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 offer strong consistency (no forks) in an open system relying on two ingredients (i) a set of validators that generate blocks via a variant of Practical Byzantine Fault Tolerant (PBFT) consensus protocol and (ii) a rewarding mechanism that dynamically selects nodes to be validators for the next block via proof-of-stake, a non-energy consuming alternative of proof-of-work. It is well-known that in those open systems the main threat is the tragedy of commons that may yield the system to collapse if the rewarding mechanism is not adequate. At minima the rewarding mechanism must be $fair$, i.e. distributing the rewards in proportion to the merit of participants. The contribution of this paper is two-fold. First, we provide a formal description of Tendermint-core protocol and we prove that in eventual synchronous systems (i) it verifies 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. Our second contribution relates to the fairness of Tendermint rewarding mechanism. We prove that Tendermint rewarding is not fair. However, a small twist in the protocol makes it eventually fair. Additionally, we prove that there exists an (eventual) fair rewarding mechanism in repeated consensus-based blockchains if and only if the system is (eventually) synchronous.

1 Introduction

Blockchain is today one of the most appealing technologies since its introduction in the BitCoin White Paper [30] in 2008. Blockchains 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 [30] or Ethereum [34]) 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 others 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 [13]. The block that will be appended to the blockchain is owed 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 [6] became recently popular not only for in-chain transaction systems but also in systems that provide cross-chain transactions. Tendermint [28] was the first in this line of research having the merit to link the *Practical Byzantine Fault-Tolerant* consensus to the proof-of-stake technique and to propose a blockchain where a dynamic set of validators (subset of the participants) decide on the next block to be appended to the blockchain. Although, the correctness of Tendermint has never been formally analyzed from the distributed computing perspective, Tendermint protocol or slightly modified variants became recently the core of several popular systems such as Casper [26] for in-chain transactions or Cosmos [27] for cross-chain transactions.

In this paper we analyse the correctness of Tendermint agreement protocol. One of the fundamental results is as follows.

Theorem 1.1. In an eventual synchronous system Tendermint agreement protocol verifies the one-shot and repeated consensus specifications provided that the number of Byzantine validators, f, is $f < n/3$ where n is the number of validators participating in one-shot consensus.

We are further interested in the *fairness* of the rewarding mechanism in Tendermint blockchains. As explained previously, without a glimpse of fairness a protocol based on voting committees may collapse. Our fairness study is in line with Francez definition of fairness [19]. We define the fairness of a protocol based on voting committees (e.g. Tendermint, Byzcoin, PeerCensus, RedBelly) by the fairness of its *selection mechanism* and the fairness of its *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 is how the rewards are distributed among processes that participate in appending a new block. Our fundamental result related to the fairness of rewarding in repeated-consensus blockchains is as follows:

Theorem 1.2. There is a (eventual) fair rewarding mechanism for repeated-consensus blockchains if and only if the system is (eventual) synchronous.

A direct corollary is that Tendermint protocol is not fair, however with a small twist in the way delays are handled in Tendermint its reward mechanism becomes eventually fair.

The rest of the paper is organized as follows. Section 2 discuss esrelated works. Section 3 defines the model and the formal specifications of one shot and repeated consensus. Section 4 formalizes the Tendermint protocol and analyzes its correctness in eventual synchronous systems. 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 scientists focus their attention on the theoretical aspects of blockchains motivated mainly by the intriguing claim of popular blockchains including Bitcoin and Ethereum that they implement consensus in an asynchronous dynamic open system. This claim is refuted by the famous impossibility result in distributing computing [17].

In distributed systems, the theoretical studies of *proof-of-work* based blockchains have been pioneered by Garay et *al* [20]. Garay *et al.* decorticate the pseudo-code of Bitcoin and analyse its agreement aspects considering a synchronous round-based communication model. This study has been extended by Pass *et al.* [31] to round based systems where messages sent in a round can be received later. In order to overcome the drawbacks of Bitcoin, [15] proposes a mix between proof-of-work blockchains and proof-of-work free blockchains referred as Bitcoin-NG. The idea is that the execution of the system is organized in epochs. In each epoch a leader elected via a proof-of-work mechanism will decide the order transactions will be committed in the blockchain till the next epoch. 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 do not change the history at will if it is corrupted.

On another line of research *Decker et al.* [10] propose 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 of the same drawbacks as Bitcoin and Byzcoin face to dynamic adversaries. The exact level of coherency ensured by PeerCensus and other blockchain systems altogether with a formal framework to analyse the blockchains coherency has been proposed in [2]. This work continues the study conducted in [1]. In parallel and independent of the work in [2], [3] proposes a formalization of the distributed ledgers as an ordered list of records.

In the same spirit, Byzcoin [25] builds on top of *Practical Byzantine Fault-Tolerant* consensus [6] enhanced with a scalable collective signing process. [25] is based on a leader-based consensus over a group of members chosen based on a proof-of-membership mechanism. As in Bitcoin, when a miner succeeds to mine a block it is included in the voting members set that excludes one member. This protocol also inherits some of the Bitcoin problems and vulnerabilities. Also, the distributed implementation of the collective signing is still an open problem.

In order to avoid some of the previously cited problems, Micali [29] introduced (further extended in [5, 7]) the *sortition* based blockchains that completely replaces the proof-of-work mechanism by *sortition*. These works focus again on the agreement aspects of blockchains using probabilistic ingredients and providing probabilistic guarantees. More specifically, the set of nodes that are allowed to produce and validate blocks are randomly chosen and they change over the time. Interestingly, the study focuses only on synchronous round-based communication models.

The only academic work that addresses the consensus in *proof-of-stake* based blockchains is authored by Daian *et al.* [9] that proposes a protocol for weakly synchronous networks. The execution of the protocol is organized in epochs. Similar to Bitcoin-NG [15] 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 [24], proposes a sortition based proof-of-stake protocol and addresses mainly the security aspects of the proposed protocol.

Red Belly [8] 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 and the literature around this problem [4, 12, 11] 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 [20], Garay *et al.* define the notion of *chain-quality* as the proportion of blocks mined by honest miners in any given window; and the authors study the conditions where the ratio of blocks in the chain that were mined by malicious players over the total blocks in a given window is bounded. Kiayias *et al.* in [24] proposes Ourobouros[24], also analyses the chain-quality property. Pass *et al.* address in [32] one of the vulnerabilities of Bitcoin studied formally in Eyal and Sirer [16]. In [16] the authors prove that if the adversary controls a coalition of miners holding even a minority fraction of the computational power, this coalition can gain twice its share. Fruitchain [32] overcomes this problem by ensuring that no coalition controlling less than a majority of the computing power can gain more than a factor $1+3\delta$ by not respecting the protocol, where δ is a parameter of the protocol. In [14], 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 [21], Guerraoui and Wang study the effect of the delays of messages propagation in Bitcoin, and show that in a system of two miners, a miner can take advantage of the delays and be rewarded exponentially more that its expectation. In [22], Gürcan *et al.* study the fairness in Bitcoin from the point of view of the processes that do not participate to the mining. A similar work is done by Herlihy and Moir in [23] where the authors study the users fairness and consider as an example Tendermint. The authors discussed how processes with malicious behaviour can violate fairness by choosing transactions and then they propose modifcations to 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. Each process has a merit parameter that will model its stake in Proof-of-Stake based systems. We also consider a finite subset $V \in \Pi$ of size n of processes called validators. The set V may change during the execution. A process is promoted in V based on its merit parameter.

As local processing time are negligible with respect to message transfer delays, they are considered as being equal to zero.

Communication network. The processes communicate by exchanging messages through an eventually synchronous point-to-point network. *Eventually Synchronous* means that after a finite unknown a priori time τ there is a bound δ on the message transfer delay. *Point-to-point* means that in the network any pair of processes is connected by a bidirectional channel. Hence, when a process receives a message, it can identify its sender.

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 message 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 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.

Failure model. There is no bound on processes that can exhibit a Byzantine behaviour [33] in the whole 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 process 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 $f < n/3$.

Problem definition. In this paper we prove that Tendermint implements 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[8].** A decided value is valid, it satisfies the predefined predicate denoted isValid().

The concept of multi-consensus is presented in [4], 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 [12] 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 a lot over time. None of these specifications is appropriate for blockchain systems. In [11], 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 [11]. 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's protocol [28] aims at building a blockchain without fork. Tendermint has two architectural levels: Tendermint-core (that implements a variant of PBFT consensus) and Tendermint application, in charge of selecting the validators set that will execute the Tendermint-core agreement protocol. The Tendermint blockchain is an infinite chain of blocks which serve as an immutable distributed ledger. When building the blockchain, a subset of fixed size n of processes called validators should agree on the next block to append to the history. The set of validators is deterministically determined by the current history. We note that this subset may change once a block is appended. The mechanism to choose the validator for a given history is done by an upper layer application and is further referred as *selection mechanism*.

The genesis block of Tendermint blockchain is at *height* 0, and 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 for the previous block. Except the first block, each block refers to the previous block in the chain. We say that the *history* of length l of the blockchain is the finite chain composed of the blocks at the height smaller than l.

Giving a current height of the Tendermint blockchain, a total order set of validators is selected to add a new block. The validators start a *One-Shot Consensus*. The first validator proposes a block B that it creates, if at least $2n/3$ of the validators accept B, then 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 the 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* to build a blockchain, and at each height, it relies on a *One-Shot Consensus* to decide the block to be appended.

Although the choice of validators in the sets of validators is managed by the Tendermint application layer, 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. More details on the fairness issues in Tendermint and similar protocols are discussed in Section 6.

4.2 Tendermint One-Shot Consensus algorithm

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

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

---

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

Figure 1: Begining of the pseudocode of the One-shot Consensus Algorithm executed by a correct process p_i.

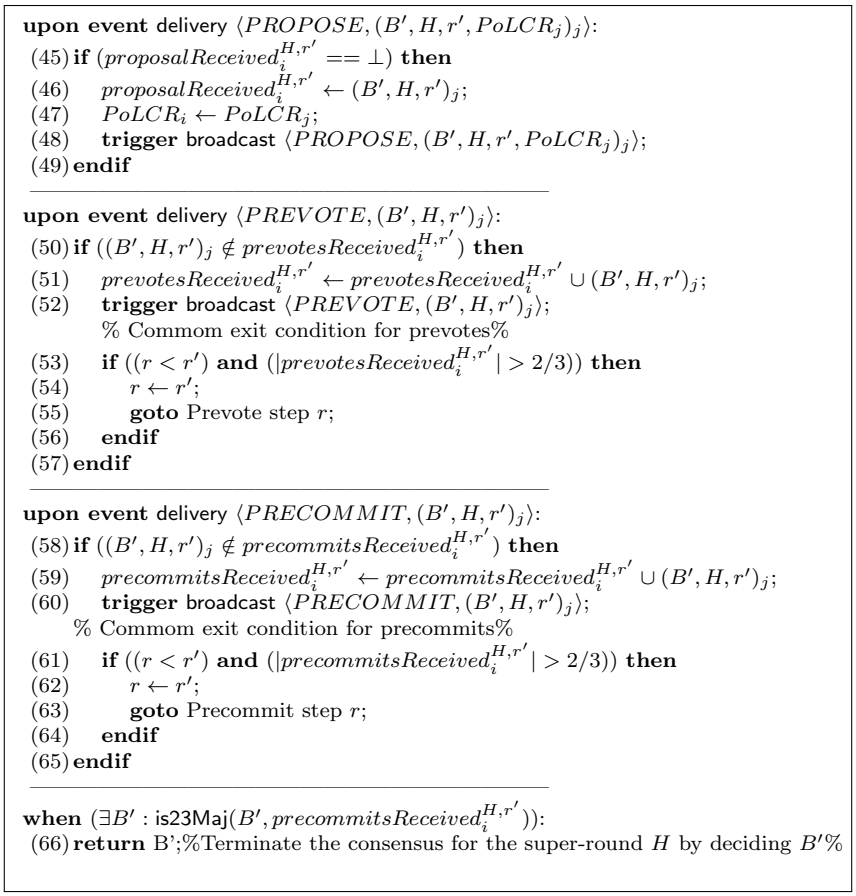

Figure 1: End of the pseudocode of the One-shot Consensus Algorithm executed by a correct process p_i.

The One-Shot Consensus process in deciding the next block for a given height H proceeds in *round*s. A *round* r, during which the processes try to decide on a proposed block, consists in three steps: (i) the *Propose step*, the proposer of the round broadcasts a proposal; (ii) the *Prevote step*, all processes broadcast their prevotes; and (iii) the *Precommit step*, all processes broadcast their precommits. When p_i broadcasts a message ($\langle TAG, m \rangle$), m contains a block B, and we say that p_i prevotes or precommint on m if $TAG{=}PREVOTE$ or $TAG{=}PRECOMMIT$ respectively. A process 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 $2n/3$ prevotes for B (resp. for *nil*). A *PoLC-Round* (PoLCR) is a round such that there was a PoLC for a block at round PoLCR.

Messages syntax, variables and data structures. $PoLCR_i$ is an integer. $lockedBlock_i$ is the last block on which p_i is locked. If it is equal to a block B, we say that p_i is locked on B, otherwise it is equal to *nil*, and we say that p_i is not locked. When $lockedBlock_i \neq nil$ and is taking the value *nil*, then p_i unlocks. Last-Locked-Round (LLR_i) is an integer representing the last round where p_i locked on a block. $TimeOutPropose$ is the maximum time a process has to stay in the Propose step. At the beginning of the height, the value of $TimeOutPropose$ is set to $\Delta_{Propose}$ and is incremented each time $TimeOutPropose$ expires. $TimeOutPrevote$ is the maxi-

mum time a process has to stay in the Prevote step. At the beginning of the height, the value of $TimeOutPrevote$ is set to $\Delta_{Prevote}$ and is incremented each time $TimeOutPrevote$ expires. B is the block the process created. $proposalReceived_i^{H,r}$ is 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.

- $\mathsf{proposer} : V \times Height \times Round \to \Pi$ is a deterministic function which gives the proposer for a given round at a given height in a round robin fashion.
- $\mathsf{createNewBlock} : 2^{\Pi} \times MemPool \to Block$ is an application-dependent function which creates a valid block (w.r.t. the application).
- $\mathsf{is23Maj} : (Block \cup nil) \times (prevotesReceived \cup precommitsReceived) \to Bool$ is a predicate that checks if there is at least $2n/3$ of prevotes or precommits on the given block or nil in the given set.
- $\mathsf{isValid} : Block \to Bool$ 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 $\mathsf{isValid}$ to false, (e.g. $\mathsf{isValid}(\perp) = \mathsf{false}$).

Detailed description of the algorithm. In Figure 1 we describe the algorithm to solve the One-Shot Consensus as defined in Section 3 for a given height H. Note that in order to prove that the protocol implements the One-Shot Consensus, we add the line 32 which is not present in the original implementation of Tendermint.

The algorithm proceeds in 3 phases for any given round r at height H:

1. Propose step (lines 6 - 16): If p_i is the proposer of the round, it creates a valid proposal and broadcasts it. Otherwise, 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 - 33): If p_i delivered 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 34 - 44): p_i checks if there was a PolC for a particular block or nil during the round (lines 34 and 38). There are three cases: (i) if there is a PoLC for a block

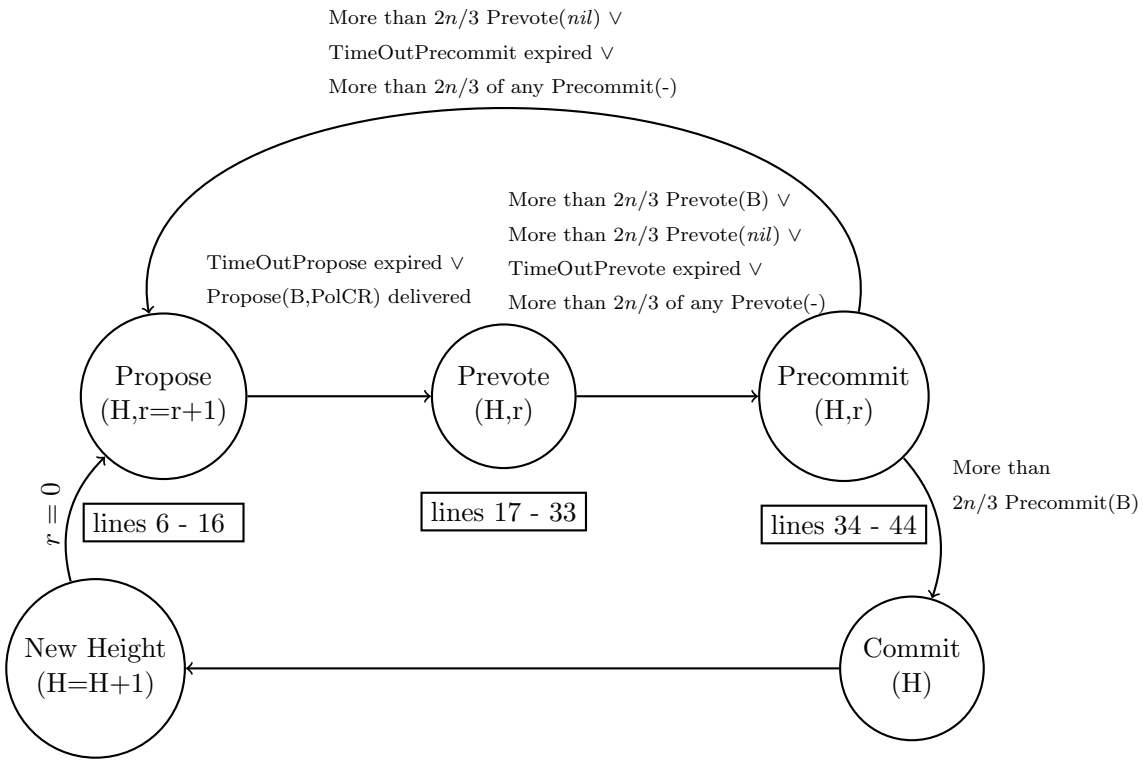

Figure 2: State Machine for the Tendermint Consensus described in Figure 1. For ease of readability, not all the common exit conditions are represented.

B, then it locks on B, and precommits on B (lines 34 - 36); (ii) if there is a PoLC for *nil*, then it unlocks and precommits on *nil* (lines 38 - 40); (iii) otherwise, it precommits on *nil* (line 42); in any case, p_i waits until $|precommitsReceived_i^{H,r}| > 2n/3$, and it goes to the next round.

Whenever p_i delivers a message, its broadcasts it (lines 48, 52 and 60). 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 $r' > r$). Those conditions are called the common exists conditions:

- For any round r', if for a block B, $\mathsf{is23Maj}(B, prevotesReceived_i^{H,r'}) = \mathsf{true}$, 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'.

Complexity. The One-Shot Consensus requires $f < n/3$ and weakly synchrony assumption. During the period of synchrony, if the proposer is correct and if $2n/3$ of processes can prevote on the proposal then the consensus terminates at the end of the round (see Lemma 5.4), thus in the

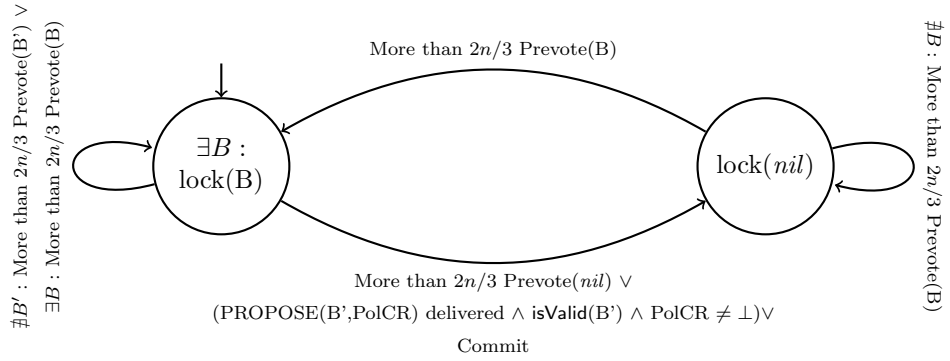

Figure 3: State machine Lock/Unlock

best case scenario the Terdermint One-Shot Consensus terminates in $O(1)$ round. Otherwise, in the worst case scenario, the algorithm terminates when a process already locked become a proposer, which is eventually true due to the round robin selection function. Thus, in the worst case the protocol terminates in $O(n)$ rounds, while the optimum is $O(f)$ [18]. Thus, contrarily to RedBelly, it is not optimal w.r.t. the number of rounds.

4.3 Tendermint Repeated Consensus algorithm

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

Data structures. The integer H is the height where is called the Consensus instance. V is 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 on instance of consensus. $TimeOutCommit$ is set to Δ_{Commit}.

Functions.

- validatorSet : $Height \rightarrow 2^{\Pi}$ 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 $\forall H \in Height, |validatorSet(H)| = n$.
- 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, we specified that it rewards the given subset of processes.
- atLeastOneThird : $Block \cup 2^{\Pi} \rightarrow Bool$ is a predicate which checks if there is at least $n/3$ of commits of the given block in the given set.

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

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

---

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

---

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

Figure 4: Pseudocode of the Repeated Consensus Algorithm executed by a correct process p_i.

- 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 Figure 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 expires. Then p_i decides B and goes to the next height.

Whenever p_i delivers a commit, it broadcasts it (lines 20 - 24). 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 10).

5 Tendermint Correctness

5.1 Correctness of Tendermint One-Shot Consensus

In this section we prove the correctness of Tendermint One-Shot Consensus algorithm in Figure 1 for a height H.

Lemma 5.1 (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 66). $\square_{Lemma\ 5.1}$

Lemma 5.2 (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 block 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 66), since the signature of the messages are unforgeable by hypothesis and $f < n/3$, then more than $n/3$ of those precommits for round r are from correct processes. It means that each of those correct processes had a PoLC for B during the round r (lines 34 - 36), 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 block B during a round r in two cases. Either Case a, during r, p_j is not locked on any block and checks the validity of B, or Case b: p_j is already locked on B and does not checks its validity (lines 21 - 26).

- Case a: If p_j is not locked on any block then before prevoting it checks the validity of B and prevotes on B if B is valid;
- 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 34 - 36), by the same argument, there is a round which happens previously than r' where p_j was locked or B is valid.

  Since a process locks during a round smaller than the current one, and that there exists a first round where all correct processes are not locked (line 3), there is a round $r'' < r'$ where p_j was not locked on B but prevoted B, so B is valid (lines 22 and 23).

Therefore a decided value by a correct process is valid. $\square_{Lemma\ 5.2}$

Lemma 5.3 (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 decides B. If p_i decides B, then $\mathsf{is23Maj}(B, precommitsReceived_i^{H,r}) = \mathsf{true}$ (line 66), 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 (line 35). p_i broadcasts all the precommits it delivers (line 60), 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 block than B. Since at least $n/3$ of the correct processes are locked on B, they only prevote on B (line 21), there cannot be a PoLC for a block different than B, even if all the faulty processes voted for it, so the correct processes that are locked cannot unlock after the round r (line 19). The number of precommit for a different block than B is then bounded by f, which is not enough to decide. All the correct processes will eventually deliver the $2n/3$ signed precomits p_i delivered and broadcasted, and then will decide B. $\square_{Lemma\ 5.3}$

Lemma 5.4. In an eventual synchronous system, Tendermint One-Shot Consensus Algorithm verifies the following property: If the system becomes synchronous, the proposal is delivered during the Propose step, the prevotes for correct processes are delivered during the prevotes step and there is a correct proposal, then eventually a correct process decides.

Proof Let r be the round where the communication becomes synchronous and all the messages are delivered by the correct processes within their respective step. If a correct process decides before r, then by lemma 5.3 all the correct process eventually decided. Otherwise, no correct process decided yet, and we have three cases:

- Case 1: No correct process is locked on a block before r.

  That means that there is no PoLCR for the round and the correct processes delivered the proposal with a block B (lines 12 and 46 and 48) before the Prevote step. Since the proposal is valid, then they all broadcast their prevote for that block (line 23), and deliver the others' prevotes and broadcast them before entering the Precommit step (lines 27 - 33 and 52). Then $\mathsf{is23Maj}(B, precommitsReceived_i^{H,r}) = \mathsf{true}$ for them. The correct processes will lock on B, precommit on B, and broadcast their precommit (lines 34 - 36). Eventually a process p_i will have $\mathsf{is23Maj}(B, precommitsReceived_i^{H,r}) = \mathsf{true}$ then p_i will decide (line 66).

- Case 2: Some correct process is locked on a block, and there is a valid PoLCR in the proposal.

  Since the PoLCR is valid, then the processes that are locked will unlock (line 19) and the proof follows as in the previous case.

- Case 3: Some correct processes are locked on a block, and there is no PoLCR in the proposal.

  - If the number of processes that are locked are less than $n/3 - f$ (which means that even without them, there are more than $2n/3$ other correct processes unlocked), then as in the case 1, a correct process will decide.
  - If the number of processes that are locked is greater or equal than $n/3 - f$, then during that round, $\nexists B' : \mathsf{is23Maj}(B', precommitsReceived_i^{H,r}) = \mathsf{true}$, but since the proposer is selected in a round robin fashion, then eventually, a process that is locked on a block will be the proposer and propose a valid PoLCR, and the proof follows the case 2.

$\square_{Lemma\ 5.4}$

Lemma 5.5 (Termination)**.** In an eventual synchronous system, 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 28 - 33), so eventually, during the synchrony period of the system all the correct processes will deliver the proposal and the prevotes respectively during the Propose and the Prevote step. In the synchronous period, the function proposer() (line 6) operates in a round robin fashion and eventually (in at most after $f + 1$ rounds) there is a round where the proposer is a correct process and that a correct process proposes a valid proposal. By lemma 5.4, a correct process decides a value, and by the lemma 5.3, every correct process eventually decides the same value. $\square_{Lemma\ 5.5}$

Theorem 5.6. In an eventual synchronous system, Tendermint One-Shot Algorithm implements the One-Shot Consensus.

Proof The proof follows directly from Lemmas 5.1, 5.2, 5.3 and 5.5. $\square_{Theorem\ 5.6}$

5.2 Correctness of Tendermint Repeated Consensus

In this section we prove the correctness of Tendermint Repeated Consensus algorithm in Figure 4.

Lemma 5.7 (Termination)**.** In an eventual synchronous system, 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 advance to a new height, or from a certain height H it outputs only $\perp$.

- If p_i cannot advance, one of the following case is satisfied:
  - The function consensus() does not terminate (line 10), which is a contradiction due to Lemma 5.5; or
  - p_i waits an infinite time for receiving enough commits (line 13), which cannot be the case because of the best effort broadcast guarantees. Since all the correct validators terminate the One-Shot Consensus and broadcast their commit.
- If p_i decides at each height (line 18), 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 9 and 10), or (ii) p_i is not a validator for H but received at least $n/3$ commits for $\perp$ (lines 13 and 23).
  - (i): Since consensus() returns the value $\perp$, that means by Lemma 5.2 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 9 - 11), and because $f < n/3$, it means that p_i delivered a commit from at least one correct validator (process). By Lemma 5.2, 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. $\square_{Lemma\ 5.7}$

Lemma 5.8 (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 9 and 10). By lemma 5.3 p_j and p_k decide the same value and then output that same value (line 18).
- 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 5.3, and broadcast their commit (line 11), 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 13). 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.

$\square_{Lemma\ 5.8}$

Lemma 5.9 (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 18), then that value has been set during the execution and for that height (line 7).

- If p_i is a validator for the height H, then B is the value returned by the function consensus(), by the lemma 5.2 we have that isValid(B) = true.
- If p_j is not a validator for the height H, it means that it received at least $n/3$ signed commits from the validators for the block B (lines 9 - 11 and 20 - 24), hence at least one correct validator commited B, and by lemma 5.2 we have have that isValid(B) = true.

So each block that a correct process outputs satisfies the predicate isValid(). $\square_{Lemma\ 5.9}$

Theorem 5.10. In an eventual synchronous system, Tendermint Repeated Consensus Algorithm implements the repeated consensus.

Proof The proof follows directly from Lemmas 5.7, 5.8 and 5.9. $\square_{Theorem\ 5.10}$

6 Tendermint Fairness

In [32] the authors define the fairness of a Proof-of-Work based blockchain protocol for a system of n processes as follows: *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. In Tendermint, for each block, there is a subset of processes called the validators that produced that block. The correct validators for a block are the processes that have to be rewarded for that block.

We therefore extend the definition of [32] for a system with an infinite number of processes, and where each block is produced by a subset of processes.

Informally, we say that a blockchain protocol is fair if any correct process that wield ϕ of the total merit in the system will get at least ϕ of the total reward that is given in the system.

In order to study the fairness of a protocol in a consensus based blockchain, we split the protocol in two mechanisms: (i) the *selection mechanism* which selects for each new height the validators for that height taking into account the merit of each process, and (ii) the *reward mechanism*, which is the mechanism giving the rewards for the correct processes that agreed on a new block in the blockchain.

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.

For a process p_i we denote α_i its merit proportional to the total merit, and by v_i the number of time p_i becomes validator proportionally to the number of blocks.

Definition 6.1 (Fairness of a Selection Mechanism)**.** We say that a selection mechanism is fair if it respects the following properties:

- If $\alpha \neq 0$ then $v_i \neq 0$, informally, this means that each process with a positive merit parameter should become a validator infinitely often;
- If a process p_i has a merit parameter α_i and a process p_j has a merit parameter α_j, then if $\alpha_i \geq \alpha_j$ then $v_i \geq v_j$.

For each block that is in the blockchain, the validators who contributed to its creation should be rewarded. We define the following properties for the fairness of a reward mechanism.

1. For any block in the blockchain, all the correct validators for that block have a reward parameter for that block greater than 0,
2. For any 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 greater than 0.

3'. There exists a height H such that for all blocks 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 greater than 0.

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

If a reward mechanism is fair, it means that all and only the correct validators for each block in the blockchain get the reward for it.

Definition 6.3 (Eventual fairness of a reward mechanism)**.** A reward mechanism is *eventually fair* if it satisfies the conditions 1, 2 and 3'.

If a reward mechanism is eventually fair, it means that some correct validators may not get some rewards at the begin of the execution, but there is a height from which all and only the correct validators for the next blocks are rewarded.

We note that if a reward mechanism is fair, then it also is eventually fair but the reverse is not necessarily true.

Definition 6.4 ((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.

Remark 1. If $|V| > 1$ then for a reward mechanism to be (eventually) fair, the reward cannot be given directly in the block. In fact the set of correct validators cannot be known in advance. If $|V| = 1$ the reward can be directly given to the only validator, so in the block.

6.1 Tendermint's reward mechanism

We note that in Tendermint, the selection of the validators is let at the discretion of the Tendermint application layer. We study the reward mechanism of Tendermint, and assume the fairness of its selection mechanism.

The reward mechanism in Tendermint is the following:

- Once a new block is decided for height H, processes wait for at most TimeOutCommit time to collect the decision from the other validators for H, and put them in their set *toReward* (Figure 4, lines 17 and 20 - 24).
- 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 set *toReward* (Figure 4, line 10).

Only the processes from which p_i delivered a commit will get a reward for the block at height $H-1$.

Lemma 6.1. The reward mechanism of Tendermint is not eventually fair.

Proof We assume that the system becomes synchronous, and that $TimeOutPropose < \Delta$, 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. The Tendermint's reward mechanism is not eventually fair. $\square_{Lemma\ 6.1}$

Lemma 6.2. The variant of Tendermint where the TimeOutCommit increases for each round until it catches up the message delay, has an eventual fair reward mechanism.

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 commited the block from $H-1$, and they give to those validators from $H-1$ a reward parameter greater than 0. The validators in H give the reward to the 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. $\square_{Lemma\ 6.2}$

Theorem 6.3. In an eventual synchronous system, if the selection mechanism of Tendermint is fair, then Tendermint with modulable time-out is eventually fair.

Proof The proof follows by lemma 6.2. $\square_{Theorem\ 6.3}$

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 (RC) blockchain protocols. We say that a blockchain protocol is a RC if it builds the blockchain by a mechanism of repeated consensus, where at each height, a subset of processes called validators produces a block executing an instance of One-Shot Consensus.

Theorem 6.4. There exists a fair reward mechanism in a RC 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 that the reward mechanism presented is fair.

The system is synchronous and messages sent are delivered within at most x blocks. The exact set of correct validators from $H - x$ is known by all at H. 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 satisfy 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, we know 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 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.

$\square_{Theorem\ 6.4}$

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.

Corollary 6.4.1. There exists an eventual fair reward mechanism in a RC 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 by theorem 6.4. Otherwise, we prove 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 validator from 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 all 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 from 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 all the messages 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 care about the beginning. So by applying the theorem 6.4 from the time when the system becomes synchronous, we end the proof.

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

  If the reward mechanism is fair, by theorem 6.4, 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 6.4, 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.

The proof follows directly by the theorem 6.4. $\square_{Corollary\ 6.4.1}$

Remark 2. In an asynchronous system, there is no (eventual) reward mechanism, so if the communication system is asynchronous, then there is no (eventual) fair RC Blockchain protocol. Note that this result is valid even if all the processes are correct.

7 Conclusion

This paper has formalized Tendermint a PBFT-based repeated consensus protocol where the set of validators is dynamic. One-shot instance of Tendermint takes $O(n)$ rounds where n is the number of validators in that round. We also studied the fairness aspects of the reward mechanism in Tendermint and proved it becomes eventually fair with a small twist in the timeouts tuning. We also prove that the fairness of the reward mechanism of repeated-consensus blokchains is (eventually) fair iff the system communication is (eventually) synchronous. As future work we intend to study the fairness of the selection mechanism.

References

[1] Emmanuelle Anceaume, Romaric Ludinard, Maria Potop-Butucaru, and Frédéric Tronel. Bitcoin a distributed shared register. In *Stabilization, Safety, and Security of Distributed Systems - 19th International Symposium, SSS 2017, Boston, MA, USA, November 5-8, 2017, Proceedings*, pages 456–468, 2017.

[2] Emmanuelle Anceaume, Antonella Del Pozzo, Romaric Ludinard, Maria Potop-Butucaru, and Sara Tucci Piergiovanni. Blockchain abstract data type. *CoRR*, abs/1802.09877, 2018.

[3] Antonio Fernández Anta, Chryssis Georgiou, Kishori M. Konwar, and Nicolas C. Nicolaou. Formalizing and implementing distributed ledger objects. *CoRR*, abs/1802.07817, 2018.

[4] Amotz Bar-Noy, Xiaotie Deng, Juan A. Garay, and Tiko Kameda. Optimal amortized distributed consensus. *Inf. Comput.*, 120(1):93–100, 1995.

[5] Iddo Bentov, Rafael Pass, and Elaine Shi. The sleepy model of consensus. *IACR Cryptology ePrint Archive*, 2016:918, 2016.

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

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

[8] 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`, 2017.

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

[10] 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.

[11] 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.

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

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2), April 1985.

[18] Michael J Fischer and Nancy A Lynch. A lower bound for the time to assure interactive consistency. *Information processing letters*, 14(4):183–186, 1982.

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

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

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

[22] 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.

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

[24] 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.

[25] 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.

[26] Artem Koltsov, Vitaly Cheremensky, and Stanislav Kapulkin. Casper White Paper. `https://casperproject.io/docs/Casper_whitepaper_eng.pdf`.

[27] Jae Kwon and Ethan Buchman. Cosmos: A Network of Distributed Ledgers. `https://cosmos.network/resources/whitepaper`.

[28] Jae Kwon and Ethan Buchman. Tendermint. https://tendermint.readthedocs.io/en/master/specification.html

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

[30] S. Nakamoto. Bitcoin: A Peer-to-Peer Electronic Cash System. `https://bitcoin.org/bitcoin.pdf`, 2008.

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

[32] Rafael Pass and Elaine Shi. 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.

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

[34] G. Wood. Ethereum: A secure decentralised generalised transaction ledger. `http://gavwood.com/Paper.pdf`.