Correctness and Fairness of Tendermint-core Blockchains

Yackolley Amoussou-Guenou, Antonella del Pozzo, Maria Potop-Butucaru, Sara Tucci-Piergiovanni

Correctness and Fairness of Tendermint-core Blockchains

Yackolley Amoussou-Guenou[‡,⋆], Antonella Del Pozzo[‡],
Maria Potop-Butucaru[⋆], Sara Tucci-Piergiovanni[‡]

[‡]CEA LIST, PC 174, Gif-sur-Yvette, 91191, France
[⋆]Sorbonne Université, CNRS, Laboratoire d'Informatique de Paris 6, LIP6, Paris, 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. However, the exact assumptions on the system model under which Tendermint underlying algorithms are correct and the exact properties Tendermint verifies have never been formally analyzed. The contribution of this paper is two-fold. First, while formalizing Tendermint algorithms we precisely characterize the system model and the exact problem solved by Tendermint. 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. Our second contribution relates to the fairness of the rewarding mechanism. 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. We prove, for the first time in blockchain systems, that in repeated-consensus based blockchains there exists an (eventual) fair rewarding mechanism if and only if the system is (eventual) synchronous. We also show that the original Tendermint rewarding is not fair, however, a modification of the original protocol makes it eventually fair.

1 Introduction

Blockchain is today one of the most appealing technologies since its introduction in the Bitcoin White Paper [37] in 2008. Blockchain systems, similar to P2P systems in the early 2000, take their roots in the non academical research. After the releasing of the most popular blockchains (e.g. Bitcoin [37] or Ethereum [43]) 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 [18]. 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. 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 that 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 [10] became recently popular not only for in-chain transaction systems but also in systems that provide cross-chain transactions. Tendermint [34, 8, 32, 35] 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 [34, 8, 32] 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 [33] for cross-chain transactions.

In this paper we analyse the correctness of the original Tendermint agreement protocol as it was described in [34, 8, 32] and discussed in [35, 28]. The code of this protocol is available in [42]. 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 11).

More in detail, we prove that the original Tendermint 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.

We are further interested in the *fairness* of Tendermint-core blockchains because without a glimpse of fairness in the way rewards are distributed, these blockchains may collapse. Our fairness study, in line with Francez definition of fairness [23], generally defines the fairness of protocols based on voting committees (e.g. Byzcoin[31], PeerCensus[14], RedBelly [12], SBFT [25] 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. The analysis of the reward mechanism allowed to establish our second fundamental 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 (Theorem 16).

Moreover, we show that even in an eventual synchronous setting, the original Tendermint protocol is not eventually fair, however with a small twist in the way delays and commit messages are handled it becomes eventually fair.

Note that our work is the first to analyze the fairness of protocols based on voting committees elected by selection mechanisms as *proof-of-stake*.

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 protocol through pseudo-code and proves the correctness of the One-Shot Consensus anf the Repeated Consensus algorithms. In Section 5 we present a full descriptions of the counter-example that motivates the modification of the original algorithm and the additioinal assumptions needed for the correctness. Section 6 discusses the necessary and sufficient conditions for a protocol based on repeated consensus to achieve a fair rewarding.

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 [22].

In distributed systems, the theoretical studies of *proof-of-work* based blockchains have been pioneered by Garay et *al* [24]. 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.* [38] to round based systems where messages sent in a round can be received later. In order to overcome the drawbacks of Bitcoin, [20] proposes a mix between proof-of-work blockchains and proof-of-work free blockchains referred as Bitcoin-NG. Bitcoin-NG inherits 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 Decker *et al.* [14] 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 and Byzcoin against dynamic adversaries.

Byzcoin [31] builds on top of *Practical Byzantine Fault-Tolerant* consensus [10] enhanced with a scalable collective signing process. [31] 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 [25] and Hyperledger Fabric [3] build on top of [10].

In order to avoid some of the previously cited problems, Micali [36] introduced (further extended in [6, 11]) *sortition* based blockchains, where the proof-of-work mechanism is completely replaced by a probabilistic ingredient.

The only academic work that addresses the consensus in *proof-of-stake* based blockchains is authored by Daian *et al.* [13], which proposes a protocol for weakly synchronous networks. The

execution of the protocol is organized in epochs. Similar to Bitcoin-NG [20] 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 [29] proposes a sortition based proof-of-stake protocol and addresses mainly the security aspects of the proposed protocol. Red Belly [12] focuses on consortium blockchains, where only a predefined subset of processes are allowed to append blocks, and proposes a Byzantine consensus protocol.

Interestingly, none of the previous academic studies made the connection between the repeated consensus specification [5, 16, 15] 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.

The closest works in blockchain systems to our fairness study (however very different in its scope) study the *chain-quality*. In [24], Garay *et al.* define the notion of *chain-quality* as the proportion of blocks mined by honest miners in any given window and study the conditions under which the ratio of blocks in the chain mined by malicious players over the total number of blocks in a given window is bounded. Kiayias *et al.* in [29] propose Ouroboros [29] and also analyse the chain-quality property. Pass *et al.* address in [39] one of the vulnerabilities of Bitcoin studied formally in Eyal and Sirer [21]. In [21] the authors prove that if the adversary controls a coalition of miners holding even a minority of the computational power, this coalition can gain twice its share. Fruitchain [39] overcomes this problem by ensuring that no coalition controlling less than a majority of the computational power can gain more than a factor $1 + 3\delta$ by not respecting the protocol, where δ is a parameter of the protocol. In [19], Eyal analyses the consequences of attacks in systems where pools of miners can infiltrate each other and shows that in such systems there is an equilibrium where all pools earn less than if there were no attack. In [26], Guerraoui and Wang study the effect of message propagation delays in Bitcoin and show that, in a system of two miners, a miner can take advantage of the delays and be rewarded exponentially more than its expectation. In [27], Gürcan *et al.* study the fairness from the point of view of users that do not participate to the mining. A similar work is done by Herlihy and Moir in [28] where the authors study the users fairness and consider as an example the original Tendermint [34, 8, 32]. The authors discussed how processes with malicious behaviour can violate fairness by choosing transactions, then they propose modifications to the original Tendermint to make those violations detectable and accountable.

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 time 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 [33] that is in charge of implementing the selection of V.

Communication network. The processes communicate by exchanging messages through an eventually synchronous network [17]. *Eventually Synchronous* means that after a finite unknown time τ there is a bound δ on the message transfer delay.

Failure model. There is no bound on processes that can exhibit a Byzantine behaviour [40] 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 transition, 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 the same content, while they should send messages with distinct 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* or *honest*. 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 $\mathsf{broadcast}(\langle TAG, m\rangle)$, 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 $\mathsf{broadcast}()$ 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 $\mathsf{delivery}()$. 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 [4].

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

Definition 3.1 (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.
- **Agreement.** If there is a correct process that decides a value B, then eventually all the correct processes decide B.
- **Validity[12].** A decided value is valid, it satisfies the predefined predicate denoted $\mathsf{isValid}()$.

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 [16] 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 [15], 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 [15]. 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 3.2 (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 [34, 8] 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 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 [33] 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. Let us recall that Tendermint agreement protocol relies on the assumption that the selection mechanism, for each block, selects up to f Byzantine processes from the current history.

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.

Although the choice of validators is managed by a separate module (see Cosmos project [33]) 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.

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 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 2 is depicted the state machine for Tendermint Consensus.

To preserve safety, 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 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 3 the state machine concerning the process of locking and unlocking on a block B is shown. On the edges are reported the conditions on the delivered messages that have to be met in order to lock or unlock on a block. Intuitively, when a process delivers $2n/3$ of the same message B of type prevote when in a precommit step, it locks on B. A process unlocks a block only if it delivers $2n/3$ of a value B' or when it commits. When a process is locked on a block B, it does not send any value different than B. This mechanism is necessary to ensure the safety of the protocol and to satisfy the Agreement property stated in Section 3.

Preamble. Note that our analysis of the original Tendermint protocol [34, 8, 32] led to the conclusion that several modifications are needed in order to implement One-Shot Consensus problem. Full description of these bugs in the original Tendermint protocol are reported in Section 5. In more details, with respect to the original Tendermint, our Tendermint One-shot Consensus algorithm (see Figure 1) 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 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

Function consensus$(H, \Pi, \text{signature})$; %One-Shot Consensus for the super-round H with the set Π of processes%

Init:
(1) $r \leftarrow 0$; $LLR_i \leftarrow -1$; $PoLCR_i \leftarrow \perp$; $lockedBlock_i \leftarrow nil$; $B \leftarrow \perp$;
(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** $((\text{is23Maj}(nil, prevotesReceived_i^{H,r})) \vee (\exists B'' : (\text{is23Maj}(B'', prevotesReceived_i^{H,r}))) \vee$
$(|prevotesReceived_i^{H,r}| > 2n/3))$; %Delivery of any 2n/3 prevotes for the round r%
(26) **if** $(\neg(\text{is23Maj}(nil, prevotesReceived_i^{H,r})) \wedge \neg(\exists B'' : (\text{is23Maj}(B'', prevotesReceived_i^{H,r}))))$ **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** $((\text{is23Maj}(nil, prevotesReceived_i^{H,r})) \vee (|precommitsReceived_i^{H,r}| > 2n/3))$
endwhile

Figure 1: First part of Tendermint One-shot Consensus algorithm at correct process p_i.

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 switch the value to *nil*, then p_i unlocks. Last-Locked-Round (LLR_i) is an

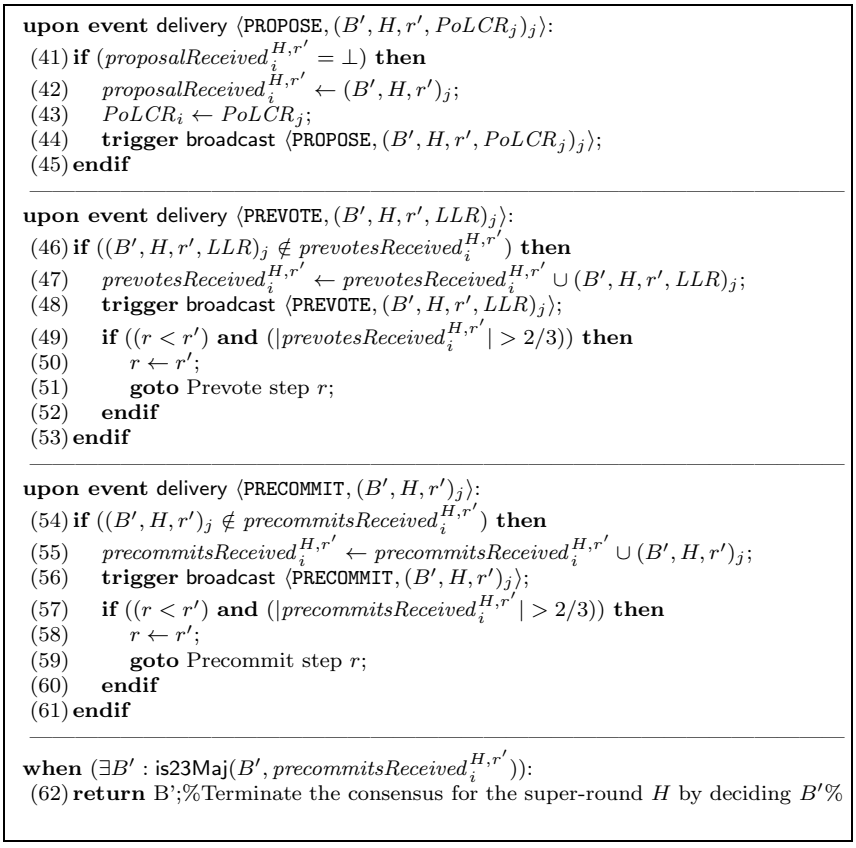

Figure 1: Second part of Tendermint One-shot Consensus algorithm at correct process p_i.

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 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 : $2^{\Pi} \times MemPool \rightarrow Block$ 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 the subset of processes that send a commit for the block at the previous height, called the signature of the previous block.

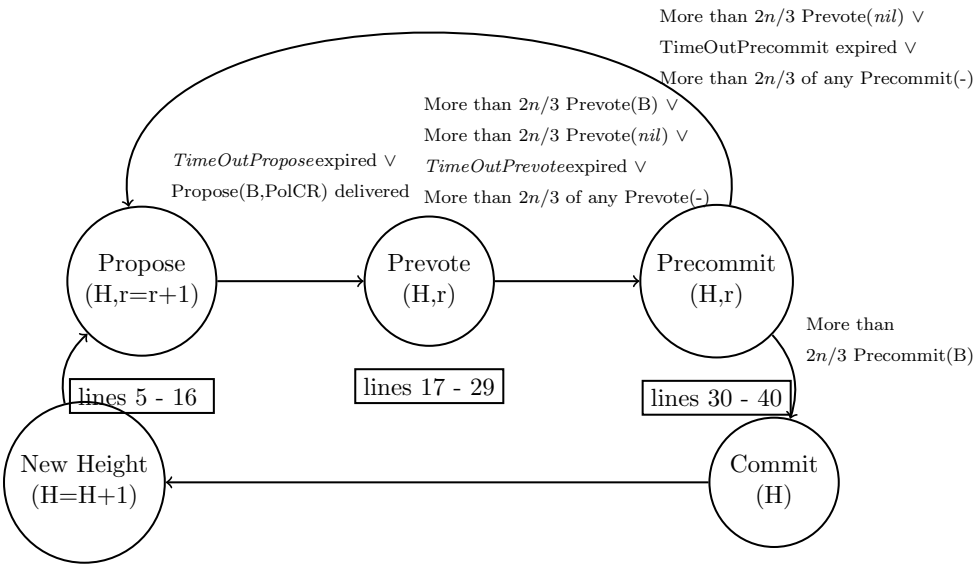

Figure 2: State Machine for Tendermint One-Shot algorithm described in Figure 1. For ease of readability, common exit conditions are represented.

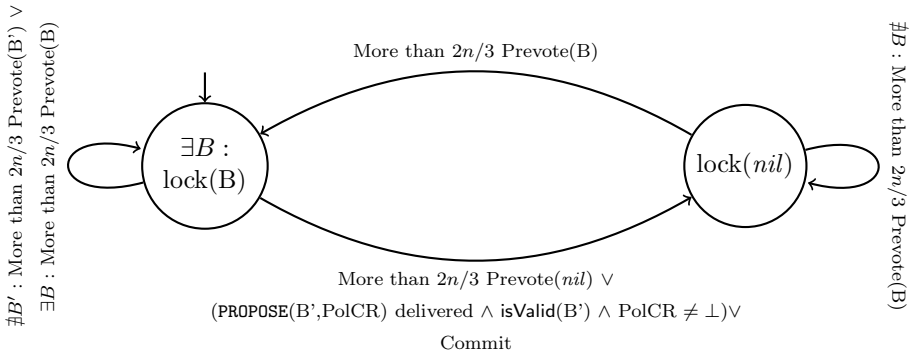

Figure 3: State machine Lock/Unlock

- $\mathsf{is23Maj} : (Block \cup nil) \times (prevotesReceived \cup precommitsReceived) \to \mathsf{Bool}$ 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.
- $\mathsf{isValid} : Block \to \mathsf{Bool}$ 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. $\mathsf{isValid}(\bot) = \mathsf{false}$).

Detailed description of the algorithm. In Figure 1 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'.

4.3 Correctness of Tendermint One-Shot Consensus

In this section we prove the correctness of Tendermint One-Shot Consensus algorithm (Fig. 1) 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.

Proof The proof follows by construction. A correct process decides when it returns (line 62). $\square_{Lemma\ 1}$

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 isValid().

Proof

Let p_i be a correct process, we assume that there exists a value B, such that p_i decides B. We show by construction that if p_i decides on a value B, then B is valid.

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 and $f < n/3$ by hypothesis, then more than $n/3$ of those precommits for round r are from correct processes. This means that for each of those correct processes p_j, $\mathsf{is23Maj}(B, prevotesReceived_j^{H,r}) = \mathsf{true}$ (lines 30 - 32), and thus at least $n/3$ of those prevotes are from correct processes.

Let p_j be one of the correct processes which prevoted on B during the round r. p_j prevotes on a value B during a round r in two cases: Case a, during r, p_j is not locked on any value or Case b: p_j is already locked on B and does not checks its validity (lines 21 - 24).

- Case a: If p_j is not locked on any value than before prevoting it checks the validity of B and prevotes on B if B is valid (line 22);
- Case b: If p_j was locked on B, it did not check the validity of B, it means that p_j was locked on B during the round r; which means that there was a round $r' < r$ such that p_j had a PoLC for B for the round r' (lines 30 - 32), by the same argument, there is a round which happened before r' where p_j was locked or B is valid.

  Since a process locked during a round smaller than the current one, and that there exists a first round where all correct processes are not locked (line 1), there is a round $r'' < r'$ where p_j was not locked on B but prevoted B, as in Case a, p_j checks if B is valid and then prevotes on B if B is valid (lines 22).

A value prevoted by a correct process is thus valid. Therefore a decided value by a correct process is valid since more than $n/3$ correct processes prevote that value. $\square_{Lemma\ 2}$

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 $B' = B$.

$\square_{Lemma\ 3}$

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. $\square_{Lemma\ 4}$

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. $\forall p_j \in \Pi$ 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. $\square_{Lemma\ 5}$

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. $\square_{Lemma\ 6}$

Theorem 7. In an eventual synchronous system, and under the assumption that during the synchronous periods 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.

Proof The proof follows directly from Lemmas 1, 2, 4 and 6. $\square_{Theorem\ 7}$

4.4 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 validator has for collecting commits after an instance of consensus. *TimeOutCommit* is set to Δ_{Commit}.

Functions.

- validatorSet : $\Pi \times Height \rightarrow 2^{\Pi}$ is an application dependent and deterministic selection function which gives the set of validators for a given height w.r.t the blockchain history. We have $\forall H \in Height, |validatorSet(H)| = n$.

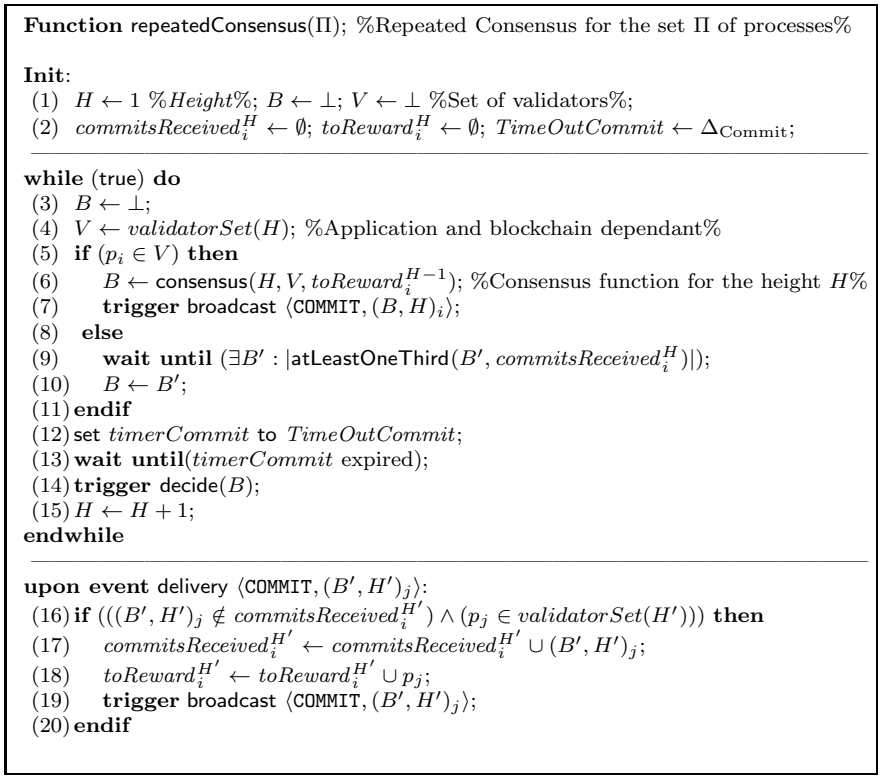

Figure 4: Tendermint Repeated Consensus algorithm at correct process p_i.

- consensus : $Height \times 2^{\Pi} \times commitsReceived \rightarrow Block$ is the One-Shot Consensus instance presented in 4.2.
- createNewBlock : $2^{\Pi} \times MemPool \rightarrow Block$ is the application-dependent function that creates a valid block (w.r.t. the application) from the One-Shot Consensus.
- 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.
- isValid : $Block \rightarrow$ Bool is the same predicate as in the One-shot Consensus, which checks if a block is valid or not.

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* for receiving more commits and lets it expire. Then p_i decides B and goes to the next height.

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).

4.5 Correctness of Tendermint Repeated Consensus

In this section we prove the correctness of Tendermint Repeated Consensus algorithm in Figure 4. We now show that Tendermint Repeated Algorithm (Fig. 4) implements the Repeated Consensus.

Lemma 8 (Repeated Termination)**.** In an eventual synchronous system, and under the additional 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 Repeated Consensus Algorithm verifies the following property: Every correct process has an infinite output.

Proof By contradiction, let p_i be a correct process, and we assume that p_i has a finite output. Two scenarios are possible, either p_i cannot go to a new height, or from a certain height H it outputs only $\perp$.

- If p_i cannot progress, one of the following cases is satisfied:
  - The function consensus() does not terminate (line 6), which is a contradiction due to Lemma 6; or
  - p_i waits an infinite time for receiving enough commits (line 9), which cannot be the case because of the best effort broadcast guarantees and the eventual synchronous assumption, all the correct validators terminate the One-Shot Consensus and broadcast their commit.
- If p_i decides at each height (line 14), it means that from a certain height H, p_i only outputs $\perp$. That means that: (i) either p_i is a validator for H and the function consensus(H') is only returning $\perp$ for all $H' \geq H$ (lines 5 and 6), or (ii) p_i is not a validator for H but delivered at least $n/3$ commits for $\perp$ (lines 9 and 19).
  - (i): Since consensus() returns the value $\perp$, that means by Lemma 2 that isValid($\perp$) = true, which is a contradiction with the definition of the function isValid().
  - (ii): Since only the validators commit, and each of them broadcasts its commit (lines 5 - 7), and because $f < n/3$, it means that p_i delivered a commit from at least one correct validator (process). By Lemma 2, correct processes only decide/commit on valid value, and $\perp$ is not valid, which is a contradiction.

We conclude that if p_i is a correct process, then it has an infinite output. $\square_{Lemma\ 8}$

Lemma 9 (Repeated Agreement)**.** In an eventual synchronous system, Tendermint Repeated Consensus Algorithm verifies the following property: 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.

Proof We prove this lemma by construction. Let p_j and p_k be two correct processes. Two cases are possible:

- p_j and p_k are validators for the height i, so they call the function consensus() (lines 5 and 6). By Lemma 4 p_j and p_k decide the same value and then output that same value (line 14).
- At least one of p_j and p_k is not a validator for the height i. Without loss of generality, we assume that p_j is not a validator for the height i. Since all the correct validators commit the same value, let say B, thanks to Lemma 4, and since they broadcast their commit (line 7), eventually there will be more than $2n/3$ of commits for B. So no other value $B' \neq B$ can be present at least $n/3$ times in the set $commitReceived_i^H$. So p_j outputs the same value B as all the correct validators (line 9). If p_k is a validator, that ends the proof. If p_k is not a validator, then by the same argument as for p_j, p_k outputs the same value B. Hence p_j and p_k both output the same value B.

$\square_{Lemma\ 9}$

Lemma 10 (Repeated Validity)**.** In an eventual synchronous system, Tendermint Repeated Consensus Algorithm verifies the following property: Each value in the output of any correct process is valid, it satisfies the predefined predicate denoted isValid().

Proof We prove this lemma by construction. Let p_i be a correct process, and we assume that the H^{th} value of the output of p_i is B. If p_i decides a value (line 14), then that value has been set during the execution and for that height (line 3).

- If p_i is a validator for the height H, then B is the value returned by the function consensus(), by the Lemma 2 we have that isValid(B) = true.
- If p_j is not a validator for the height H, it means that it delivered more than $n/3$ signed commits from the validators for the value B (lines 5 - 7 and 16 - 20), hence at least one correct validator committed B, and by Lemma 2 we have that isValid(B) = true.

So each value that a correct process outputs satisfies the predicate isValid(). $\square_{Lemma\ 10}$

Theorem 11. In an eventual synchronous system, Tendermint Repeated Consensus algorithm implements the Repeated Consensus. In an eventual synchronous system, and under the additional 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 Repeated Consensus algorithm implements the repeated consensus.

Proof The proof follows directly from Lemmas 8, 9 and 10. we showed that Tendermint protocol satisfies respectively the Termination property, the Agreement property and the Validity property.

$\square_{Theorem\ 11}$

5 Bugs in the original Tendermint

5.1 Addition of line 29 on Fig. 1

The line 29 allows the correct process p_i to increase its time-out to catch up the communication delay in the network. If the correct processes never increase their *TimeOutPrevote*, even when the

system becomes synchronous, the correct process may never deliver enough prevotes at time. Thus it can never precommit. To decide, a correct process needs more than $2n/3$ precommits for the same value for the same round (and so more that $n/3$ precommits from correct processes) to decide (line 62), no correct process ever decides, which does not satisfies the One-Shot Termination property. When a process increases its *TimeOutPrevote*whenever it does not deliver enough messages, it will eventually catch-up the delay, and during the synchronous period, there will be a time from when it will deliver prevotes from all correct processes.

5.2 Modification of line 19 on Fig. 1

Originally, the line 19 was

$$\textbf{if}(\exists B' : (\mathsf{is23Maj}(B', prevotesReceived_i^{H,PoLCR_i})) \textbf{ then } lockedBlock_i \leftarrow nil; \textbf{endif}$$

In that version, it is possible that if a process p_i is locked on a value B during a round r, a process locked on the same value B during a round $r' > r$ makes p_i unlock, but does not ensure that p_i locks again. That is a problem since it causes a violation of the Agreement property.

In the following, we exhibit a problematic scenario. Assume that there are 4 processes in the network. 3 correct processes p_1, p_2, p_3 and a Byzantine process p_4.

1. Round 1: p_1 is the proposer and proposes B. All process deliver the proposal and prevote on B from round 1.

   p_1 and p_2 deliver all prevotes for B, and then lock and precommit on B.

   p_1 delivers the precommit of p_1, p_2 and from p_4 for $B$and then it decides B. Neither p_2 and p_3 delivers enough precommit to decide.

   The state is: p_1 decides B and left. p_2 is locked on $(B, 1)$, p_3 is not locked, p_4 is Byzantine so we do not say anything about its state.

2. Round 2: p_1 exit, and do not take part any more. p_2 and p_3 do not deliver the precommit for B.

   p_2 is the proposer. Since p_2 is locked on B, it proposes B along with 1 where it locked. p_2 and p_3 deliver the proposal. p_2 does not unlock since it locked at round 1 and prevote on B. p_3 is not locked, but since it delivered the proposal, it prevotes on B. p_4 sends a prevote on B only to p_3 such that p_3 delivers all the prevotes during this step but not p_2, so during the precommit step, p_3 locked on B but for the round 2.

   The state is: p_1 already decided. p_2 is locked on $(B, 1)$, p_3 is locked on $(B, 2)$, p_4 is Byzantine so we do not say anything about its state.

3. Round 3: p_2 and p_3 do not deliver the precommit for B from round 1.

   p_3 is the proposer. Since p_3 is locked on B, it proposes B along with 2 where it locked. p_2 and p_3 deliver the proposal. p_2 unlocks since it receives 2 but was locked at 1 and prevote the proposal B. p_3 does not unlock since it locked exactly at round 1 and prevote on B. p_4 sends a prevote on B such that p_3 delivers all the prevotes during its step but not p_2, so during the precommit step, p_3 locked on B but for the round 2.

   The state is: p_1 already decided B. p_2 is not locked, p_3 is locked on $(B, 2)$, p_4 is Byzantine so we do not say anything about its state.

4. Round 4: p_2 and p_3 do not deliver the precommit for B from round 1.

   p_4 is the proposer and proposes $(B, 3)$. p_2 is not locked, but since it delivered the proposal, it prevotes on B. p_3 unlocks since it receives 3 but was locked at 2 and prevote the proposal B. p_4 does nothing. Neither p_2 nor p_3 delivers enough prevotes to lock.

   The state is: p_1 already decided B. p_2 is not locked, p_3 is not locked, p_4 is Byzantine so we do not say anything about its state.

5. Round 5: p_2 and p_3 do not deliver the precommit for B from round 1.

   p_1 is the proposer but since it left, there is no proposal. p_2 and p_3 are not locked and did not deliver a proposal so they prevote on *nil*. p_4 does nothing. Neither p_2 nor p_3 delivers enough prevotes to lock.

   The state is: p_1 already decided B. p_2 is not locked, p_3 is not locked, p_4 is Byzantine so we do not say anything about its state.

6. Round 4: p_2 and p_3 do not deliver the precommit for B from round 1.

   p_2 is the proposer. Since it is not locked, it proposes a new value B'. p_2 and p_3 deliver the proposal before their respective prevote step. p_2 and p_3 are not locked and did deliver the proposal B', so they prevote on B'. p_4 sends prevote B' to both p_2 and p_3.

   p_2 and p_3 deliver the 3 prevotes before entering the precommit step, and hence locked on B' and precommit on B, and p_4 also send precommit to p_2.

   p_2 delivers precommit from p_2, p_3 and p_4 and thus it decides B'.

   The state is: p_1 already decided B. p_2 decides B', p_3 is locked on B', p_4 is Byzantine so we do not say anything about its state.

At the end of the round 4, p_1 and p_2 decide on two different values, which does not satisfy the One-Shot Agreement property.

5.3 Counter-example for the One-Shot Termination without the additional assumption

We recall the additional assumption : eventually there is a correct proposer p_k such that $|\{p_j : LLR_k \leq LLR_j \text{ and } p_j \text{ is correct}\}| < n/3 - f$.

In [9] the authors advocate without providing any evidence that there is a livelock problem in Tendermint description proposed in the Buchman's manuscript [8]. Hereafter, we exhibit this evidence. We thank anonymous reviewers of the preliminary version of this work appeared as technical report [2] to point us the scenario below.

We consider a system of 4 processes, p_1 to p_4, where p_4 is a Byzantine process. The round number 1 has p_1 as a proposer, and we assume that it happens before the system is synchronous (before GST in DLS terminology [17]), and only p_1 locks value v_1 in this round ($lockedBlock = v_1, LLR = 1$ at p_1, and for other processes equal to initial values. $PoLCR = \perp$ for p_1, p_2, and p_3 (p_4 is faulty so we don't talk about it's state).

Starting with round number 2, all rounds happen during synchronous period (after GST), so communication between correct processes is reliable and timely, i.e., all correct processes receive messages from all correct processes. Note that during synchronous period we don't have the same

guarantee on messages sent by Byzantine processes, i.e., a Byzantine process can send a message only to a subset of correct processes on time so it is delivered in the current round, and although a message is eventually delivered by all correct processes, it might not be delivered by all correct processes in the round r in which it is sent.

1. In round 2, p_2 proposes $(v_2, \perp)$, where $B = v_2$, and $PoLCR = \perp$ as p_2 hasn't locked any value. p_1 rejects this proposal and Prevote v_1 as it has v_1 locked in round 1 (condition at line 17 evaluates to false as upon receipt of Proposal all correct processes set $PoLCR$ to $\perp$). p_2 and p_3 accept the proposal and prevote v_2, but as p_4 stay silent, no process locks a value in round 2. So at the end of round 2, we have the following state: $lockedBlock = v_1, LLR = 1$ at p_1, and for other processes equal to initial values ($lockedBlock = nil$ and $LLR = \perp$) $PoLCR = \perp$ at p_1, p_2, and p_3 (p_4 is faulty so we don't talk about it's state).

2. In round 3, p_3 proposes $(v_3, \perp)$ as it hasn't locked any value. p_1 rejects this proposal and prevote v_1 as it has v_1 locked in round 1 (condition at line 17 evaluates to false as upon receipt of Proposal all correct processes set $PoLCR$ to $\perp$). p_2 and p_3 accept the proposal and prevote v_3, but p_4 sends Prevote message for v_3 only to p_3. Furthermore, p_4 sends Prevote v_3 message to p_3 just before $timerPrevote$ expires at p_3, and after $timerPrevote$ expired at p_1 and p_2, and they moved to round 4. So although Prevote messages that are received by p_3 are propagated to other processes, they will be received by p_1 and p_2 after they moved to round 4. So in the round 3 only p_3 locks v_3. At the end of the round 3, $lockedBlock = v_1, LLR = 1$ at p_1, $lockedBlock = v_3, LLR = 3$ at p_3, $lockedBlock = nil, LLR = \perp$ at p_2, and $PoLCR = \perp$ for p_1, p_2, and p_3 (p_4 is faulty so we don't talk about it's state).

3. In round 4, p_4 is a proposer and as it is Byzantine process we assume it stays silent, so nothing change.

4. In round 5, p_1 proposes $(v_1, 1)$. p_3 rejects this proposal and prevote v_3 as it has v_3 locked in round 3, therefore the condition at line 17 evaluates to false as $LLR = 3$ and $PoLCR = 1$ at p_3. p_1 and p_2 accept the proposal, and prevote v_1. p_4 sends Prevote message for v_1 only to p_1 just before $timerPrevote$ expires at p_1 and after $timerPrevote$ expires at p_2 and p_3. So p_2 and p_3 receives Prevote from p_4 from round 5 only after they moved to round 6. So in the round 5 only p_1 locks v_1. At the end of the round 5, $lockedBlock = v_1, LLR = 5$ at p_1, $lockedBlock = v_3, LLR = 3$ at p_3, $lockedBlock = nil, LLR = \perp$ at p_2, and $PoLCR = 1$ for p_1, p_2, and p_3 (p_4 is faulty so we don't talk about it's state).

5. In round 6, p_2 proposes $(v_2, 1)$ as it hasn't locked any value. p_1 rejects this proposal and prevote v_1 as it has v_1 locked in round 5. p_3 rejects the proposal and prevote v_3 as it has locked v_3 in round 3, and p_4 stays silent, so no process lock a value in round 2. At the end of the round 6, $lockedBlock = v_1, LLR = 5$ at p_1, $lockedBlock = v_3, LLR = 3$ at p_3, $lockedBlock = nil, LLR = \perp$ at p_2, and $PoLCR = 1$ for p_1, p_2, and p_3 (p_4 is faulty so we don't talk about it's state).

6. In round 7, p_3 proposes $(v_3, 3)$. p_1 rejects this proposal and prevote v_1 as it has v_1 locked in round 5. p_2 and p_3 accept the proposal and prevote v_3, and p_4 sends Prevote message for v_3 only to p_3, just before $timerPrevote$ expires at p_3 and after $timerPrevote$ expires at p_1 and p_2, and after they moved to round 8. So similar as above in the round 7 only p_3 locks v_3. At the end of the round 7, $lockedBlock = v_1, LLR = 5$ at p_1, $lockedBlock = v_3, LLR = 7$ at p_3,

$lockedBlock = nil, LLR = \perp$ at p_2, and $PoLCR = 3$ for p_1, p_2, and p_3 (p_4 is faulty so we don't talk about it's state).

This scenario repeats forever, so the algorithm never terminates, which violates the One-Shot Termination property. This scenario cannot repeat forever with the additional assumption that during the synchronous period eventually there is a correct proposer p_k such that $|\{p_j : LLR_k \leq LLR_j \text{ and } p_j \text{ is correct}\}| < n/3 - f$.

Let us call that assumption assumption $\mathcal{T}$.

Assumption $\mathcal{T}$ vs. Symmetric Byzantine : Symmetric Byzantine processes [30] are processes that behave arbitrarily, but their behaviour is perceived the same by all correct processes. We note that in our case; the assumption of Symmetric Byzantine is stronger than the assumption $\mathcal{T}$. In fact the Symmetric Byzantine assumption restrict the Byzantine behaviour during the whole execution of the algorithm, whereas the assumption $\mathcal{T}$ requires that eventually, during an interval of one round of the execution after the synchronous period, Byzantine processes behaviour do not impact at least $2n/3$ correct processes.

6 Tendermint Fairness

Recently Pass and Shi defined the fairness of a Proof-of-Work based blockchain protocol for a system of n processes as follows (please refer to [39] for the formal definition): *A blockchain protocol is fair if honest process that wield ϕ fraction of the computational resources will reap at least ϕ fraction of the blocks in any sufficiently long window of the chain*, where the computational resources represent the merit of the process. We note that in their model, a block in the blockchain was created by only one process, and that process gets a reward for the created block. We extend the definition of [39] for a system with an infinite number of processes, and where each block is produced by a subset of processes. This is the case of Tendermint for example where for each block there is a subset of processes called the *validators* that produce that block. The *correct validators* for a block (those that followed the protocol and participated in the agreement process) are the processes that have to be rewarded for that block. Informally, we say that a blockchain protocol is fair if any *correct process* (a process that followed the protocol) that wield ϕ fraction of the total merit in the system will get more or less ϕ fraction of the total reward that is given in the system. In order to study the fairness of a protocol in a consensus-based blockchain such as Tendermint, Redbelly, SBFT or Hyperledger Fabric, we split the protocol in two mechanisms: (i) the *selection mechanism* which selects for each new height the *validators* (the processes that will run the consensus instance) for that height taking into account the merit of each process, and (ii) the *reward mechanism*, which is the mechanism giving rewards to correct validators that decided on the new block. Informally, if the selection mechanism is fair, then every process will become validator proportionally to its merit parameter; and if the reward mechanism is fair then for each height only the correct validators get a reward. By combining the two mechanisms, a correct process gets rewarded at least a number proportional to its merit parameter, since the faulty processes do not get any reward.

We define the following properties for characterizing the fairness of a reward mechanism. For each height, each validator has a boolean variable which we call a *reward parameter*.

1. For each block in the blockchain, all correct validators for that block have a reward parameter equal to 1,

2. For each block in the blockchain, all faulty validators and the processes that are not validators should have a reward parameter for that block equal to 0.

3. A process gets a reward for a block if and only if it has a reward parameter for that block equal to 1.

3'. There exists a height H such that for a block in the blockchain at height $H' > H$ a process gets a reward for that block if and only if it has a reward parameter for that block equal to 1.

Definition 6.1 (Fairness of a reward mechanism)**.** A reward mechanism is *fair* if it satisfies the conditions 1, 2 and 3.

Definition 6.2 (Eventual fairness of a reward mechanism)**.** A reward mechanism is *eventually fair* if it satisfies the conditions 1, 2 and 3'.

Definition 6.3 ((Eventual) Fairness of a blockchain protocol)**.** A blockchain protocol is *fair* (resp. *eventually fair*) if it has a fair selection mechanism and a fair (resp. eventually fair) reward mechanism.

6.1 Tendermint's Reward Mechanism

The validators selection mechanism is part of a separate module of Cosmos project [33], which has Tendermint as core-blockchain. The selection mechanism is today left configurable by the application, therefore in the following we do not address this part. The rewarding mechanism, on the other hand, referred as the *Tendermint's reward mechanism* is part of the original Tendermint protocol and it is reported in Figure 4 at lines 13 and 16 - 20. Tendermint's reward mechanism works as follows:

- Once a new block is decided for height H, processes wait for *TimeOutCommit* time to collect the decision from the other validators for H, and put them in their set *toReward* (Fig. 4, lines 13 and 16 - 20).

- During the consensus at height H, let us assume that p_i proposes the block that will get decided in the consensus. p_i proposes to reward processes in its set *toReward* (Fig. 4, line 6). That is, only the processes from which p_i delivered a commit will get a reward for the block at height $H - 1$.

Lemma 12. The reward mechanism of Tendermint is not eventually fair.

Proof We assume that the system becomes synchronous, and that $TimeOutCommit < \Delta$, where Δ is the maximum message delay in the network. For any height H, let p_i be a validator for the height $H-1$ and p_j the validator whose proposal get decided for the height H. It may happen that p_j did not receive the commit from p_i before proposing its block. Hence when the block is decided, p_i does not get a reward for its effort, which contradicts the condition 3' of the reward mechanism fairness. Tendermint's reward mechanism is not eventually fair. $\square_{Lemma\ 12}$

Let us observe that to make Tendermint's reward mechanism at least eventually fair it is necessary to increase *TimeOutCommit* for each round until it catches up the message delay. We refer to this variant as the *Tendermint's reward mechanism with modulable timeouts.* Moreover, the commit

message should contain enough information to keep track of process participation in each phase, in order to exclude from the reward a process that did not send a propose or vote message but that sends a commit because he is aware about the block produced. This scenario can be avoided, for instance, by including in the *toReward* variable a process p_i only if $f+1$ commit messages contain the process p_i.

Lemma 13. When the commit messages are sufficient to detect a process that did not send expected messages, Tendermint's reward mechanism with modulable timeouts is eventually fair.

Proof We change the reward mechanism in Tendermint as follows:

- Once a new block is decided, say for height H, processes wait for at most *TimeOutCommit* to collect the decision from the other validators for that height, and put them in their set *toReward*.
- If a process did not get the commits from all the validators for that height before the expiration of the time-out, it increases the time-out for the next height.
- During the consensus at height H, let us assume that p_i proposes the block that will get decided in the consensus. p_i gives the reward to the processes in its *toReward*.

In this reward mechanism, *TimeOutCommit* is increased whenever a process does not have the time to deliver all the commits for the previous round. We prove that this reward mechanism is eventually fair.

There is a point in time t from when the system will become synchronous, and all the commits will be delivered by correct processes before the next height. From the time t, at height H all correct processes know the exact set of validators that committed the block from $H-1$, and from those commit messages, they can exclude the set of processes that did not participate to the consensus, and they give to those validators from $H-1$ a reward parameter greater than 0. The validators in H give the reward to the correct validators that committed and which are the only one with a reward parameter greater than 0 for $H-1$, which satisfy the fairness conditions 1, 2 and 3', so the reward mechanism presented is eventually fair. $\square_{Lemma\ 13}$

Theorem 14. In an eventual synchronous system, if the selection mechanism is fair, and from the commit messages processes can differentiate correct and Byzantine behaviour, then Tendermint Repeated Consensus with the reward mechanism with modulable time-outs, is eventually fair.

Proof The proof follows by Lemma 13. $\square_{Theorem\ 14}$

6.2 Necessary and Sufficient Conditions for a Fair Reward Mechanism

In this section, we discuss the consequences of the synchrony on the existence of a fair reward in Repeated Consensus based blockchain protocols where the blockchain is constructed by a mechanism of repeated consensus. That is, at each height, a subset of processes called validators produce a block executing an instance of One-Shot Consensus.

Theorem 15. There exists a fair reward mechanism in a Repeated Consensus based blockchain protocol iff the system is synchronous.

Proof We prove this theorem by double implication.

- If the system is synchronous, then there exists a fair reward mechanism.

  We assume that the system is synchronous and all messages are delivered before the x following blocks. We consider the following reward mechanism. For all the correct validators at any height H, if $H - x \leq 0$, do not reward yet, otherwise:

  - Set to 1 the reward parameters of all correct validators in $H - x$, and to 0 the merit parameters of the others.
  - Reward only the validators with a reward parameter equal to 1 from the height $H - x$.

  We prove in the following that the above reward mechanism is fair.

  Note that the system is synchronous and messages sent are delivered within at most x blocks. Therefore, the exact set of correct validators at $H - x$ is known by all at H. By construction, the validators in H exactly give the reward to the correct validators who are the only one with a reward parameter for $H - x$ greater than 0, which satisfies the fairness conditions (1, 2 and 3).

- If there exists a fair reward mechanism, then the system is synchronous.

  By contradiction, we assume that $\mathcal{P}$ is a protocol having a fair reward mechanism and that, the system is not synchronous. We say that the validators following $\mathcal{P}$ are the correct validators. Let V^i be a set of validators for the height i, and $V^j, (j > i)$ be the set of validators who gave the reward to the correct validators in V^i. Since the system is not synchronous, the validators in V^j may not receive all messages from V^i before giving the reward.

  By conditions 1, and 2, it follows that all and only the correct validators in V^i have a reward parameter greater than 0. Since the reward mechanism is fair, with the condition 3, we have the validators in V^j gave the reward only to the correct validators in V^i. That means that the correct validators in V^j know exactly who were the correct validators in V^i, so they got all the messages before giving the reward. Contradiction, which conclude the proof.

$\square_{Theorem\ 15}$

If there is no synchrony, then there cannot be a fair consensus based protocol for blockchain. The fairness we define states that every time during the execution, the system is fair, so if a process leaves the system, it receives all rewards it deserves for the time it was in the system.

Theorem 16. There exists an eventual fair reward mechanism in a Repeated Consensus based blockchain protocol iff the system is eventually synchronous or synchronous.

Proof We proof this result by double implication.

- If the system is eventually synchronous or synchronous, then there exists an eventual fair reward mechanism.

  If the system is synchronous, the proof follows directly from theorem 15. Otherwise, we prove that the following reward mechanism is eventually fair. When starting the height H, the correct validator for H do the following:

  - Start the time-out for the reception of the messages from validators for the block $H - 1$;

– Wait for receiving the messages from validators for the block $H-1$ or the time-out to expire. If the time-out expires before the reception of more that $2n/3$ of the messages, increase the time-out for the next time;
– Set to 1 the reward parameters for $H-1$ of the correct validators which messages where received, and to 0 the merit parameters of the others;
– Reward only the validators for the height $H-1$ which have a merit parameter different than 0.

Since the system is eventually synchronous, eventually when the system will become synchronous, the processes, in particular the validators for H will receive messages for all the correct validators from round $H-1$ before the round H. We note that the condition 3' is a weaker form of the condition 3 where we do not consider the beginning. So we end the proof by applying the theorem 15 from the time when the system becomes synchronous.

- If there exists an eventual fair reward mechanism, then system is eventually synchronous or synchronous.

  If the reward mechanism is fair, by theorem 15, the communication is synchronous, which ends the proof. Otherwise, since the reward mechanism is eventually fair, then there is a point in time t from when all the rewards are correctly distributed. By considering t as the beginning of our execution, then we have that the reward mechanism is fair after t, so by the theorem 15, the system is synchronous from t. If the system were not synchronous before t, that means that it is eventually synchronous, otherwise, it is synchronous. Which ends the proof.

$\square_{Theorem\ 16}$

Corollary 16.1. In an asynchronous system, there is no (eventual) fair reward mechanism, so if the communication system is asynchronous, then there is no (eventual) fair Repeated Consensus-based Blockchain protocol. Note that this result is valid even if all the processes are correct.

7 Conclusion

The first 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 introduce is motivated by bugs we discover 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 [7] 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. [41]).

Our second major contribution is the study of the fairness of the reward mechanism in repeated-consensus blockchains. We proved that there exists a reward mechanism in repeated-consensus

blockchains that is (eventually) fair if and only if the system communication is (eventually) synchronous. In addition, we show that even if Tendermint protocol evolves in an eventual synchronous setting, it is not eventually fair. However, it becomes eventually fair when timeouts are carefully tuned, and under the assumption that commit messages contains enough information to distinguish between correct and Byzantine processes in the synchronous period. 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/1805.08429v1, 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] Iddo Bentov, Rafael Pass, and Elaine Shi. The sleepy model of consensus. *IACR Cryptology ePrint Archive*, 2016:918, 2016.

[7] E. Buchman, J. Kwon, and Z. Milosevic. The latest gossip on BFT consensus. *CoRR*, abs/1807.04938v1, jul 2018. URL: `https://arxiv.org/abs/1807.04938v1`.

[8] 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`.

[9] Christian Cachin and Marko Vukolić. Blockchains consensus protocols in the wild. *arXiv preprint arXiv:1707.01873*, 2017.

[10] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance and proactive recovery. *ACM Trans. Comput. Syst.*, 20(4):398–461, November 2002.

[11] Jing Chen and S. Micali. Algorand. *arXiv preprint arXiv:1607.01341*, 2017.

[12] T. Crain, V. Gramoli, M. Larrea, and M. Raynal. (Leader/Randomization/Signature)-free Byzantine Consensus for Consortium Blockchains. `http://csrg.redbellyblockchain.io/doc/ConsensusRedBellyBlockchain.pdf` (visited on 2018-05-22), 2017.

[13] Daian, Rafael Pass, and Elaine Shi. Snow white: Provably secure proofs of stake. *IACR Cryptology ePrint Archive*, 2016:919, 2016.

[14] C. Decker, J. Seidel, and R. Wattenhofer. Bitcoin Meets Strong Consistency. In *Proceedings of the 17th International Conference on Distributed Computing and Networking Conference (ICDCN)*, 2016.

[15] 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.

[16] Shlomi Dolev and Sergio Rajsbaum. Stability of long-lived consensus. *J. Comput. Syst. Sci.*, 67(1):26–45, 2003.

[17] Cynthia Dwork, Nancy A. Lynch, and Larry J. Stockmeyer. Consensus in the presence of partial synchrony. *J. ACM*, 35(2):288–323, 1988.

[18] 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.

[19] Ittay Eyal. The miner's dilemma. In *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*, pages 89–103, 2015.

[20] 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*, pages 45–59, 2016.

[21] Ittay Eyal and Emin Gün Sirer. Majority is not enough: Bitcoin mining is vulnerable. In *Financial Cryptography and Data Security - 18th International Conference, FC 2014, Christ Church, Barbados, March 3-7, 2014, Revised Selected Papers*, pages 436–454, 2014.

[22] 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.

[23] Nissim Francez. *Fairness*. Texts and Monographs in Computer Science. Springer, 1986.

[24] J. A. Garay, A. Kiayias, and N. Leonardos. The bitcoin backbone protocol: Analysis and applications. In *Proc. of the EUROCRYPT International Conference*, 2015.

[25] 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.

[26] Rachid Guerraoui and Jingjing Wang. On the unfairness of blockchain. In *NETYS 2018*, 2018.

[27] Önder Gürcan, Antonella Del Pozzo, and Sara Tucci Piergiovanni. On the bitcoin limitations to deliver fairness to users. In *On the Move to Meaningful Internet Systems. OTM 2017 Conferences - Confederated International Conferences: CoopIS, C&TC, and ODBASE 2017, Rhodes, Greece, October 23-27, 2017, Proceedings, Part I*, pages 589–606, 2017.

[28] Maurice Herlihy and Mark Moir. Enhancing accountability and trust in distributed ledgers. *CoRR*, abs/1606.07490, 2016.

[29] 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.

[30] Roger M. Kieckhafer and Mohammad H. Azadmanesh. Reaching approximate agreement with mixed-mode faults. *IEEE Trans. Parallel Distrib. Syst.*, 5(1):53–63, 1994.

[31] 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.

[32] Jae Kwon. Tendermint: Consensus without mining. Technical report, Tendermint, 2014.

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

[34] Jae Kwon and Ethan Buchman. Tendermint. `https://tendermint.readthedocs.io/projects/tools/en/v` (visited on 2018-05-22).

[35] Dahlia Malkhi. The BFT lens: Tendermint. `https://dahliamalkhi.wordpress.com/2018/04/03/tendermi` (visited on 2018-05-22), apr 2018.

[36] Silvio Micali. Algorand: The efficient and democratic ledger. *arXiv preprint arXiv:1607.01341*, 2016.

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

[38] 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.

[39] Rafael Pass and Elaine Shi. Fruitchains: A fair blockchain. In *Proceedings of the ACM Symposium on Principles of Distributed Computing, PODC 2017, Washington, DC, USA, July 25-27, 2017*, pages 315–324, 2017.

[40] M. Pease, R. Shostak, and L. Lamport. Reaching agreement in the presence of faults. *Journal of the ACM*, 27(2):228–234, April 1980.

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

[42] Tendermint. Tendermint: Tendermint Core (BFT Consensus) in Go. `https://github.com/tendermint/tendermint/blob/e88f74bb9bb9edb9c311f256037fcca217b45ab6/cons` (visited on 2018-05-22).

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