher height.

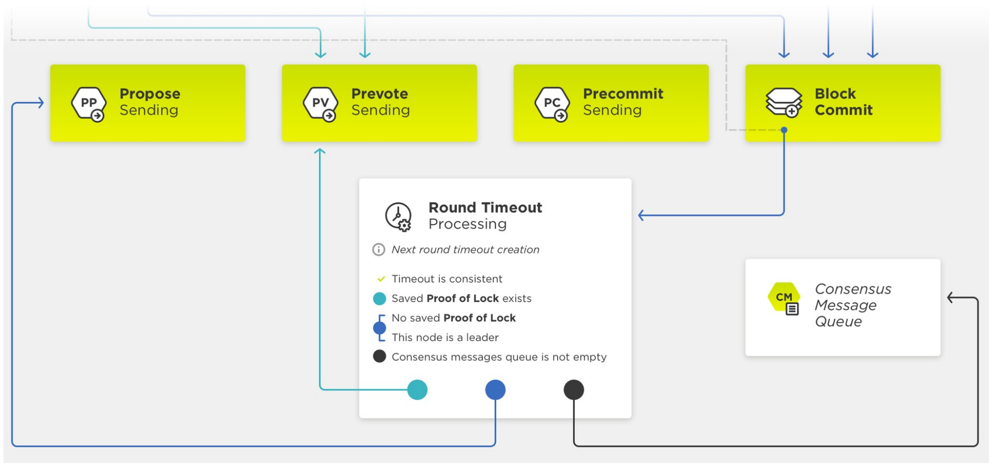

Figure 2: Round timeout processing

4.2.11. Round Timeout Processing

- If the timeout does not match the current height and round, skip further timeout processing.
- Add a timeout (its length is specified by the `round_timeout` variable and a corresponding function) for the next round.
- Process all messages from the `queued`, if they become relevant (their round and height coincide with the current ones).
- If the node has a saved PoL, send `Prevote` for the `locked_propose` in a new round, proceed to availability of 2/3+ Prevotes.
- Else, if the node is the leader, form and send `Propose` and `Prevote` messages (after the expiration of the `propose_timeout`, if the node has just moved to a new height).

4.2.12. Status Timeout Processing

- If the height of the node has not changed since the timeout was set, then send out a ‘Status‘ message to all peers.
- Add a timeout for the next ‘Status‘ broadcast.

4.3. Leader Election

Recall, that f is the number of Byzantine validators, as well as there are $N \geq 3f + 1$ validators in total. H is the current height for the blockchain.

1. Every node whose block proposal is accepted by the network (that is, after the node becomes the author of the newly accepted block) is moving to the locked state for F blocks. During the next $\alpha \cdot N$ blocks it does not have a right to create new block proposals, where α is a parameter of the leader election algorithm such that $\alpha \in [1/3, 2/3)$ and $\alpha \cdot N$ is an integer. The node behaves itself as usual in other activities, that is it votes for a new block, signs messages, etc. Remove authors of the accepted block proposals at the previous $\alpha \cdot N$ heights from the list of candidates for leaders. The list of candidates to be elected as a leader at each height includes $M = N - \alpha \cdot N = (1 - \alpha)N > N \cdot 1/3 > f$ validators. So there is an honest validator among candidates for each height H. Let us re-enumerate them as $0, 1, \ldots, M - 1$ according to their base numbers.
2. Shuffle this list pseudo-randomly. Shuffling should be deterministic but the place of each node should be uniformly distributed over the list. To do so, we take a permutation over these M validators. The permutation number is calculated as $I = \texttt{Hash}(H) \texttt{ mod } M!$. This calculation provides uniform distribution of the orders, so Byzantine validators are pseudo uniformly randomly distributed inside the current H height.

5. Algorithm Properties

Theoretical properties of the proposed algorithm are formulated and proved in this Section.

Statement 1. Round Beginning

All non-Byzantine nodes being at a height of not less than H, will be in the state (H, R) or higher (either bigger round or bigger height), where R is an arbitrary fixed constant.

Proof

We will prove the statement for every single non-Byzantine node. That node shall move to a new height in a finite time (and in this case the condition will be satisfied) or remain at the height H. In the second case, the node increments the round counter at known finite intervals (by stopwatch). The round counter of any non-Byzantine validator will be increased to the value

R no more than in finite time T as the processes are partially synchronous and due to the round timeout increase schedule. Thus, all non-Byzantine validators will move to the state (H, R) or higher.

Statement 2. Non-Byzantine Leader

For each height H there exists a round in which a non-Byzantine node will become a leader.

Proof

Property of leader election algorithm: even if a malefactor takes control over f nodes, one will not be the author of every future block.

Statement 3. Deadlock Absence

A certain non-Byzantine node will sooner or later send some message relating to the consensus algorithm (`Propose`, `Prevote`, `Precommit`).

Proof

Let us prove it by contradiction. Assume that each non-Byzantine node sends no messages for an arbitrarily long period of time; then that node updates neither the current height nor the PoL state (**Prevote** message is sent for the new PoL upon the coming of a new round in the case of a PoL update, if no other message has been sent before this time). Consider the cases of PoL status:

1. *Some non-Byzantine node has a saved PoL.* Then this node will send a **Prevote** message for the proposal saved in the PoL when the next round timeout occurs (unless it sends any other message earlier).
2. *No a single non-Byzantine node has a saved PoL.* Then there will always come another round in which some non-Byzantine node will be the leader (see Statement 2). In this case, the node will form a new proposal and send **Propose** and **Prevote** messages.

Corollary

If there exists an unlimited number of heights at which the validator can become a leader (property of leader election algorithm), then any non-Byzantine node will send an arbitrarily large number of messages related to the consensus algorithm (`Propose`, `Prevote`, `Precommit`).

Statement 4. Obligatory Block Acceptance (Liveness)

There necessarily will come a point in the system when the non-Byzantine node will accept a new block to commit it into the blockchain.

Proof

Suppose the network be at a certain height H; then the maximum height of the blockchain of a non-Byzantine node is equal to H. In accordance with

Statement 3, all non-Byzantine nodes will be able to move up to the height H or higher. Next, the state is considered when all the non-Byzantine nodes are at height H.

Let $R(T)$ denote the round following the maximum round among non-Byzantine validators, at the moment T according to the clock of an outside observer.

Similarly, $T(R)$ is the time of coming of the R round according to the clock of an outside observer for all non-Byzantine validators.

Let the non-Byzantine node be the leader for the first time in the round with the number $R < R^*$ (where R^* denotes a uniform estimate of R over validators and $R^* \leq f+2$ for our leader election algorithm). Then the coming time of the round R^* for all non-Byzantine nodes according to the outside observer's watch is $T(R^*)$.

Not later than at the moment $T(R^*) + \delta t +$ `propose_timeout` each non-Byzantine node will receive a correct proposal from the R round. Further, not later than through $2\delta t$, that node will know all the transactions from this proposal (request mechanism). Denote this time $T^* = T(R^*) +$ `propose_timeout` $+ 3\delta t$.

If no non-Byzantine node has PoL to the $R(T^*)$ round, then in this round the node will receive PoL (for the proposal from the R round). Indeed, if no one has PoL, then the nodes cannot send the `Prevote` message in the $R(T^*)$ round. In accordance with the algorithm for processing complete proposals, the confirming `Prevote` message will be sent.

Thus, by the time $T' = T(R(T^*)) + \delta t$ at least one non-Byzantine node will have PoL.

Not later than $T'' = T(R(T')) + 2\delta t$ each non-Byzantine node will have some PoL. Indeed, starting with the $R(T')$ round, the non-Byzantine node will send `Prevote` messages for the proposal from its PoL. Non-Byzantine nodes that do not have PoL will be able to get this PoL through the request mechanism by the time T''.

None of the non-Byzantine nodes will send `Prevote` for new proposals since the moment T''. Hence, new PoL will not appear in the network.

During one iteration $T(R(\star)) + 2\delta t$ at least one non-Byzantine validator will increase its PoL. Indeed, all the non-Byzantine nodes already have some PoLs. In this case, they will always send `Prevote` messages for the corresponding proposals. And according to the logic of their processing, if the non-Byzantine node receives `Prevote` pointing to a larger PoL, a request for missing `Prevote` for this (bigger) PoL occurs.

Since there exists finite number of the validators and possible proposals, it follows that in some finite time T'', 2/3+ of all validators will receive PoL for the same proposal. After that they will be able to send `Precommit` messages.

Not later than time $T(R(T''')) + \delta t$ at least one non-Byzantine validator will accept the new block and hence some node will correctly add the block to the blockchain.

Statement 5. Absence of Forks (Consensus Finality)

If some non-Byzantine node commits a block to the blockchain, then no other node can add another block, confirmed with 2/3+ `Precommit` messages, to the blockchain at the same height.

Proof

Suppose some node adds block B block to the blockchain. This can only happen if that node goes into the `Commit` stage. There exist three possibilities of the transition to the `Commit`: from `Lock`, prevote message processing, and full proposal. In all these cases, the condition of the transition is the presence of 2/3+ `Precommit` messages for some proposal P from round R and the result of applying the corresponding block leads to the same `state_hash`. Since the number of Byzantine nodes is 1/3−, 1/3− of the non-Byzantine nodes send `Precomit` messages in the corresponding round. Such a message could only be sent within the `Lock` stage in which the PoL is stored for the proposal P in the round R. This can happen only if these nodes do not send `Prevote` messages in rounds $R' > R$ for $P' \neq P$ (special condition for sending the `Precommit` message). Also, these nodes send `Prevote` messages in all rounds after R until their current rounds. Thus, since the remaining nodes are 2/3−, we have two consequences.

1. In no round after R we can get PoL (in other words go to the `Lock` stage) for the $P' \neq P$ proposal, because this requires 2/3+ `Prevote` messages.
2. In all rounds of $R' > R$, new PoLs cannot emerge in the network, except for PoLs related to the proposal P (and, accordingly, to the block B). Indeed, at the beginning of the round following the current round, the specified 1/3+ of the non-Byzantine nodes will be in the state with the saved PoL corresponding to the P proposal. And consequently they will send `Prevote` messages only for the saved P proposal according to the round timeout processing.

Thus, messages of the `Precommit` type cannot be sent for any other block proposal. This means that none of the non-Byzantine nodes can add another block to the blockchain.

Corollary. Deadlock absence for asynchronous network

The property of fork absence will be preserved also in the case of an asynchronous network.

Proof

The proof of Statement 5 does not in any way use the assumption of partial synchronism. Therefore, it is also true in an asynchronous network.

Statement 6. Pulling Nodes Up

Any non-Byzantine node eventually will get all the blocks included in the blockchain by any other non-Byzantine node.

Proof

Let node A fall behind for some reason from node B. And node A is at height H, while node B is at height $H + h$, where $h > 0$. We will show that in a finite time node A can be pulled to the height $H + 1$.

All messages described in the algorithm and related to the consensus algorithm (`Propose`, `Prevote`, `Precommit`, `Status`) contain the current height. Thus, as soon as node B sends any of these messages and the message is delivered to A, node A will understand that it is behind and will request the next block (it can do this not from node B, but for any other node); if the block is added, then the block will be correct due to the absence of forks. In accordance with the corollary from Statement 3, node B always sends some message of the consensus algorithm.

Theorem 7. Consensus Algorithm Correctness

The proposed consensus algorithm is correct for each height, i.e. it satisfies termination, agreement, and integrity.

Proof

Terminations follows from Statements 4 and 6. Agreement is equivalent to Statement 5. Integrity follows from Statement 4.

Statement 8. Censorship Resistance

There are least $\alpha N - f > 0$ committed blocks from honest leaders among each αN consequent blocks.

Proof

Each validator can be a leader for a committed block once per $\alpha N > f$ consequent blocks by leader election algorithm design. So for each αN consequent blocks, there can be f or less blocks from Byzantine validators. All other blocks are from honest validators.

Note. Statement 7 guaranties chain quality [33] with fraction $\frac{f}{\alpha N} < 1$ *of of the commitTed blocks, proposed by Byzantine nodes, which is consistent with [34] for* $\alpha \approx 2/3$ *and* $f: N = 3f + 1$.

6. Exonum Distinguishing Features

Exonum uses consensus algorithm slightly similar to PBFT [7], but it has a number of distinctive features compared to other BFT algorithms.

6.1. Unbounded Rounds

Rounds have a fixed start time but they do not have a definite end time (a round ends only when the next block is received). This helps decrease delays when the network connection among validators is unstable.

Assume that consensus messages from a certain round need to be processed within the round. If the state of the network deteriorates, the network might not manage to accept the proposal until the end of the round. Then in the next round the entire process of nominating a proposal and voting for it must begin again. The timeout of the next round should be increased so that the block could be accepted during the new round timeout with a poor network connectivity. The need to repeat anew the work that has already been done and an increase in the timeout will lead to additional delays in accepting the block proposal.

In contrast to the case discussed in the previous paragraph, the absence of a fixed round end in Exonum allows the system to accept the proposal with a minimum necessary delay.

6.2. Work Split

`Propose` messages include only transactions hashes. (Transactions are included directly into `BlockResponse` messages). Furthermore, transactions execution is delayed; transactions are applied only at the moment when a node locks on a Propose.

Delayed transactions processing reduces the negative impact of malicious nodes on the system throughput and latency. Indeed, it splits transactions processing among the stages of the algorithm:

At the prevote stage, validators only ensure that a list of transactions included in the proposal is correct (the validator checks that all the transactions in the Propose are already stored by this node. The correctness of a transaction is verified when the transaction is received; nodes do not store incorrect transactions.) At the precommit stage, validators apply the transactions to the current blockchain state. At the commit stage, validators ensure that they achieved the same state after applying the transactions in the proposal. If a Byzantine validator sends out proposals with a different transactions order to different validators, the validators do not need to spend time checking the order and applying the transactions at the prevote stage. A different transactions order will be detected when comparing the `propose_hash` received in the `Prevote` messages from other validators and the `propose_hash` received in the `Propose` message.

Thus, the split of work helps reduce the negative impact of Byzantine nodes on the overall system performance.

6.3. Requests Algorithm

Requests algorithm (see Appendix A) allows a validator to restore any consensus info from other validators. This has a positive effect on system liveness.

7. Experiments

7.1. Environment Design

We performed experiments with two network configurations: in a single data center (DC) and multiple geographically distributed DCs. In both cases, twenty-one virtual machines were used (16 validators, 4 transaction generators, 1 benchmark control instance). Each validator was running on a separate virtual machine with 3.75 GiB RAM, 2 Core Intel Xeon Platinum CPUs running @3.4GHz, and the blockchain database was stored on an EBS drive connected to each instance. Nodes used Exonum version 0.9. In case of

- **Single DC:** virtual machines were in one availability zone within one AWS region (eu-central-1).
- **Multiple DCs:** validators were distributed among 14 AWS DCs in different locations: N.Virginia, Ohio, N.California, Oregon, Mumbai, Seoul, Singapore, Sydney, Tokyo, Canada, Frankfurt, Ireland, London and Paris. Generators were distributed in 4 different DCs in different regions: N.Virginia, Sydney, Tokyo and Frankfurt. Benchmark control instance was located at the Frankfurt AWS DC.

7.2. Blockchain Design

In all experiments the following consensus parameters were used

- block capacity: 2000 transactions
- propose timeout: 0 seconds
- signature size: 64 bytes.

Two types of blockchain were considered

- `timestamping`: each transaction contains an author's public key and a hash of the file to be timestamped. Transactions have no business logic interaction and there are no specific Merkle proof structures.
- `cryptocurrency`: each public key is associated with a cryptocurrency wallet. A transaction has an input address, an output address and amount. One needs enough tokens to send and be able to sign the transaction. Blockchain storage contains Merkle index for each wallet.

Four generators send transactions to all validators during the experiments with a constant flow. The flow is chosen in order to be a bit bigger than the blockchain tps. Each validator check transaction signature before adding a transaction into the pull of unconfirmed transaction. Given scenario can be considered as a real-life high-load mode.

Experiments code will be available at `https://github.com/exonum` after paper release.

7.3. Performance Tests

We measured transactions per second (`TPS`, the bigger the better) for the different total amount of validators. The results were averaged over 100 000 transactions processed for each case. The mean value and standard deviation were of interest. Hereafter the blue line with squares and magenta filling represents the mean and standard deviation for the `timestamping`, the black line with circles and green filling represents the same for the `cryptocurrency`.

Note. Almost all the blocks were filled with transactions in our experiments, so one can estimate block acceptance time in seconds as 2000/`TPS`.

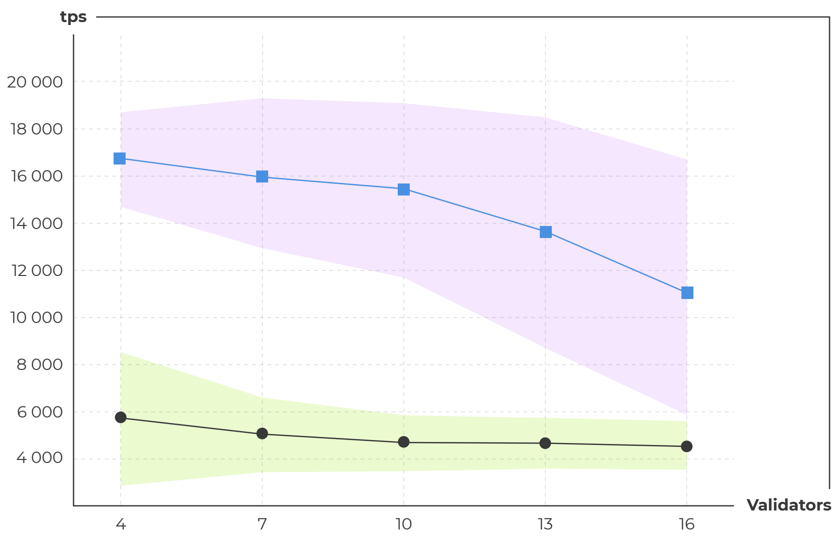

Figure 3: **TPS** as a function of validators number in a **Single DC**. The blue line with squares and magenta filling represents the mean and standard deviation for the timestamping, the black line with circles and green filling represents the same for the cryptocurrency.

7.3.1. Different Validators Number

The number of consensus messages over network grows as a square of validators number in Exonum. It decreases blockchain performance. We considered different number of validators to estimate this effect.

`TPS` experimental results are in Figure 3 for a **Single DC** and in Figure 4 for **Multiple DCs**.

Exonum shows around 5000 tps in a **Single DC** for the cryptocurrency. This amount is almost independent of validators number as most likely writing Merkle proofs to EBS Drive process is the most time-consuming. Exonum shows more than 10 000 tps for the timestamping in a **Single DC** and this amount slightly decreases with the validators number growth.

Exonum shows more than 4000 tps in the **Multiple DC** for the cryptocurrency with 4 validators and more than 2000 tps for 16 validators. It shows from 13 000 to 4 000 tps for the timestamping and this amount slightly decreases with the validators number growth.

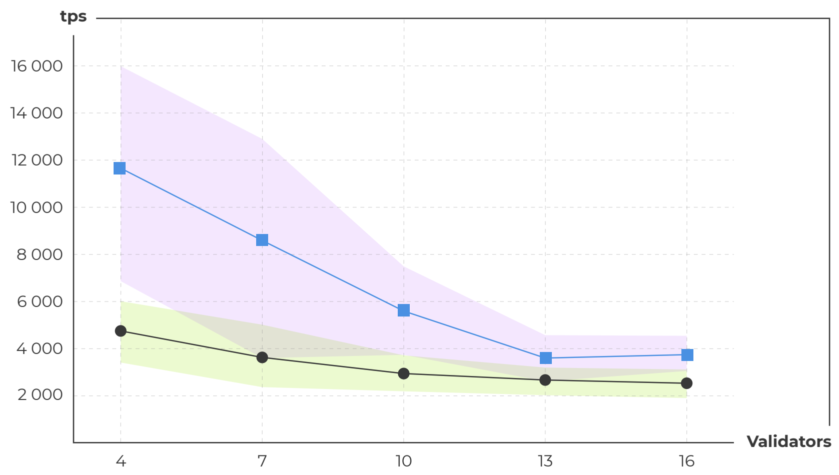

Figure 4: **TPS** as a function of validators number in **Multiple DCs**.

7.3.2. Fail-Stop Validators

The simplest model of validators failures is fail-stop. We choose it to demonstrate how improper nodes behavior slows consensus down. The total number of validators was 16 and from 0 to 5 were stopped (up to 1/3). TPS as a function of working validators fraction is in Figures 5 and 6.

Exonum shows more than 4000 tps in a **Single DC** and more than 1800 tps in **Multiple DCs** for the cryptocurrency. This amount decreases with the working validators fraction decrease up to 20% maximum value. The results for the timestamping are better: more than 10 000 tps in a **Single DC** and more than 2700 tps in **Multiple DCs**. The number of fail-stop validators slightly increases variance for **Multiple DCs**.

8. Conclusion

We proposed and implemented Exonum – the new Byzantine fault tolerant consensus algorithm for blockchains. It is able to process 4 000 transactions per second with 0.5 seconds block acceptance time over the globally distributed network. Exonum's performance is stable in case of fail-stop validators, but can significantly decrease with the total number of validators growth.

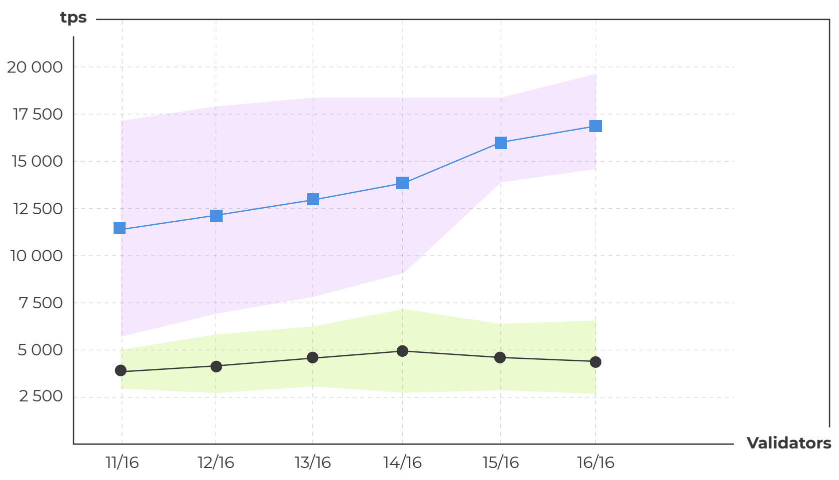

Figure 5: **TPS** as a function of working validators fraction in a **Single DC**. The blue line with squares and magenta filling represents the mean and standard deviation for the timestamping, the black line with circles and green filling represents the same for the cryptocurrency.

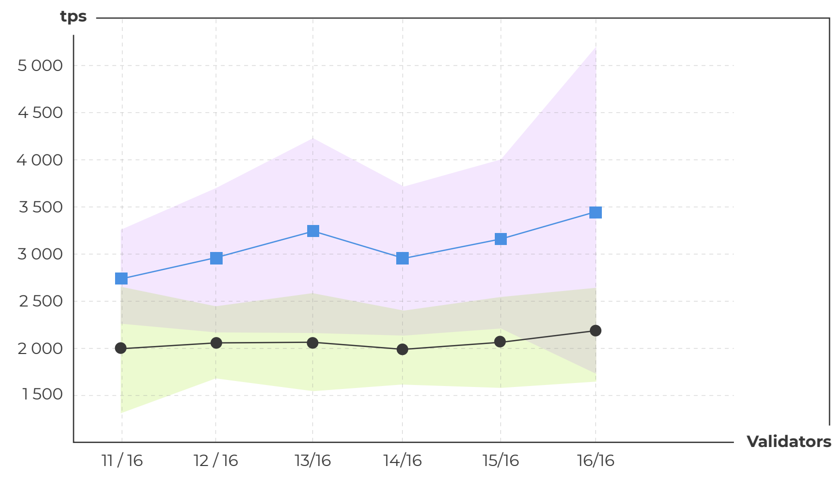

Figure 6: **TPS** as a function of working validators fraction in **Multiple DCs**.

9. Acknowledgments

Authors would like to express their most profound appreciation to Exonum team.

References

[1] Lynch, N.: Distributed Algorithms. Morgan Kaufmann Publishers (1996)

[2] Dollimore, J., Kindberg, T., Coulouris, G.: Distributed Systems: Concepts and Design (2005)

[3] Dwork, C., Lynch, N., Stockmeyer, L.: Consensus in the presence of partial synchrony. Journal of the ACM **35**(2) (4 1988) 288–323

[4] Lamport, L., Shostak, R., Pease, M.: The Byzantine Generals Problem. ACM Transactions on Programming Languages and Systems **4**(3) (7 1982) 382–401

[5] Driscoll, K., Hall, B., Sivencrona, H., Zumsteg, P.: Byzantine Fault Tolerance, from Theory to Reality. In: International Conference on Computer Safety, Reliability, and Security. Springer, Berlin, Heidelberg (2003) 235–248

[6] Fischer, M.J., Lynch, N.A., Paterson, M.S.: Impossibility of distributed consensus with one faulty process. Journal of the ACM **32**(2) (4 1985) 374–382

[7] Castro, M., Liskov, B.: Practical Byzantine fault tolerance and proactive recovery. ACM Transactions on Computer Systems **20**(4) (11 2002) 398–461

[8] Cowling, J., Myers, D., Liskov, B., Rodrigues, R., Shrira, L.: HQ replication: A hybrid quorum protocol for Byzantine fault tolerance. Proceedings of the 7th symposium on Operating systems design and implementation. USENIX Association (2006) 177–190

[9] Kotla, R., Alvisi, L., Dahlin, M., Clement, A., Wong, E.: Zyzzyva. ACM SIGOPS Operating Systems Review **41**(6) (2007) 45

[10] Aublin, P.L., Guerraoui, R., Knežević, N., Quéma, V., Vukolić, M.: The Next 700 BFT Protocols. ACM Transactions on Computer Systems **32**(4) (2015) 1–45

[11] Nakamoto, S.: Bitcoin: A Peer-to-Peer Electronic Cash System. www.bitcoin.org (2008) 1–9

[12] Dwork, C., Naor, M.: Pricing via Processing or Combatting Junk Mail. In: Advances in Cryptology CRYPTO 92. Springer Berlin Heidelberg, Berlin, Heidelberg (1992) 139–147

[13] Abadi, M., Burrows, M., Manasse, M., Wobber, T.: Moderately Hard, Memory-bound Functions. ACM Transactions on Internet Technology **5** (5 2005)

[14] Malone, D., O'Dwyer, K.: Bitcoin Mining and its Energy Footprint. In: 25th IET Irish Signals & Systems Conference 2014 and 2014 China-Ireland International Conference on Information and Communities Technologies (ISSC 2014/CIICT 2014), Institution of Engineering and Technology (2014) 280–285

[15] Tschorsch, F., Scheuermann, B.: Bitcoin and Beyond: A Technical Survey on Decentralized Digital Currencies. IEEE Communications Surveys & Tutorials **18**(3) (2016) 2084–2123

[16] Brewer, E.A.: Towards robust distributed systems (abstract). In: Proceedings of the nineteenth annual ACM symposium on Principles of distributed computing - PODC '00, New York, USA, ACM Press (2000) 7

[17] Bitfury Group, Garzik, J.: Public versus Private Blockchains Part 2: Permissionless Blockchains. bitfury.com (2015) 1–20

[18] Bitfury Group: On Blockchain Auditability. bitfury.com (2016) 1–40

[19] King, S., Nadal, S.: PPCoin : Peer-to-Peer Crypto-Currency with Proof-of-Stake. Self-published paper (2012) 1–6

[20] Kiayias, A., Russell, A., David, B., Oliynykov, R.: Ouroboros: A provably secure proof-of-stake blockchain protocol. In: Lecture Notes in

Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics). Volume 10401 LNCS., Springer, Cham (8 2017) 357–388

[21] Bentov, I., Lee, C., Mizrahi, A., Rosenfeld, M.: Proof of Activity. ACM SIGMETRICS Performance Evaluation Review **42**(3) (2014) 34–37

[22] Clark, J., Essex, A.: CommitCoin: Carbon Dating Commitments with Bitcoin. In: Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics). Volume 7397 LNCS. Springer (2012) 390–398

[23] Lauter, K.: Cryptographic Cloud Storage. Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics) **6054 LNCS** (2010) 136–149

[24] Zheng, Q., Xu, S.: Secure and efficient proof of storage with deduplication. In: Proceedings of the second ACM conference on Data and Application Security and Privacy - CODASKY '12, New York, New York, USA, ACM Press (2012) 1

[25] Swan, M.: Summary for Policymakers. In Intergovernmental Panel on Climate Change, ed.: Climate Change 2013 - The Physical Science Basis. Cambridge University Press, Cambridge (2015) 1–30

[26] Pilkington, M.: Blockchain Technology: Principles and Applications. In: Research Handbook on Digital Transformations. Springer (2016) 225 – 253

[27] Kim, H.M., Laskowski, M.: Towards an Ontology-Driven Blockchain Design for Supply Chain Provenance. SSRN Electronic Journal **25**(1) (8 2016) 18–27

[28] Kuo, T.T., Kim, H.E., Ohno-Machado, L.: Blockchain distributed ledger technologies for biomedical and health care applications. Journal of the American Medical Informatics Association **24**(6) (11 2017) 1211–1220

[29] Angraal, S., Krumholz, H.M., Schulz, W.L.: Blockchain Technology. Circulation: Cardiovascular Quality and Outcomes **10**(9) (9 2017) 5665–5690

[30] Mamoshina, P., Ojomoko, L., Yanovich, Y., Ostrovski, A., Botezatu, A., Prikhodko, P., Izumchenko, E., Aliper, A., Romantsov, K., Zhebrak, A., Ogu, I.O., Zhavoronkov, A.: Converging blockchain and next-generation artificial intelligence technologies to decentralize and accelerate biomedical research and healthcare. Oncotarget **9**(5) (1 2018) 5665–5690

[31] Buterin, V.: On Public and Private Blockchains - Ethereum Blog (2015)

[32] Vukolić, M.: The quest for scalable blockchain fabric: Proof-of-work vs. BFT replication. In: Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics). Volume 9591. (2016) 112–125

[33] Kiayias, A., Panagiotakos, G.: Speed-Security Tradeoffs in Blockchain Protocols. Cryptology ePrint Archive (2015) 1–19

[34] Garay, J., Kiayias, A., Leonardos, N.: The Bitcoin Backbone Protocol: Analysis and Applications. In Oswald, E., Fischlin, M., eds.: Advances in Cryptology - EUROCRYPT 2015. Volume 9057 of Lecture Notes in Computer Science. Springer Berlin Heidelberg, Berlin, Heidelberg (2015) 281–310

[35] Kwon, J.: TenderMint : Consensus without Mining (2014)

[36] Cachin, C.: Architecture of the Hyperledger Blockchain Fabric. IBM Research **July** (2016)

[37] Zhao, W.: Byzantine fault tolerance for nondeterministic applications. In: Proceedings - DASC 2007: Third IEEE International Symposium on Dependable, Autonomic and Secure Computing, IEEE (9 2007) 108–115

[38] Défago, X., Schiper, A., Urbán, P.: Total order broadcast and multicast algorithms. ACM Computing Surveys **36**(4) (12 2004) 372–421

[39] Miller, A., Xia, Y., Croman, K., Shi, E., Song, D.: The Honey Badger of BFT Protocols. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security - CCS'16, New York, New York, USA, ACM Press (2016) 31–42

[40] Miller, A., Xia, Y., Croman, K., Shi, E., Song, D.: Efficient Asynchronous Atomic Broadcast. IBM Research (2016) 1–4

[41] Crespo, S.D.P.A., Luis García Cuende, I.: Stampery Blockchain Timestamping Architecture (BTA). arXiv (2016) 1–21

Appendix A. Requests in Consensus Algorithm

Requests are used to obtain unknown information from nodes that signal the presence of such information via consensus messages (for example, via a message indicating a blockchain height greater than the local blockchain height). The algorithm for generating and handling requests is an integral part of the Exonum consensus algorithm.

Appendix A.1. Learning from Consensus Messages

Receiving a consensus message from a node gives the message recipient an opportunity to learn certain information about the state of the message author (a node that has signed the message; the message author may differ from the peer that the message recipient got the message from), if the author is not Byzantine. The receiving node saves this information in the `RequestState` structure.

Appendix A.2. `Sending Requests`

This algorithm determines the behavior of the node at different stages of the consensus algorithm if the node needs to request information from other nodes. The following subsections describe events that cause a specific response.

For each sent request, the node stores a `RequestState` structure, which includes the number of request attempts made and a list of nodes that should have the required information. `RequestState` for each request is placed into a hash map where the key is an identifier of the requested data (hash for `Propose` and `Transactions`, round and hash for `Prevotes`, height for `Block`). When the requested info is obtained, the node deletes `RequestState` for the corresponding request (cancels request).

The node sets a timeout for each sent request. The timeout is implemented as a message to this node itself. This message is queued and processed when it reaches its queue. Timeout deletion (cancelling timeout) means its deletion from the message queue.

Cancelling the request means cancelling a corresponding timeout as well.

Appendix A.2.1. `Receiving Transaction`

If this is the last transaction required to collect a known `Propose`, cancel the corresponding `RequestTransactions`.

Consensus Message from a bigger height

- Update info about the height of the blockchain on the corresponding node

- Send `RequestBlock` for the current height (height of the latest committed block +1) to the message author, if such a request has not been sent yet

Appendix A.2.2. ***`Receiving Propose`***

- If this `Propose` has been requested, cancel the request. A list of nodes that should have all transactions mentioned in the `Propose` message is copied from the `RequestState` before its deletion, to request missing transactions, if necessary

- If certain transactions from the `Propose` are not known, send `RequestTransactions` to the author of the `Propose`. Set the nodes in `RequestState` for this request as calculated at the previous step.

Appendix A.2.3. ***`Receiving Prevote`***

- If the node does not have the corresponding the `Propose`, send `RequestPropose` to the author of `Prevote`

- If the sender specifies `lock_round`, which is greater than the stored Proof-of-Lock (PoL)], send `RequestPrevotes` for the locked proposal to the author of the `Prevote`

- If the node has formed 2/3+ `Prevote` messages for the same proposal, cancel the `RequestPrevotes` request for the `Prevote` messages corresponding to this proposal (if they have been requested already requested earlier).

Appendix A.2.4. ***`Receiving Precommit`***

- If the node does not have a corresponding `Propose`, send `RequestPropose` to the author of the `Precommit`

- If the message corresponds to a larger round than the saved PoL, send `RequestPrevotes` for this round to the author of the `Precommit`

- If the node has formed 2/3+ `Precommit` messages for the same proposal, cancel the corresponding `RequestPrecommits` (if they have been requested already).

Appendix A.2.5. ***`Receiving Block`***

- Request the following block in the blockchain from the node (if one exists) that sent any message from the height greater than the `current height` + 1. If there are several such nodes, request is sent to the one from which the message from the height greater than `current height` + 1 has been received earlier
- Update current height after committing the block locally
- Cancel `RequestBlock` for the height at which the block has just been committed.

Appendix A.2.6. ***`Peers Timeout`***

- Send a `RequestPeers` request to a random peer (auditor or validator) from the list of known peers specified in the local configuration.
- Move to a new height
- Cancel all requests.

Appendix A.2.7. ***`Request Timeout`***

- Delete the node, to which the request has been sent, from the list of nodes that should have the requested data (that list is a part of the `RequestState` structure)
- If the list of nodes having the data to be requested is empty, cancel the request
- Otherwise, make one more request attempt to another node from the list of nodes that should have the requested data and start a new timer.

Appendix A.3. ***`Requests Processing`***

This algorithm determines the processing of different types of request messages by the node.

Appendix A.3.1. ***`RequestPropose`***

- If the message corresponds to a height that isn't equal to the current height of the node, ignore the message
- If the node has a `Propose` with the corresponding hash at the given height, send it

Appendix A.3.2. ***RequestTransactions***

Send all the requested transactions the node has as separate messages. Transactions can either be already committed or be in the pool of unconfirmed transactions.

Appendix A.3.3. ***RequestPrevotes***

- If the message does not match the height at which the node is, ignore the message
- Send as individual messages all the corresponding `Prevotes` except those that the requestor has.

Appendix A.3.4. ***RequestBlock***

- If the message corresponds to a height not less than that of the node, ignore the message
- Form a `Block` message from the blockchain data and send it to the requestor.

Appendix A.3.5. ***RequestPeers***

Send all the saved `Connect` messages from peers to the requestor.

Appendix B. Algorithm Operation Example

Suppose that the system has only four validator nodes (see Figure 1). Also, the fourth node is hacked and behaves arbitrarily (it is Byzantine). Sending of messages and changing state in the Figure are designated by circles with the inscriptions corresponding to action type

- PP: `Propose` message
- PV: `Prevote` message
- PC: `Precommit` message
- RQ: `Request` message
- C: moving to the `Commit` state.

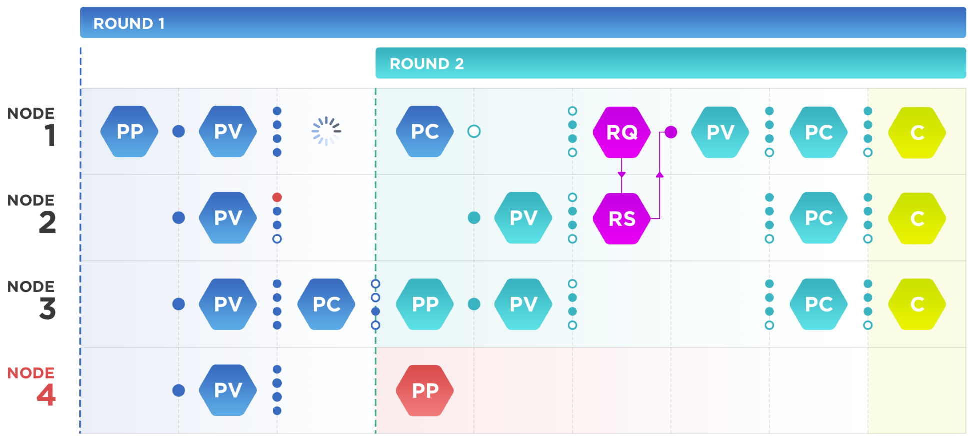

Figure B.7: Consensus algorithm operation example

Note that 2/3+ nodes in such a system equals 3 or more nodes, and 1/3− is one node or less.

Let all nodes simultaneously move to the next height H and start their work with round 1, and their stopwatches are synchronous. Let node 1 be selected as a leader node according to the leader-election algorithm (see Subsection 4.3) in this round. This node forms a `Propose` and sends it to all the other nodes. Having received this message, the nodes (including node 1) check it and, if it is compiled in the proper way, send out their own `Prevote` messages to everyone. Byzantine node 4 can send the `Prevote` message to one node, while do not send anything to the others or send messages with the wrong format. Suppose, after sending `Prevote` messages, it turned out that nodes 1 and 3 received 4 `Prevote` messages, and node 2 received only two (for example, node 4 did not send `Prevote` to node 2, and the message from node 1 was damaged).

Nodes that received 2/3+ `Prevote` messages (i.e., nodes 1 and 3 in this case) claim this by sending all `Precommit` messages. If a node collects 2/3+ messages `Precommit`, it adds the corresponding block to itself in the block system.

Note that due to delays in the delivery and processing of messages, the sending of any messages of the algorithm can be asynchronous from the point of view of different nodes. So, according to the picture, node 1 sent its `Precommit` message later than node 2.

Suppose node 2 has not yet received any of the `Precommit` messages. From this node, the block, referring to the proposal from the first round, has not yet received enough `Prevote` votes to be accepted into the blockchain. From nodes 1 and 3, the same block has not received enough `Precommits` votes.

Within a short time after the start of the first round, the second round begins in the system (this time is called the round timeout). Let node 3 be the leader in the second round. It forms a proposal and sends it out. Note that the proposal sent by the leader node may not be completely new in relation to proposals that have already been at this height. So, in round 2, node 3 can, as a proposal, send the proposal from round 1.

Byzantine node 4 can try to independently send out its offer. Due to the fact that the general algorithm for choosing leaders is used, its proposal will be rejected by the whole network.

Suppose that the correct `Propose` in the second round has been received only by node 2 and, as a consequence, nodes 2 and 3 sent `Prevote` messages. Let these `Prevote` messages reach node 1. In this case, node 1, which has not received a `Propose` in this round, will understand that it does not have enough information to process `Prevote` messages. According to the algorithm, it will request additional information (in this case it's a `Propose` message) from the node that owns it (for example, node 2). As a result, node 1 will be able to independently check the `Propose` message and send its `Prevote`.

Finally, nodes 1, 2, and 3 will receive three `Prevote` messages. This will result in the sending of `Precommit` messages. Having received them, the nodes will decide to add a block corresponding to the proposal from the second round, to the block, which will transfer the non-Byzantine nodes to the height of $H + 1$.

In this example, for simplicity, Proof-of-Look (PoL) was not mentioned. The PoL is required to prove the lack of forks and to reduce the number of proposals considered within the network.