Correctness of Tendermint-Core Blockchains

Yackolley Amoussou-Guenou
Institut LIST, CEA, Université Paris-Saclay, F-91120, Palaiseau, France
Sorbonne Université, CNRS, Laboratoire d'Informatique de Paris 6, F-75005 Paris, France

Antonella Del Pozzo
Institut LIST, CEA, Université Paris-Saclay, F-91120, Palaiseau, France

Maria Potop-Butucaru
Sorbonne Université, CNRS, Laboratoire d'Informatique de Paris 6, F-75005 Paris, France

Sara Tucci-Piergiovanni
Institut LIST, CEA, Université Paris-Saclay, F-91120, Palaiseau, France

Abstract

Tendermint-core blockchains (e.g. Cosmos) are considered today one of the most viable alternatives for the highly energy consuming proof-of-work blockchains such as Bitcoin and Ethereum. Their particularity is that they aim at offering strong consistency (no forks) in an open system combining two ingredients (i) a set of validators that generate blocks via a variant of Practical Byzantine Fault Tolerant (PBFT) consensus protocol and (ii) a selection strategy that dynamically selects nodes to be validators for the next block via a proof-of-stake mechanism. The exact assumptions on the system model under which Tendermint underlying algorithms are correct and the exact properties Tendermint verifies, however, have never been formally analyzed. The contribution of this paper is as follows. First, while formalizing Tendermint algorithms we precisely characterize the system model and the exact problem solved by Tendermint, then, we prove that in eventual synchronous systems a modified version of Tendermint solves (i) under additional assumptions, a variant of one-shot consensus for the validation of one single block and (ii) a variant of the repeated consensus problem for multiple blocks. These results hold even if the set of validators is hit by Byzantine failures, provided that for each one-shot consensus instance less than one third of the validators is Byzantine.

2012 ACM Subject Classification Computer systems organization → Dependable and fault-tolerant systems and networks

Keywords and phrases Blockchain, Consensus, Proof-of-Stake, Fairness

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2018.16

Related Version A full version is available at `https://eprint.iacr.org/2018/574.pdf`.

Acknowledgements The authors thank anonymous reviewers for their insightful comments.

1 Introduction

Blockchain is today one of the most appealing technology since its introduction in the Bitcoin White Paper [30] in 2008. Blockchain systems, similar to P2P systems in the early 2000, take their roots in the non academic research. After the releasing of the most popular blockchains (e.g. Bitcoin [30] or Ethereum [36]) with a specific focus on economical transactions, their huge potential for various other applications ranging from notary to medical data recording became evident. In a nutshell, Blockchain systems maintain a continuously-growing history of ordered information, encapsulated in blocks. Blocks are linked to each other by relying on

collision resistant hash functions, i.e., each block contains the hash of the previous block. The Blockchain itself is a distributed data structure replicated among different peers. In order to preserve the chain structure those peers need to agree on the next block to append in order to avoid forks. The most popular technique to decide which block will be appended is the *proof-of-work* mechanism of Dwork and Naor [15]. The block that will be appended to the blockchain is owned by the node (miner) having enough CPU power to solve first a crypto-puzzle first. The only possible way to solve this puzzle is by repeated trials. The major criticisms for the *proof-of-work* approach are as follows: it is assumed that the honest miners hold a majority of the computational power, the generation of a block is energetically costly, which yield to the creation of mining pools and finally, multiple blockchains might coexist in the system due to accidental or intentional forks.

Recently, the non academic research developed alternative solutions to the proof-of-work technique such as *proof-of-stake* (the power of block building is proportional to the participant wealth), *proof-of-space* (similar to proof-of-work, instead of CPU power the prover has to provide the evidence of a certain amount of space) or *proof-of-authority* (the power of block building is proportional to the amount of authority owned in the system). These alternatives received little attention in the academic research. Among all these alternatives *proof-of-stake* protocols and in particular those using variants of *Practical Byzantine Fault-Tolerant* consensus [8] became recently popular not only for in-chain transaction systems but also in systems that provide cross-chain transactions. Tendermint [27, 7, 25, 28] was the first in this line of research having the merit to link the *Practical Byzantine Fault-Tolerant* consensus to the proof-of-stake technique and to propose a blockchain where a dynamic set of validators (subset of the participants) decide on the next block to be appended to the blockchain. Although, the correctness of the original Tendermint protocol [27, 7, 25] has never been formally analyzed from the distributed computing perspective, it or slightly modified variants became recently the core of several popular systems such as Cosmos [26] for cross-chain transactions.

In this paper we analyse the correctness of the original Tendermint agreement protocol as it was described in [27, 7, 25] and discussed in [28, 22]. The code of this protocol is available in [35]. One of our fundamental results proved in this paper is as follows:

In an eventual synchronous system, a slightly modified variant of the original Tendermint protocol implements the one-shot and repeated consensus, provided that (i) the number of Byzantine validators, f, is $f < n/3$ where n is the number of validators participating in each single one-shot consensus instance and (ii) eventually a proposed value will be accepted by at least $2n/3 + 1$ processes (Theorem 7 and Theorem 8).

More in detail, we prove that the original Tendermint (specified for the first time in a preliminary version of this work, see technical report [2]) verifies the consensus termination with a small twist in the algorithm (a refinement of the timeout) and with the additional assumption stating that there exists eventually a proposer such that its proposed value will be accepted, or voted, by more than two-third of validators. The rest of the paper is organized as follows. Related works are discussed in Section 2. Section 3 defines the model and the formal specifications of one-shot and repeated consensus. Section 4 formalizes the original Tendermint One-Shot and Repeated Consensus protocols through pseudo-code and proves the correctness of the One-Shot Consensus algorithm. Due to space limitations, the fairness study along with full descriptions of the counter-examples that motivate the modification of the original algorithm and the additional assumptions for correctness are provided in [2].

2 Related Work

Interestingly, only recently distributed computing academic scholars focus their attention on the theoretical aspects of blockchains motivated mainly by the intriguing claim of popular blockchains, as Bitcoin and Ethereum, that they implement consensus in an asynchronous dynamic open system. This claim is refuted by the famous impossibility result in distributing computing [17]. In distributed systems, the theoretical studies of *proof-of-work* based blockchains have been pioneered by Garay et *al* [19]. Garay *et al.* decorticate the pseudo-code of Bitcoin and analyse its agreement aspects considering a synchronous round-based communication model. This study has been extended by Pass *et al.* [31] to round based systems where messages sent in a round can be received later. [16] proposes a mix between proof-of-work blockchains and proof-of-work free blockchains referred as Bitcoin-NG. Bitcoin-NG inherits, however, the drawbacks of Bitcoin: costly proof-of-work process, forks, no guarantee that a leader in an epoch is unique, no guarantee that the leader does not change the history at will if it is corrupted. On another line of research, in [11] Decker *et al.* propose the PeerCensus system that targets linearizability of transactions. PeerCensus combines the proof-of-work blockchain and the classical results in Practical Byzantine Fault Tolerant agreement area. PeerCensus suffers the same drawbacks as Bitcoin because of the proof-of-work. Byzcoin [24] builds on top of *Practical Byzantine Fault-Tolerant* consensus [8] enhanced with a scalable collective signing process. [24] is based on a leader-based consensus over a group of members chosen by a proof-of-membership mechanism. When a miner succeeds to mine a block, it gains a membership share, and the miners with the highest shares are part of the fixed size voting member set. In the same spirit, SBFT [21] and Hyperledger Fabric [3] build on top of [8]. In [32] and [20], *sortition* based blockchains are discussed, where the proof-of-work mechanism is completely replaced by a probabilistic ingredient.

The only academic works that address the consensus in *proof-of-stake* based blockchains are [10] and [23]. In [10], Daian *et al.* which proposes a protocol for weakly synchronous networks. The execution of the protocol is organized in epochs. Similar to Bitcoin-NG [16] in each epoch a different committee is elected and inside the elected committee a leader will be chosen. The leader is allowed to extend the new blockchain. The protocol is validated via simulations and only partial proofs of correctness are provided. Ouroboros [23] proposes a sortition based proof-of-stake protocol and addresses mainly the security aspects of the proposed protocol. Red Belly [9] focuses on consortium blockchains, where only a predefined subset of processes are allowed to update the blockchain, and proposes a Byzantine consensus protocol.

Interestingly, none of the previous academic studies made the connection between the repeated consensus specification [5, 13, 12] and the repeated agreement process in blockchain systems. Moreover, in terms of fairness of rewards, no academic study has been conducted related to blockchains based on repeated consensus.

3 System model and Problem Definition

The system is composed of an infinite set Π of asynchronous sequential processes, namely $\Pi = \{p_1, \dots\}$; i is called the *index* of p_i. *Asynchronous* means that each process proceeds at it own speed, which can vary with time and remains unknown to the other processes. *Sequential* means that a process executes one step at a time. This does not prevent it from executing several threads with an appropriate multiplexing. As local processing times are negligible with respect to message transfer delays, they are considered as being equal to zero.

Arrival model. We assume a *finite arrival model* [1], i.e. the system has infinitely many processes but each run has only finitely many. The size of the set $\Pi_\rho \subset \Pi$ of processes that participate in each system run is not a priori-known. We also consider a finite subset $V \subseteq \Pi_\rho$ of validators. The set V may change during any system run and its size n is a-priori known. A process is promoted in V based on a so-called merit parameter, which can model for instance its stake in proof-of-stake blockchains. Note that in the current Tendermint implementation, it is a separate module included in the Cosmos project [26] that is in charge of implementing the selection of V.

Communication network. The processes communicate by exchanging messages through an eventually synchronous network [14]. *Eventually Synchronous* means that after a finite unknown time τ there is an upper bound δ on the message transfer delay.

Failure model. There is no bound on processes that can exhibit a Byzantine behaviour [33] in the system, but up to f validators can exhibit a Byzantine behaviour at each point of the execution. A Byzantine process is a process that behaves arbitrarily: it can crash, fail to send or receive messages, send arbitrary messages, start in an arbitrary state, perform arbitrary state transitions, etc. Byzantine processes can control the network by modifying the order in which messages are received, but they cannot postpone forever message receptions. Moreover, Byzantine processes can collude to "pollute" the computation (e.g., by sending messages with different contents, while they should send messages with the same content if they were non-faulty). A process (or validator) that exhibits a Byzantine behaviour is called *faulty*. Otherwise, it is *non-faulty* or *correct*. To be able to solve the consensus problem, we assume that $f < n/3$.

Communication primitives. In the following we assume the presence of a broadcast primitive. A process p_i broadcasts a message by invoking the primitive broadcast($\langle TAG, m\rangle$), where TAG is the type of the message, and m its content. To simplify the presentation, it is assumed that a process can send messages to itself. The primitive broadcast() is a best effort broadcast, which means that when a correct process broadcasts a value, eventually all the correct processes deliver it. A process p_i receives a message by executing the primitive delivery(). Messages are created with a digital signature, and we assume that digital signatures cannot be forged. When a process p_i delivers a message, it knows the process p_j that created the message.

Let us note that the assumed broadcast primitive in an open dynamic network can be implemented through *gossiping*, i.e. each process sends the message to current neighbors in the underlying dynamic network graph. In these settings the finite arrival model is a necessary condition for the system to show eventual synchrony. Intuitively, a finite arrival implies that message losses due to topology changes are bounded, so that the propagation delay of a message between two processes not directly connected can be bounded [29, 4].

Problem definition. In this paper we analyse the correctness of Tendermint protocol against two abstractions in distributed systems: one-shot consensus and repeated consensus defined formally as follows.

▶ **Definition 1** (One-Shot Consensus)**.** We say that an algorithm implements One-Shot Consensus if and only if it satisfies the following properties:

- **Termination.** Every correct process eventually decides some value.
- **Integrity.** No correct process decides twice.

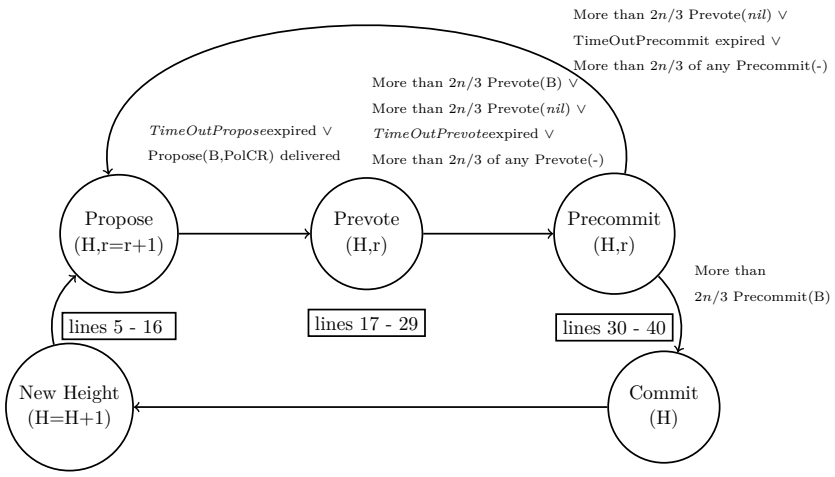


Figure 1 State Machine for Tendermint One-Shot algorithm described in Figure 3.

- **Agreement.** If there is a correct process that decides a value B, then eventually all the correct processes decide B.
- **Validity[9].** A decided value is valid, it satisfies the predefined predicate denoted isValid().

The concept of multi-consensus is presented in [5], where the authors assume that only the faulty processes can postpone the decision of correct processes. In addition, the consensus is made a finite number of times. The long-lived consensus presented in [13] studies the consensus when the inputs are changing over the time, their specification aims at studying in which condition the decisions of correct process do not change over time. None of these specifications is appropriate for blockchain systems. In [12], Delporte-Gallet *et al.* defined the Repeated Consensus as an infinite sequence of One-Shot Consensus instances, where the inputs values may be completely different from one instance to another, but where all the correct processes have the same infinite sequence of decisions. We consider a variant of the repeated consensus problem as defined in [12]. The main difference is that we do not predicate on the faulty processes. Each correct process outputs an infinite sequence of decisions. We call that sequence the *output* of the process.

▶ **Definition 2** (Repeated Consensus). An algorithm implements a repeated consensus if and only if it satisfies the following properties:

- **Termination.** Every correct process has an infinite output.
- **Agreement.** If the i^{th} value of the output of a correct process is B, then B is the i^{th} value of the output of any other correct process.
- **Validity.** Each value in the output of any correct process is valid, it satisfies the predefined predicate denoted isValid().

4 Tendermint Formalization

4.1 Informal description of Tendermint and its blockchain

Tendermint protocol [27, 7] aims at building a blockchain without forks relying on a variant of PBFT consensus. When building the blockchain, a subset of fixed size n of processes called

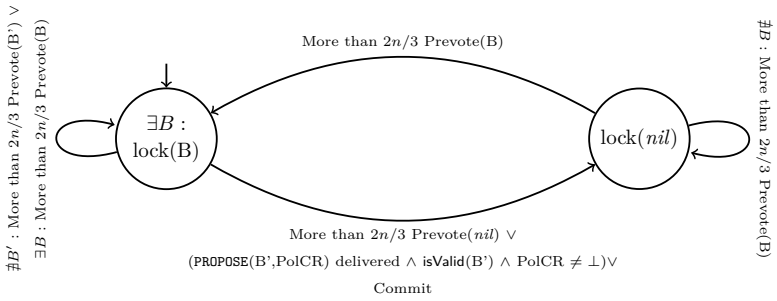


Figure 2 State machine Lock/Unlock

validators should agree on the next block to append to the blockchain. The set of validators is deterministically determined by the current content of the blockchain, referred as the history. We note that this subset may change once a block is appended. The mechanism to choose the validators from a given history is further referred as *selection mechanism*. Note that in the current Tendermint implementation, it is a separate module included in the Cosmos project [26] that is in charge of implementing the selection mechanism. Intuitively, such mechanism should be based on the proof-of-stake approach but its actual implementation is currently left open.

The first block of Tendermint blockchain, called the *genesis block*, is at *height* 0. The *height* of a block is the distance that separates that block to the genesis block. Each block contains: (i) a *Header* which contains a pointer to the previous block and the height of the block, (ii) the *Data* which is a list of transactions, and (iii) a set *LastCommit* which is the set of validators that signed on the previous block. Except the first block, each block refers to the previous block in the chain. Given a current height of Tendermint blockchain, a total ordered set of validators V is selected to add a new block. The validators start a *One-Shot Consensus algorithm*. The first validator creates and proposes a block B, then if more than $2n/3$ of the validators accept B, B will be appended as the next block, otherwise the next validator proposes a block, and the mechanism is repeated until more than $2n/3$ of the validators accept a block. For each height of Tendermint blockchain, the mechanism to append a new block is the same, only the set of validators may change. Therefore, Tendermint applies a *Repeated Consensus algorithm* to build a blockchain, and at each height, it relies on a *One-Shot Consensus algorithm* to decide the block to be appended. Due to space limitations, Repeated Consensus proofs are provided in [2].

Although the choice of validators is managed by a separate module (see Cosmos project [26]) the rewards for the validators that contributed to the block at some specific height H are determined during the construction of the block at height $H + 1$. The validators for H that get a reward for H are the ones that validators for $H + 1$ "saw" when proposing a block. This mechanism can be unfair, since some validator for H may be slow, and its messages may not reach the validators involved in $H + 1$, implying that it may not get the rewards it deserved. Due to space limitations, our fairness study are provided in [2].

4.2 Tendermint One-Shot Consensus algorithm

Tendermint One-Shot Consensus algorithm is a round-based algorithm used to decide on the next block for a given height H. In each *round* there is a different proposer that proposes a block to the validators that try to decide on that block. A round consists of three steps: (i) the *Propose step*, the proposer of the round broadcasts a proposal for a block; (ii) the *Prevote*

Function consensus(H, Π_ρ, signature); %One-Shot Consensus for the height H with the set Π_ρ of processes%

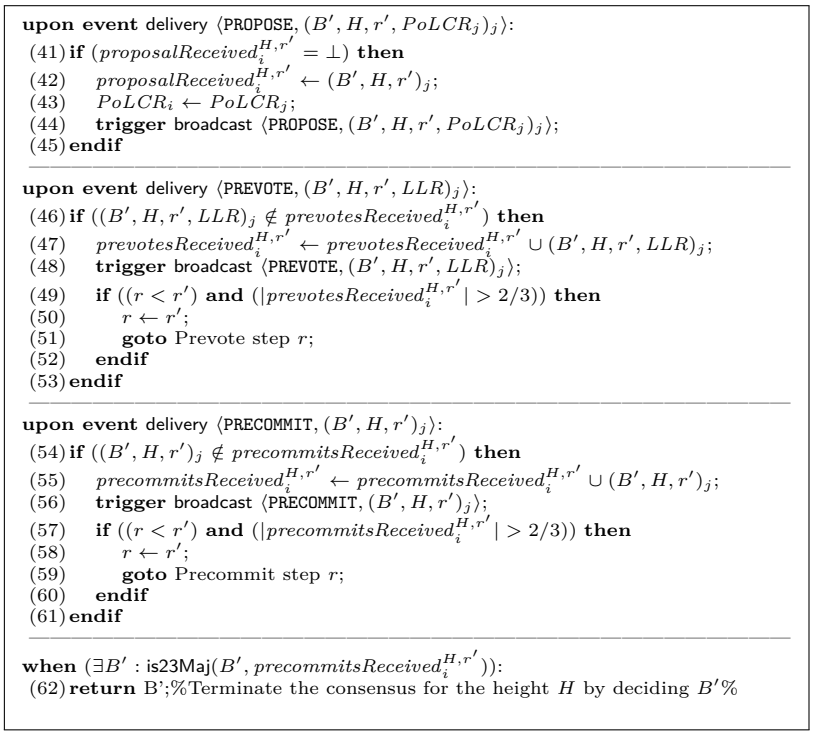
Init:
(1) $r \leftarrow 0$; $LLR_i \leftarrow -1$; $PoLCR_i \leftarrow \perp$; $lockedBlock_i \leftarrow nil$; $B \leftarrow nil$;
(2) $TimeOutPropose \leftarrow \Delta_{\text{Propose}}$; $TimeOutPrevote \leftarrow \Delta_{\text{Prevote}}$;
(3) $proposalReceived_i^{H,r} \leftarrow \perp$; $prevotesReceived_i^{H,r} \leftarrow \perp$; $precommitsReceived_i^{H,r} \leftarrow \perp$;

while (true) **do**
(4) $r \leftarrow r + 1$; $PoLCR_i \leftarrow \perp$;
Propose step r
(5) **if** ($p_i ==$ proposer(H, r)) **then**
(6) **if** ($LLR_i \neq -1$) **then** $PoLCR_i \leftarrow LLR_i$; $B \leftarrow lockedBlock_i$;
(7) **else** $B \leftarrow$ createNewBlock(signature);
(8) **endif**
(9) **trigger** broadcast $\langle$PROPOSE, $(B, H, r, PoLCR_i)_i\rangle$;
(10) **else**
(11) set $timerProposer$ to $TimeOutPropose$;
(12) **wait until** (($timerProposer$ expired) $\vee$ ($proposalReceived_i^{H,r'} \neq \perp$));
(13) **if** (($timerProposer$ expired) $\wedge$ ($proposalReceived_i^{H,r'} == \perp$)) **then**
(14) $TimeOutPropose \leftarrow TimeOutPropose + 1$;
(15) **endif**
(16) **endif**
Prevote step r
(17) **if** (($PoLCR_i \neq \perp$) $\wedge$ ($LLR_i \neq -1$) $\wedge$ ($LLR_i < PoLCR_i < r$)) **then**
(18) **wait until** $|prevotesReceived_i^{H,PoLCR}| > 2n/3$;
(19) **if** ($\exists B'$: (is23Maj($B', prevotesReceived_i^{H,PoLCR_i}$)) $\wedge$ ($B' \neq lockedBlock_i$)) **then** $lockedBlock_i \leftarrow nil$; **endif**
(20) **endif**
(21) **if** ($lockedBlock_i \neq nil$) **then trigger** broadcast $\langle$PREVOTE, $(lockedBlock_i, H, r)_i\rangle$;
(22) **else if** (isValid($proposalReceived_i^{H,r}$)) **then trigger** broadcast $\langle$PREVOTE, $(proposalReceived_i^{H,r}, H, r)_i\rangle$; **endif**
(23) **else trigger** broadcast $\langle$PREVOTE, $(nil, H, r)_i\rangle$;
(24) **endif**
(25) **wait until** ((is23Maj($nil, prevotesReceived_i^{H,r}$)) $\vee$ ($\exists B''$: (is23Maj($B'', prevotesReceived_i^{H,r}$)))$\vee$ ($|prevotesReceived_i^{H,r}| > 2n/3$)); %Delivery of any 2n/3 prevotes for the round r%
(26) **if** ($\neg$(is23Maj($nil, prevotesReceived_i^{H,r}$)) $\wedge$ $\neg$($\exists B''$: (is23Maj($B'', prevotesReceived_i^{H,r}$)))) **then**
(27) **set** $timerPrevote$ **to** $TimeOutPrevote$;
(28) **wait until** ($timerPrevote$ expired);
(29) **if** ($timerPrevote$ expired) **then** $TimeOutPrevote \leftarrow TimeOutPrevote + 1$; **endif**
Precommit step r
(30) **if** ($\exists B'$: (is23Maj($B', prevotesReceived_i^{H,r}$))) **then**
(31) $lockedBlock_i \leftarrow B'$;
(32) **trigger** broadcast $\langle$PRECOMMIT, $(B', H, r)_i\rangle$;
(33) $LLR_i \leftarrow r$;
(34) **else if** (is23Maj($nil, prevotesReceived_i^{H,r}$)) **then**
(35) $lockedBlock_i \leftarrow nil$; $LLR_i \leftarrow -1$;
(36) **trigger** broadcast $\langle$PRECOMMIT, $(nil, H, r)_i\rangle$;
(37) **endif**
(38) **else trigger** broadcast $\langle$PRECOMMIT, $(nil, H, r)_i\rangle$);
(39) **endif**
(40) **wait until** ((is23Maj($nil, prevotesReceived_i^{H,r}$)) $\vee$ ($|precommitsReceived_i^{H,r}| > 2n/3$))
endwhile

Figure 3 First part of Tendermint One-shot Consensus algorithm at correct process p_i.

step, validators broadcast their prevotes depending on the proposal they delivered during the previous step; and (iii) the *Precommit step*, validators broadcast their precommits depending on the occurrences of prevotes for the same block they delivered during the previous step. To preserve liveness, steps have a timeout associated, so that each validator moves from one step to another either if the timeout expires or if it delivers enough messages of a particular typology. When p_i broadcasts a message ($\langle TAG, m\rangle$), m contains a block B along with other information. We say that p_i prevotes (resp. precommits) on B if $TAG =$ `PREVOTE` (resp. TAG=`PRECOMMIT`). In Figure 1 is depicted the state machine for the Tendermint one-Shot Consensus.

To preserve safety of the protocol and to satisfy the Agreement property, when a validator delivers more than $2n/3$ prevotes for B then it "locks" on such block. Informally, it means that there are at least $n/3 + 1$ prevotes for B from correct processes, then B is a possible candidate for a decision so that validators try to stick on that. More formally, a validator



upon event delivery $\langle$PROPOSE, $(B', H, r', PoLCR_j)_j\rangle$:
(41) **if** $(proposalReceived_i^{H,r'} = \perp)$ **then**
(42) $\quad proposalReceived_i^{H,r'} \leftarrow (B', H, r')_j$;
(43) $\quad PoLCR_i \leftarrow PoLCR_j$;
(44) $\quad$ **trigger** broadcast $\langle$PROPOSE, $(B', H, r', PoLCR_j)_j\rangle$;
(45) **endif**

---

upon event delivery $\langle$PREVOTE, $(B', H, r', LLR)_j\rangle$:
(46) **if** $((B', H, r', LLR)_j \notin prevotesReceived_i^{H,r'})$ **then**
(47) $\quad prevotesReceived_i^{H,r'} \leftarrow prevotesReceived_i^{H,r'} \cup (B', H, r', LLR)_j$;
(48) $\quad$ **trigger** broadcast $\langle$PREVOTE, $(B', H, r', LLR)_j\rangle$;
(49) $\quad$ **if** $((r < r')$ **and** $(|prevotesReceived_i^{H,r'}| > 2/3))$ **then**
(50) $\quad\quad r \leftarrow r'$;
(51) $\quad\quad$ **goto** Prevote step r;
(52) $\quad$ **endif**
(53) **endif**

---

upon event delivery $\langle$PRECOMMIT, $(B', H, r')_j\rangle$:
(54) **if** $((B', H, r')_j \notin precommitsReceived_i^{H,r'})$ **then**
(55) $\quad precommitsReceived_i^{H,r'} \leftarrow precommitsReceived_i^{H,r'} \cup (B', H, r')_j$;
(56) $\quad$ **trigger** broadcast $\langle$PRECOMMIT, $(B', H, r')_j\rangle$;
(57) $\quad$ **if** $((r < r')$ **and** $(|precommitsReceived_i^{H,r'}| > 2/3))$ **then**
(58) $\quad\quad r \leftarrow r'$;
(59) $\quad\quad$ **goto** Precommit step r;
(60) $\quad$ **endif**
(61) **endif**

---

when $(\exists B'$: is23Maj$(B', precommitsReceived_i^{H,r'}))$:
(62) **return** B';%Terminate the consensus for the height H by deciding B'%

Figure 3 Second part of Tendermint One-shot Consensus algorithm at correct process p_i.

has a *Proof-of-LoCk* (PoLC) for a block B (resp. for *nil*) at a round r for the height H if it received at least $2n/3 + 1$ prevotes for B (resp. for *nil*). In this case we say that a process is locked on such block. A *PoLC-Round* ($PoLCR$) is a round such that there was a PoLC for a block at round $PoLCR$. In Figure 2 the state machine concerning the process of locking and unlocking on a block B is shown.

Preamble. Note that our analysis of the original Tendermint protocol [27, 7, 25] led to the conclusion that several modifications were needed in order to implement One-Shot Consensus problem. Full description of these bugs in the original Tendermint protocol are reported in [2]. In more details, with respect to the original Tendermint, our Tendermint One-shot Consensus algorithm (see Figure 3) has the following modifications. We added line 29 in order to catch up the communication delay during the synchronous periods. Moreover, we modified the line 19 in order to guarantee the agreement property of One-Shot Consensus (defined in Section 3). The correctness of Tendermint One-shot Consensus algorithm needs an additional assumption stating that eventually a proposal is accepted by a majority of correct processes. This assumption, stated formally in Theorem 7, is necessary to guarantee the termination.

Variables and data structures. r and $PoLCR_i$ are integers representing, respectively, the current round and the PolCR. $lockedBlock_i$ is the last block on which p_i is locked, if it is equal to a block B, we say that p_i is locked on B, otherwise it is equal to *nil*, and we say that p_i is not locked. When $lockedBlock_i \neq nil$ and switches the value to *nil*, then p_i unlocks. Last-Locked-Round (LLR_i) is an integer representing the last round where p_i locked on a block. B is the block the process created.

Each validator manages timeouts, *TimeOutPropose* and *TimeOutPrevote*, concerning the propose and prevote phases respectively. Those timeouts are set to Δ_{Propose} and Δ_{Prevote} and are started at the beginning of the respective step. Both are incremented if they expire before the validator moves to the next step.

Each validator manages three sets for the messages delivered. In particular, the set $proposalReceived_i^{H,r}$ contains the proposal that p_i delivered for the round r at height H. $prevotesReceived_i^{H,r}$ is the set containing all the prevotes p_i delivered for the round r at height H. $precommitsReceived_i^{H,r}$ is the set containing all the precommits p_i delivered for the round r at height H.

Functions. We denote by *Block* the set containing all blocks, and by *MemPool* the structure containing all the transactions.

- proposer : $V \times Height \times Round \rightarrow V$ is a deterministic function which gives the proposer out of the validators for a given round at a given height in a round robin fashion.
- createNewBlock : $2^{\Pi_\rho} \times MemPool \rightarrow Block$ is an application-dependent function which creates a valid block (w.r.t. the application), where the subset of processes is a parameter of the One-Shot Consensus, and is a subset of processes that send a commit for the block at the previous height, called the signature of the previous block.
- is23Maj : $(Block \cup nil) \times (prevotesReceived \cup precommitsReceived) \rightarrow \mathsf{Bool}$ is a predicate that checks if there is at least $2n/3 + 1$ of prevotes or precommits on the given block or *nil* in the given set.
- isValid : $Block \rightarrow \mathsf{Bool}$ is an application dependent predicate that is satisfied if the given block is valid. If there is a block B such that $\mathsf{isValid}(B) = \mathsf{true}$, we say that B is valid. We note that for any non-block, we set isValid to false, (e.g. $\mathsf{isValid}(nil) = \mathsf{false}$).

Detailed description of the algorithm. In Figure 3 we describe Tendermint One-Shot algorithm to solve the One-Shot Consensus (defined in Section 3) for a given height H.
For each round r at height H the algorithm proceeds in 3 phases:

1. Propose step (lines 5 - 16): If p_i is the proposer of the round and it is not locked on any block, then it creates a valid proposal and broadcasts it. Otherwise it broadcasts the block it is locked on. If p_i is not the proposer then it waits for the proposal from the proposer. p_i sets the timer to $TimeOutProposal$, if the timer expires before the delivery of the proposal then p_i increases the time-out, otherwise it stores the proposal in $proposalReceived_i^{H,r}$. In any case, p_i goes to the Prevote step.
2. Prevote step (lines 17 - 29): If p_i delivers the proposal during the Propose step, then it checks the data on the proposal. If $lockedBlock_i \neq nil$, and p_i delivers a proposal with a valid *PoLCR* then it unlocks. After that check, if p_i is still locked on a block, then it prevotes on $lockedBlock_i$; otherwise it checks if the block B in the proposal is valid or not, if B is valid, then it prevotes B, otherwise it prevotes on *nil*. Then p_i waits until $|prevotesReceived_i^{H,r}| > 2n/3$. If there is no PoLC for a block or for *nil* for the round r, then p_i sets the timer to $TimeOutPrevote$, waits for the timer's expiration and increases $TimeOutPrevote$. In any case, p_i goes to Precommit step.
3. Precommit step (lines 30 - 40): p_i checks if there was a PolC for a particular block or *nil* during the round (lines 30 and 34). There are three cases: (i) if there is a PoLC for a block B, then it locks on B, and precommits on B (lines 30 - 32); (ii) if there is a PoLC for *nil*, then it unlocks and precommits on *nil* (lines 34 - 36); (iii) otherwise, it precommits on *nil* (line 38); in any case, p_i waits until $|precommitsReceived_i^{H,r}| > 2n/3$ or $(\mathsf{is23Maj}(nil, prevotesReceived_i^{H,r}))$, and it goes to the next round.

Whenever p_i delivers a message, it broadcasts it (lines 44, 48 and 56). Moreover, during a round r, some conditions may be verified after a delivery of some messages and either (i) p_i decides and terminates or (ii) p_i goes to the round r' (with $r' > r$). The conditions are:

- For any round r', if for a block B, $\mathsf{is23Maj}(B, precommitsReceived_i^{H,r'}) = \mathsf{true}$, then p_i decides the block B and terminates, or
- If p_i is in the round r at height H and $|prevotesReceived_i^{H,r'}| > 2n/3$ where $r' > r$, then it goes to the Prevote step for the round r', or
- If p_i is in the round r at height H and $|precommitsReceived_i^{H,r'}| > 2n/3$ where $r' > r$, then it goes to the Precommit step for the round r'.

4.2.1 Correctness of Tendermint One-Shot Consensus

In this section we prove the correctness of Tendermint One-Shot Consensus algorithm (Fig. 3) for a height H under the assumption that during the synchronous period there exists eventually a proposer such that its proposed value will be accepted by at least $2n/3 + 1$ processes.

▶ **Lemma 1** (One-Shot Integrity)**.** *In an eventual synchronous system, Tendermint One-Shot Consensus Algorithm verifies the following property: No correct process decides twice.*

▶ **Lemma 2** (One-Shot Validity)**.** *In an eventual synchronous system, Tendermint One-Shot Consensus Algorithm verifies the following property: A decided value is valid, if it satisfies the predefined predicate denoted* $\mathit{isValid}()$*.*

▶ **Lemma 3.** *In an eventual synchronous system, Tendermint One-Shot Consensus Algorithm verifies the following property: If $f + 1$ correct processes locked on the same value B during a round r then no correct process can lock during round $r' > r$ on a value $B' \neq B$.*

Proof. We assume that $f + 1$ correct processes are locked on the same value B during the round r, and we denote by X^r the set of those processes. We first prove by induction that no process in X^r will unlock or lock on a new value. Let let $p_i \in X^r$.

- *Initialization*: round $r + 1$. At the beginning of round $r + 1$, all processes in X^r are locked on B. Moreover, we have that $LLR_i = r$, since p_i locks on round r (line 31). Let p_j be the proposer for round $r + 1$. If $LLR_j = r$, it means that p_j is also locked on B, since there cannot be a value $B' \neq B$ such that $\mathsf{is23Maj}(B, prevotesReceived_j^{H,r}) = \mathsf{true}$, for that to happen, at least $n/3$ processes should prevote both B and B' during round r, which means that at least a correct process prevoted two times in the same round, which is not possible, since it is correct, and the protocol does not allow to vote two times in the same round (lines 17 - 29). Three cases can then happen:
  - p_j locked on a value B_j during the round $LLR_j \leq r$. This means that during the round LLR_j $\mathsf{is23Maj}(B_j, prevotesReceived_j^{H,LLR_j}) = \mathsf{true}$ (line 31). p_j the proposer proposes a value B_j along with LLR_j (lines 5 - 9). Since $LLR_j \leq LLR_i = r$, p_i does not unlock and prevotes B for the round $r + 1$, and so are all the other processes in X^r (lines 17 - 21). The only value that can have more than $2n/3$ prevotes is then B. So p_i is still locked on B at the end of $r + 1$.
  - If p_j is not locked, the value it proposes cannot unlock processes in X^r because $-1 = LLR_j < r$, and they will prevote on B (lines 17 - 21). The only value that can have more than $2n/3$ prevotes is then B. So p_i is still locked on B at the end of $r + 1$.
  - p_j locked on a value B_j during the round $LLR_j > r$, p_j the proposer proposes a value B_j along with LLR_j (lines 5 - 9). Since $LLR_j \geq r + 1$, p_i does not unlock and prevotes

B for the round $r+1$, and so are all the other processes in X^r (lines 17 - 21). The only value that can have more than $2n/3$ prevotes is then B. So p_i is still locked on B at the end of $r+1$.

At the end of round $r+1$, all processes in X^r are still locked on B and it may happen that other processes are locked on B for round $r+1$ at the end of the round.

- *Induction*: We assume that for a given $a > 0$, the processes in X^r are still locked on B at each round between r and $r+a$. We now prove that the processes in X^r will still be locked on B at round $r+a+1$.

  Let p_j be the proposer for round $r+a+1$. Since the $f+1$ processes in X^r were locked on B for all the rounds between r and $r+a$, no new value can have more than $2n/3$ of prevotes during one of those rounds, so $\nexists B' \neq B : \mathsf{is23Maj}(B', prevotesReceived_j^{H,r_j}) = \mathsf{true}$ where $r < r_j < r+a+1$. Moreover, if p_j proposed the value B along with a $LLR > r$, since the processes in X^r are already locked on B, they do not unlock and prevote B (lines 17 - 21). The proof then follows as in the *Initialization* case.

Therefore all processes in X^r will stay locked on B at each round after round r. Since $f+1$ processes will stay locked on the value B on rounds $r' > r$, they will only prevote on B (lines 17 - 21) for each new round. Let B' be a value, we have that $\forall r' \geq r$ if $B' : \mathsf{is23Maj}(B', prevotesReceived_j^{H,r'}) = \mathsf{true}$ then $B' = B$. ◀

▶ **Lemma 4** (One-Shot Agreement). *In an eventual synchronous system, Tendermint One-Shot Consensus Algorithm verifies the following property: If there is a correct process that decides a value B, then eventually all the correct processes decide B.*

Proof. Let p_i be a correct process. Without loss of generality, we assume that p_i is the first correct process to decide, and it decides B at round r. If p_i decides B, then $\mathsf{is23Maj}(B, precommitsReceived_i^{H,r}) = \mathsf{true}$ (line 62), since the signature of the messages are unforgeable by hypothesis and $f < n/3$, then p_i delivers more than $n/3$ of those precommits for round r from correct processes, and those correct process are locked on B at round r (line 31). p_i broadcasts all the precommits it delivers (line 56), so eventually all correct processes will deliver those precommits, because of the best effort broadcast guarantees.

We now show that before delivering the precommits from p_i, the other correct processes cannot decide a different value than B. $f < n/3$ by hypothesis, so we have that at least $f+1$ correct processes are locked on B for the round r. By Lemma 3 no correct process can lock on a value different than B. Let $B' \neq B$, since correct processes lock only when they precommit (lines 30 - 32), no correct process will precommit on B' for a round bigger than r, so $\mathsf{is23Maj}(B', precommitsReceived_i^{H,r'}) = \mathsf{false}$ for all $r' \geq r$ since no correct process will precommit on B'. No correct process cannot decide a value $B' \neq B$ (line 62) once p_i decided. Eventually, all the correct processes will deliver the $2n/3$ signed precommits p_i delivered and broadcasted, thanks to the best effort broadcast guarantees and then will decide B. ◀

▶ **Lemma 5.** *In an eventual synchronous system, and under the assumption that during the synchronous period eventually there is a correct proposer p_k such that $|\{p_j : LLR_k \leq LLR_j \text{ and } p_j \text{ is correct}\}| < n/3 - f$, Tendermint One-Shot Consensus Algorithm verifies the following property: Eventually a correct process decides.*

Proof. Let r be the round where the communication becomes synchronous and when all the messages broadcasted by correct processes are delivered by the correct processes within their respective step. The round r exists, since the system is eventually synchronous and correct processes increase their time-outs when they did not deliver enough messages (lines 13 - 15,

26 - 29 and 40). If a correct process decides before r, that ends the proof. Otherwise no correct process decided yet. Let p_i be the proposer for the round r. We assume that p_i is correct. Let B be the value such that p_i proposes (B, LLR_i), we have three cases:

- Case 1: No correct process is locked on a value before r. $\forall p_j \in \Pi_\rho$ such that p_j is correct, $LLR_j = -1$.
  Correct processes delivered the proposal (B, LLR_i) before the Prevote step (lines 12, 42 - 44). Since the proposal is valid, then all correct processes will prevote on that value (line 22), and they deliver the others' prevotes and broadcast them before entering the Precommit step (lines 25 - 29 and 48). Then for all correct process p_j, we have $\mathsf{is23Maj}(B, precommitsReceived_j^{H,r}) = \mathsf{true}$. The correct processes will lock on B, precommit on B (lines 30 - 32) and will broadcast all precommits delivered (line 56). Eventually a correct process p_j will have $\mathsf{is23Maj}(B, precommitsReceived_j^{H,r}) = \mathsf{true}$ then p_j will decide (line 62).
- Case 2: Some correct processes are locked and if p_j is a correct process, $LLR_j < LLR_i$. Since $LLR_j < LLR_i$ for all correct processes p_j, then the correct processes that are locked will unlock (line 19) and the proof follows as in the Case 1.
- Case 3: Some correct processes are locked on a value, and there exist a correct process p_j such that $LLR_i \leq LLR_j$.
  - (i) If $|\{p_j : LLR_i \leq LLR_j \text{ and } p_j \text{ is correct}\}| < n/3 - f$ (which means that even without the correct processes that are locked in a higher round than the proposer p_i, there are more than $2n/3$ other correct processes unlock or locked in a smaller round than LLR_i), then as in the case 2, a correct process will decide.
  - (ii) If $|\{p_j : LLR_i \leq LLR_j \text{ and } p_j \text{ is correct}\}| \geq n/3 - f$, then during the round r, $\nexists B' : \mathsf{is23Maj}(B', precommitsReceived_i^{H,r}) = \mathsf{true}$, in fact correct processes only precommit once in a round (lines 30 - 40). Eventually, thanks to the additional assumption, there exists a round r_1 where the proposer p_k is correct and at round r_1, $|\{p_j : LLR_k \leq LLR_j \text{ and } p_j \text{ is correct}\}| < n/3 - f$. The proof then follows as case (3.i).

If p_i is Byzantine and more than $n/3$ correct processes delivered the same message during the proposal step, and the proposal is valid, the situation is like p_i was correct. Otherwise, there are not enough correct processes that delivered the proposal, or if the proposal is not valid, then there will be less than $n/3$ processes that will prevote that value. No value will be committed. Since the proposer is selected in a round robin fashion, a correct process will eventually be the proposer, and a correct process will decide. ◀

▶ **Lemma 6** (One-Shot Termination). *In an eventual synchronous system, and under the assumption that during the synchronous period eventually there is a correct proposer p_k such that $|\{p_j : LLR_k \leq LLR_j \text{ and } p_j \text{ is correct}\}| < n/3 - f$, Tendermint One-Shot Consensus Algorithm verifies the following property: Every correct process eventually decides some value.*

Proof. By construction, if a correct process does not deliver a proposal during the proposal step or enough prevotes during the Prevote step, then that process increases its time-outs (lines 13 - 15 and 26 - 29), so eventually, during the synchrony period of the system, all the correct processes will deliver the proposal and the prevotes from correct processes respectively during the Propose and the Prevote step. By Lemma 5, a correct process decides a value, and then by the Lemma 4, every correct process eventually decides. ◀

▶ **Theorem 7.** *In an eventual synchronous system, and under the assumption that during the synchronous period eventually there is a correct proposer p_k such that $|\{p_j : LLR_k \leq LLR_j \text{ and } p_j \text{ is correct}\}| < n/3 - f$: Tendermint One-Shot Algorithm implements the One-Shot Consensus.*

Function repeatedConsensus(Π_ρ); %Repeated Consensus for the set Π_ρ of processes%

Init:
(1) $H \leftarrow 1$ %$Height$%; $B \leftarrow \perp$; $V \leftarrow \perp$ %Set of validators%;
(2) $commitsReceived_i^H \leftarrow \emptyset$; $toReward_i^H \leftarrow \emptyset$; $TimeOutCommit \leftarrow \Delta_{\text{Commit}}$;

while (true) **do**
(3) $B \leftarrow \perp$;
(4) $V \leftarrow$ validatorSet(H); %Application and blockchain dependant%
(5) **if** ($p_i \in V$) **then**
(6) $B \leftarrow$ consensus($H, V, toReward_i^{H-1}$); %Consensus function for the height H%
(7) **trigger** broadcast $\langle$COMMIT, $(B, H)_i\rangle$;
(8) **else**
(9) **wait until** ($\exists B'$: |atLeastOneThird($B', commitsReceived_i^H$)|);
(10) $B \leftarrow B'$;
(11) **endif**
(12) set $timerCommit$ to $TimeOutCommit$;
(13) **wait until**($timerCommit$ expired);
(14) **trigger** decide(B);
(15) $H \leftarrow H + 1$;
endwhile

upon event delivery $\langle$COMMIT, $(B', H')_j\rangle$:
(16) **if** $(((B', H')_j \notin commitsReceived_i^{H'}) \wedge (p_j \in$ validatorSet$(H')))$ **then**
(17) $commitsReceived_i^{H'} \leftarrow commitsReceived_i^{H'} \cup (B', H')_j$;
(18) $toReward_i^{H'} \leftarrow toReward_i^{H'} \cup p_j$;
(19) **trigger** broadcast $\langle$COMMIT, $(B', H')_j\rangle$;
(20) **endif**

Figure 4 Tendermint Repeated Consensus algorithm at correct process p_i.

Proof. The proof follows directly from Lemmas 1, 2, 4 and 6. ◀

4.3 Tendermint Repeated Consensus algorithm

For a given height, the set V of validators does not change. Note that each height corresponds to a block. Therefore, in the following we refer this set as the set of validators for a block.

Data structures. The integer H is the height where is called a One-Shot Consensus instance. V is the current set of validators. B is the block to be appended. $commitsReceived_i^H$ is the set containing all the commits p_i delivered for the height H. $toReward_i^H$ is the set containing the validators from which p_i delivered commits for the height H. $TimeOutCommit$ represents the time a process has for collecting commits after an instance of consensus. $TimeOutCommit$ is set to Δ_{Commit}.

Functions.
- validatorSet : $\Pi_\rho \times Height \rightarrow 2^{\Pi_\rho}$ is an application dependent and deterministic selection function which gives the set of validators for a given height w.r.t the blockchain history, and $\forall H \in Height, |validatorSet(H)| = n$.
- consensus : $Height \times 2^{\Pi_\rho} \times commitsReceived \rightarrow Block$ is the One-Shot Consensus instance presented in 4.2.
- atLeastOneThird : $Block \times commitsReceived \rightarrow$ Bool is a predicate which checks if there is at least $n/3$ of commits of the given block in the given set.

Detailed description of the algorithm. In Fig. 4 we describe the algorithm to solve the Repeated Consensus as defined in Section 3. The algorithm proceeds as follows:
- p_i computes the set of validators for the current height;
- If p_i is a validator, then it calls the consensus function solving the consensus for the current height, then broadcasts the decision, and sets B to that decision;

- Otherwise, if p_i is not a validator, it waits for at least $n/3$ commits from the same block and sets B to that block;
- In any case, it sets the timer to *TimeOutCommit* to receive more commits and lets it expire. Then p_i decides B and goes to the next height. We note that this timer is not adjustable, so processes might miss commits from correct processes even during the synchronous period (see fairness study [2] for further details).

Whenever p_i delivers a commit, it broadcasts it (lines 16 - 20). Note that the reward for the height H is given during the height $H + 1$, and to a subset of validators who committed the block for H (line 6).

▶ **Theorem 8.** *In an eventual synchronous system, Tendermint Repeated Consensus algorithm implements the Repeated Consensus.*

5 Conclusion & Discussion

The contribution of this paper is the improvement and the formal analysis of the original Tendermint protocol, a PBFT-based repeated consensus protocol where the set of validators is dynamic. Each improvement we introduced is motivated by bugs we discovered in the original protocol. A preliminary version of this paper has been reported in [2]. Very recently a new version of Tendermint has been advertised in [6] by Tendermint foundation without an operational release. The authors argue that their solution works if the two hypothesis below are verified: *Hypothesis 1*: if a correct process receives some message m at time t, all correct processes will receive m before $max(t, global\ stabilization\ time) + \Delta$. Note that this property called by the authors *gossip communication* should be verified even though m has been sent by a Byzantine process. *Hypothesis 2*: there exists eventually a proposer such that its proposed value will be accepted by all the other correct processes. Moreover, the formal and complete correctness proof of this new protocol is still an open issue (several not trivial bugs have been reported recently e.g. [34]).

We are further interested in the *fairness* of Tendermint-core blockchains since without a glimpse of fairness in the way rewards are distributed, these blockchains may collapse. It is common knowledge that in permisionless blockchain systems the main threat is the tragedy of commons that may yield the system to collapse if the rewarding mechanism is not adequate. Ad minimum the rewarding mechanism must be $fair$, i.e. distributing the rewards in proportion to the merit of participants. Our fairness preliminary study, reported in [2], is in line with Francez definition of fairness [18], generally defines the fairness of protocols based on voting committees (e.g. Byzcoin[24], PeerCensus[11], RedBelly [9], SBFT [21] and Hyperledger Fabric [3] etc), by the fairness of their *selection mechanism* and the fairness of their *reward mechanism*. The selection mechanism is in charge of selecting the subset of processes that will participate to the agreement on the next block to be appended to the blockchain, while the reward mechanism defines the way the rewards are distributed among processes that participate in the agreement. Our preliminary analysis of the reward mechanism allowed to establish the following result with respect to the fairness of repeated-consensus blockchains as follows:

There exists a(n) (eventual) fair reward mechanism for repeated-consensus blockchains if and only if the system is (eventual) synchronous (see [2]).

It follows that in our fairness model the original Tendermint protocol is not eventually fair, however with a small twist in the way delays are handled its reward mechanism becomes eventually fair. More details can be found in [2]. Our study opens an interesting future research direction related to the fairness of the selection mechanism in repeated-consensus based blockchains.

References

1. Marcos K Aguilera. A pleasant stroll through the land of infinitely many creatures. *ACM Sigact News*, 35(2):36–59, 2004.
2. Yackolley Amoussou-Guenou, Antonella Del Pozzo, Maria Potop-Butucaru, and Sara Tucci Piergiovanni. Correctness and Fairness of Tendermint-core Blockchains. *CoRR*, abs/1805.08429, 2018.
3. Elli Androulaki, Artem Barger, Vita Bortnikov, Christian Cachin, Konstantinos Christidis, Angelo De Caro, David Enyeart, Christopher Ferris, Gennady Laventman, Yacov Manevich, Srinivasan Muralidharan, Chet Murthy, Binh Nguyen, Manish Sethi, Gari Singh, Keith Smith, Alessandro Sorniotti, Chrysoula Stathakopoulou, Marko Vukolic, Sharon Weed Cocco, and Jason Yellick. Hyperledger Fabric: A Distributed Operating System for Permissioned Blockchains. In *Proceedings of the Thirteenth EuroSys Conference, EuroSys 2018, Porto, Portugal, April 23-26, 2018*, pages 30:1–30:15, 2018.
4. Roberto Baldoni, Marin Bertier, Michel Raynal, and Sara Tucci-Piergiovanni. Looking for a definition of dynamic distributed systems. In *International Conference on Parallel Computing Technologies*, pages 1–14. Springer, 2007.
5. Amotz Bar-Noy, Xiaotie Deng, Juan A. Garay, and Tiko Kameda. Optimal Amortized Distributed Consensus. *Inf. Comput.*, 120(1):93–100, 1995.
6. E. Buchman, J. Kwon, and Z. Milosevic. The latest gossip on BFT consensus. *CoRR*, abs/1807.04938v1, July 2018. URL: `https://arxiv.org/abs/1807.04938v1`.
7. Ethan Buchman. *Tendermint: Byzantine Fault Tolerance in the Age of Blockchains*. Thesis, University of Guelph, June 2016. URL: `https://atrium.lib.uoguelph.ca/xmlui/handle/10214/9769`.
8. Miguel Castro and Barbara Liskov. Practical Byzantine Fault Tolerance and Proactive Recovery. *ACM Trans. Comput. Syst.*, 20(4):398–461, November 2002.
9. Tyler Crain, Vincent Gramoli, Mikel Larrea, and Michel Raynal. DBFT: Efficient Byzantine Consensus with a Weak Coordinator and its Application to Consortium Blockchains, 2017.
10. Daian, Rafael Pass, and Elaine Shi. Snow White: Provably Secure Proofs of Stake. *IACR Cryptology ePrint Archive*, 2016:919, 2016.
11. Christian Decker, Jochen Seidel, and Roger Wattenhofer. Bitcoin meets strong consistency. In *Proceedings of the 17th International Conference on Distributed Computing and Networking, Singapore, January 4-7, 2016*, pages 13:1–13:10, 2016.
12. Carole Delporte-Gallet, Stéphane Devismes, Hugues Fauconnier, Franck Petit, and Sam Toueg. With Finite Memory Consensus Is Easier Than Reliable Broadcast. In *Principles of Distributed Systems, 12th International Conference, OPODIS 2008, Luxor, Egypt, December 15-18, 2008. Proceedings*, pages 41–57, 2008.
13. Shlomi Dolev and Sergio Rajsbaum. Stability of long-lived consensus. *J. Comput. Syst. Sci.*, 67(1):26–45, 2003.
14. Cynthia Dwork, Nancy A. Lynch, and Larry J. Stockmeyer. Consensus in the presence of partial synchrony. *J. ACM*, 35(2):288–323, 1988.
15. Cynthia Dwork and Moni Naor. Pricing via Processing or Combatting Junk Mail. In *Advances in Cryptology - CRYPTO '92, 12th Annual International Cryptology Conference, Santa Barbara, California, USA, August 16-20, 1992, Proceedings*, pages 139–147, 1992.
16. Ittay Eyal, Adem Efe Gencer, Emin Gün Sirer, and Robbert van Renesse. Bitcoin-NG: A Scalable Blockchain Protocol. In *13th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2016, Santa Clara, CA, USA, March 16-18, 2016*, 2016.
17. M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of Distributed Consensus with One Faulty Process. *Journal of the ACM*, 32(2), April 1985.
18. Nissim Francez. *Fairness*. Texts and Monographs in Computer Science. Springer, 1986.

19 J. A. Garay, A. Kiayias, and N. Leonardos. The Bitcoin Backbone Protocol: Analysis and Applications. In *Proc. of the EUROCRYPT International Conference*, 2015.

20 Yossi Gilad, Rotem Hemo, Silvio Micali, Georgios Vlachos, and Nickolai Zeldovich. Algorand: Scaling Byzantine Agreements for Cryptocurrencies. In *Proceedings of the 26th Symposium on Operating Systems Principles, Shanghai, China, October 28-31, 2017*, pages 51–68, 2017.

21 Guy Golan-Gueta, Ittai Abraham, Shelly Grossman, Dahlia Malkhi, Benny Pinkas, Michael K. Reiter, Dragos-Adrian Seredinschi, Orr Tamir, and Alin Tomescu. SBFT: a scalable decentralized trust infrastructure for blockchains. *CoRR*, abs/1804.01626, 2018.

22 Maurice Herlihy and Mark Moir. Enhancing Accountability and Trust in Distributed Ledgers. *CoRR*, abs/1606.07490, 2016.

23 Aggelos Kiayias, Alexander Russell, Bernardo David, and Roman Oliynykov. Ouroboros: A Provably Secure Proof-of-Stake Blockchain Protocol. In *Advances in Cryptology - CRYPTO 2017 - 37th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 20-24, 2017, Proceedings, Part I*, pages 357–388, 2017.

24 E. Kokoris-Kogias, P. Jovanovic, N. Gailly, I. Khoffi, L. Gasser, and B. Ford. Enhancing Bitcoin Security and Performance with Strong Consistency via Collective Signing. In *Proceedings of the 25th USENIX Security Symposium*, 2016.

25 Jae Kwon. Tendermint: Consensus without mining. Technical report, Tendermint, 2014.

26 Jae Kwon and Ethan Buchman. Cosmos: A Network of Distributed Ledgers. `https://cosmos.network/resources/whitepaper` (visited on 2018-05-22).

27 Jae Kwon and Ethan Buchman. Tendermint. `https://tendermint.readthedocs.io/projects/tools/en/v0.19.3/specification.html` (visited on 2018-05-22).

28 Dahlia Malkhi. The BFT Lens: Tendermint. `https://dahliamalkhi.wordpress.com/2018/04/03/tendermint-in-the-lens-of-bft/` (visited on 2018-05-22), April 2018.

29 Francesc D. Muñoz-Escoí and Rubén de Juan-Marín. On synchrony in dynamic distributed systems. *Open Computer Science*, 8(1):154–164, 2018. `doi:10.1515/comp-2018-0014`.

30 S. Nakamoto. Bitcoin: A Peer-to-Peer Electronic Cash System. `https://bitcoin.org/bitcoin.pdf` (visited on 2018-05-22), 2008.

31 Rafael Pass, Lior Seeman, and Abhi Shelat. Analysis of the Blockchain Protocol in Asynchronous Networks. In *Advances in Cryptology - EUROCRYPT 2017 - 36th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Paris, France, April 30 - May 4, 2017, Proceedings, Part II*, pages 643–673, 2017.

32 Rafael Pass and Elaine Shi. The Sleepy Model of Consensus. In *Advances in Cryptology - ASIACRYPT 2017 - 23rd International Conference on the Theory and Applications of Cryptology and Information Security, Hong Kong, China, December 3-7, 2017, Proceedings, Part II*, pages 380–409, 2017.

33 M. Pease, R. Shostak, and L. Lamport. Reaching Agreement in the Presence of Faults. *Journal of the ACM*, 27(2):228–234, April 1980.

34 Tendermint. Tendermint: correctness issues. `https://github.com/tendermint/spec/issues` (see issues 36-37 visited on 2018-09-05).

35 Tendermint. Tendermint: Tendermint Core (BFT Consensus) in Go. `https://github.com/tendermint/tendermint/blob/e88f74bb9bb9edb9c311f256037fcca217b45ab6/consensus/state.go` (visited on 2018-05-22).

36 G. Wood. Ethereum: A secure decentralised generalised transaction ledger. `http://gavwood.com/Paper.pdf` (visited on 2018-05-22).