Dissecting Tendermint

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

Dissecting Tendermint

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

In this paper we analyze Tendermint proposed in [7], one of the most popular blockchains based on PBFT Consensus. The current paper dissects Tendermint under various system communication models and Byzantine adversaries. Our methodology consists in identifying the algorithmic principles of Tendermint necessary for a specific combination of communication model - adversary. This methodology allowed to identify bugs [3] in preliminary versions of the protocol ([19], [7]) and to prove its correctness under the most adversarial conditions: an eventually synchronous communication model and asymmetric Byzantine faults.

1 Introduction

In the Blockchain systems area the recent tendency is to privilege solutions based on distributed agreement than proof-of-work. This is motivated by the fact that the majority of proof-of-work based solutions such as Bitcoin or Ethereum are energetically not viable when the economical efficiency is targeted. Moreover, proof-of-work solutions guarantee only with high probability the existence of an unique chain, the major drawback for using blockchains in industrial applications. That is, forks even though they are rare do still happen with a dramatic impact on the consistency guarantees offered by the system. Therefore, alternatives to proof-of-work have been recently considered. Interestingly, the research in blockchain systems revived a branch of distributed systems research: Byzantine fault-tolerant protocols having PBFT consensus protocol as ambassador. In the class of blockchains based on distributed agreement, Tendermint (inspired by PBFT consensus) is one of the most popular.

The Blockchain is a distributed ledger implementing an append-only list of blocks chained to each other, it serves as an immutable and non repudiable ledger in a system composed of untrusted components. These characteristics are a fruitful field to envision new industrial applications. Those applications, by their nature, demand strong consistency quality of services from the underlying data structure, the blockchain. Consensus algorithms provide strong consistency for free. Since Blockchain is a list, the append operation needs to preserve the chain shape of the data structure, leading to the necessity to have a mechanism allowing untrusted processes to agree on the next block to append. Bitcoin Blockchain, the most famous Blockchain, employs the proof-of-work mechanism [13]. That is, processes willing to append a new block have to solve a crypto-puzzle. The winning process will proceed appending the new block. While this mechanism does not require a real coordination between the processes participating to the Bitcoin system, it might lead to

inconsistencies. Indeed, if more than one process solves the crypto-puzzle to extend the same block, then the chain shape is lost as long as the conflict is unsolved. In [4], the authors proved that the only way to avoid forks is to employ a consensus mechanism.

Consensus [20] is a fundamental problem in the distributed system area, which allows coordination among processes. It can be informally described as follows: given a set of processes proposing a value (which differs from process to process) then, after a finite amount of time, all processes agree on the same value, chosen among the proposed ones.

The Byzantine Agreement problem, as proved in [20], cannot be solved with less than $3f + 1$ processes in a synchronous message-passing system (where the message delivery delay is upper bounded) in the presence of f Byzantine processes and in an asynchronous message-passing system (when there are no upper bounds on the message delivery delay) in the presence of one faulty (crash) process, as proved in the seminal FLP paper [15]. In between those impossibility results, few years later, Dwork et al. [12] proved that considering an eventual synchronous message-passing system (there is a time after which there is an upper bound on the message delivery delay) it is possible to solve Consensus even in presence of Byzantine faults. Finally, Castro and Liskov proposed the PBFT [8], which optimizes the performances of the previous solution. The eventual synchronous message-passing system in presence of Byzantine faults is the model considered nowadays in implementing blockchain. There exist different BFT based blockchain propositions (e.g., [1][17][11] based on PBFT) and real implementations as Hyperledger [5] based on BFT-SMaRt [22], Tendermint [24] based on a variation of PBFT and RedBelly based on DBFT Consensus algorithm [10].

In this paper we analyze Tendermint proposed in [7] as one of the most promising but not fully analyzed protocols so far. Tendermint targets an eventual synchronous system [12], which means that safety has to be guaranteed in the asynchronous periods and liveness in synchronous ones, when a subset of processes can be affected by Byzantine failures. To analyze the protocol, we dissect Tendermint identifying the techniques used to address different challenges due to the system model or the power of the adversary. We consider the following system models, from the strongest to the weakest: synchronous round-based model in presence of symmetric Byzantine faults (i.e., an adversary such that its behaviour is perceived identically by all non-faulty processes); synchronous round-based model in presence of asymmetric Byzantine faults (i.e., its behavior may be perceived differently by different non-faulty processes); and finally eventual synchronous communication model in presence of asymmetric Byzantine faults. For each type of model we provide the corresponding algorithm (a variant of Tendermint [7]). Finally, we provide a proved correct protocol specification of [7] in the eventual synchronous setting in presence of asymmetric Byzantine faults and computed its complexity. This methodology allowed to identify bugs [3] in the preliminary versions of the protocol ([19], [7]) that now have been solved.

2 Model

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* [2], 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 [18] that is in charge of implementing the selection of V.

Communication network. The processes communicate by exchanging messages through an eventually synchronous network [12]. *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 [21] 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. 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 $f < n/3$.

In [16], different categories of faults have been considered, in particular we consider *symmetric* and *asymmetric*: a process p_i is said to be *symmetrically faulty* if its behaviour is perceived identically by all non-faulty processes; and a process p_i is said to be *asymmetrically faulty* if its behaviour may be perceived differently by different non-faulty processes. In this work, we called the processes symmetrically faulty, the symmetric Byzantine, and we called the asymmetrically faulty the asymmetric Byzantine or just Byzantine.

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

Let us note that the assumed broadcast primitive in an open dynamic network can be implemented through *gossiping*, i.e. each process sends the message to current neighbors in the underlying dynamic network graph. In these settings the finite arrival model is a necessary condition for the system to show eventual synchrony. Intuitively, a finite arrival implies that message losses due to topology changes are bounded, so that the propagation delay of a message between two processes not directly connected can be bounded [6].

Round-based Execution model. We assume that each correct process evolves in rounds. A *round* consists of three phases, in order : (i) a *Send* phase, where the process broadcasts messages computed during the last round, or a default messages for the first round; (ii) a *Delivery* phase where the process collect messages sent during the current and previous rounds; and (iii) a *Compute* phase where the process uses the messages delivered to change its state. At the end of a round a process exit from the current round and starts the next round. Each round last a certain duration, we consider the Send and the Compute phase as being atomic, they are executed instantaneously, but not the Delivery phase. In a synchronous network, we assume the the duration of the Delivery phase, and so of the round is δ. In an eventually synchronous network, the duration of a round is monotonically increasing, it is modified each time the process does not deliver "enough" messages,

for instance a majority of correct responses.

Problem definition. In this paper we analyze the correctness of the Tendermint protocol against the Consensus abstraction in distributed systems. We say that an algorithm implements 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 v, then eventually all the correct processes decide v; **Validity[9]**, a decided value is valid, it satisfies the predefined predicate denoted valid().

3 Tendermint Algorithms

Tendermint BFT Consensus protocol [19, 24, 7] is a variant of PBFT consensus that aims at being the core layer under the Tendermint blockchain.

In this work, we dissect the Tendermint protocol. We present it in three different models, from the strongest to the weakest: (i) synchronous communication in presence of Symmetric Byzantine processes; (ii) synchronous communication in presence of Asymmetric Byzantine processes and; (iii) Eventually Synchronous communication in presence of Asymmetric Byzantine processes. We show the modifications the algorithm needs to work from (i) to the model (ii) and finally to the model (iii).

The three Tendermint protocols share a common algorithmic structure, they proceed in *epoch*s, and each epoch consists in three rounds: (i) the *PRE-PROPOSE* round where the proposer of the epoch broadcasts a value, (ii) the *PROPOSE* round, where each process accept and broadcast the pre-proposal or *nil* otherwise, and the (iii) *VOTE* round, where processes broadcast the value they are voting on according the the proposal they receive.

When p_i broadcasts a message $\langle TAG, h, e, m\rangle$, m contains a value, we say that p_i pre-proposes, proposes or votes m if TAG=`PRE-PROPOSE`, TAG=`PROPOSE` or TAG=`VOTE` respectively. When p_i broadcasts $\langle$`PRE-PROPOSE`$, h, e, m, e'\rangle$, where e' is an epoch we also say that p_i pre-proposes m with an epoch e'.

For each epoch, there is a process that is called the proposer for the epoch which should pre-propose.

Messages syntax, variables and data structures. h_p is an integer representing the consensus instance the process is currently executing. e_p is an integer representing the epoch where the process p is, we note that for each height, a process may have multiple epochs. $decision_p$ is a table that contains the sequence of decisions, $decision_p[h]$ is the decision of process p for the consensus instance h. $proposal_p$ is the value the process p proposes. $vote_p$ is the value the process p votes.

Functions. We denote as $Block$ the set containing all blocks, and as $MemPool$ the data structure containing all the transactions.

- proposer : $Height \times Epoch \rightarrow \Pi$ is a deterministic function which gives the proposer for a given epoch at a given height in a round robin fashion.
- valid : $Value \rightarrow Bool$ is an application dependent predicate that is satisfied if the given value is valid. If there is a value v such that $\mathsf{valid}(v) = \mathsf{true}$, we say that v is valid. Note that we set $\mathsf{valid}(nil) = \mathsf{false}$.
- getValue : $Void \rightarrow Value$ is an application dependent predicate which gives a valid value.
- id : $Value \rightarrow Hash$ application that gives a unique identifier to a value. In the current version of Tendermint, the hash of the value represents its identifier, such that instead of sending a whole value an identifier is sent to optimize the communication cost.

- sendByProposer : $Height \times Epoch \times Value \rightarrow Bool$ is an predicate that gives true if the given value has been pre-proposed by the proposer of the given height during the given epoch.
The data structures above are shared by all three algorithms. In each section we specify the data structures relative to a specific version of the algorithm.

Algorithm 1 Simplified Tendermint in a Synchronous and Symmetric faults model

```
1: Initialization:
2:    h_p := 0                                   /* current height, or consensus instance currently executed */
3:    e_p := 0                                   /* current epoch number */
4:    decision_p[] := nil
5:    proposal_p := getValue()
6:    vote_p := nil

7: Round PRE-PROPOSE(e_p):
8:    Send phase:
9:       if proposer(h_p, e_p) = p then
10:         broadcast ⟨PRE − PROPOSE, h_p, e_p, proposal_p⟩ to all processes
11:   Delivery phase:
12:      delivery ⟨PRE − PROPOSE, h_p, e_p, v⟩ from proposer(h_p, e_p)
13:   Compute phase:
14:      if valid(v) then
15:         proposal_p ← id(v)
16:      else
17:         proposal_p ← nil

18: Round PROPOSE(e_p):
19:   Send phase:
20:      broadcast ⟨PROPOSE, h_p, e_p, proposal_p⟩ to all processes
21:   Delivery phase:
22:      delivery ⟨PROPOSE, h_p, e_p, ∗⟩ from all processes
23:   Compute phase:
24:      if valid(select(⟨PROPOSE, h_p, e_p, ∗⟩)) then
25:         vote_p ← id(select(⟨PROPOSE, h_p, e_p, ∗⟩))
26:      else
27:         vote_p ← nil

28: Round VOTE(e_p):
29:   Send phase:
30:      broadcast ⟨VOTE, h_p, e_p, vote_p⟩ to all processes
31:   Delivery phase:
32:      delivery ⟨VOTE, h_p, e_p, ∗⟩ from all processes
33:   Compute phase:
34:      if valid(select(⟨VOTE, h_p, e_p, ∗⟩)) ∧ decision_p[h_p] = nil then
35:         decision_p[h_p] = select(⟨VOTE, h_p, e_p, ∗⟩)
36:         h_p ← h_p + 1
37:      else
38:         e_p ← e_p + 1
39:         proposal_p ← getValue()
```

Symmetric Byzantine Synchronous System The function select : $Message^* \rightarrow Value$ chooses deterministically a value with respect to the set of messages given in parameter.

Detailed description of the algorithm. In Algorithm 1 we describe the algorithm to solve the Consensus as defined in Section 2 in a synchronous system, and under the assumption that Byzantine processes can only exhibit a symmetric behaviour. The algorithm proceeds in 3 rounds for any given epoch e at height h:

1. Round PRE-PROPOSE (lines 7 - 17): If the process p is the proposer of the epoch, it pre-proposes its proposal value. Otherwise, it waits for the proposal from the proposer.

If a process p delivers the pre-proposal from the proposer of the epoch, p sets its proposal to the pre-proposal, otherwise it sets it to *nil*.

2. Round PROPOSE (lines 18 - 27): During the PROPOSE round, each process broadcasts its proposal, and collects the proposals sent by the other processes. After the Delivery phase of the round propose, each process has a set of proposals, and process p sets the variable $vote_p$ to the value returned by the deterministic function select on the set of proposals delivered.

3. Round VOTE (lines 28 - 39): In the round VOTE, correct process q votes $vote_q$. q collects all the votes that were broadcast, then it checks if the value returned by the function select let us say v is valid. If v is valid, then q decides v, otherwise it increases the epoch number and goes to the PRE-PROPOSE round.

Let us stress that such algorithm is only pedagogical. Indeed, in such system model at the end of each round all the correct processes share the same set of messages delivered and solving consensus became trivial. The first round PRE-PROPOSE just forces the processes on the value they PROPOSE after. Then, there are the classical two rounds to exchange the values among processes before taking a decision. Notice that in this case, even in presence of a faulty proposer, if there is at least one valid value in the final set at the end of the epoch, then correct processes decide.

Byzantine Synchronous System This section presents the algorithm that solves Consensus in a synchronous model in presence of asymmetric behaviour from Byzantine processes. Byzantine processes can send different messages to different processes, in other words, at the end of an epoch different correct processes can have different messages delivered, which can harm the safety condition, i.e., some correct processes might decide and some do not. Then, information from one epoch have to be kept for the next ones. That is why, along with a new function, we need for two more variables.

Functions. $2f+1 : PROPOSE^* \cup VOTE^* \rightarrow Bool$: checks if there is at least $2f+1$ proposals/vote in the given set.

Messages syntax, variables and data structures. (i) $lockedValue$ stores a value which is potentially will be decided. If process p delivered more than $2f + 1$ proposes for the same value v during its PROPOSE round, it sets its $lockedValue_p$ to v, (ii) $validValue$ stores a value which is potentially will be decided. If the process p delivered at least $2f + 1$ proposes for the same value v whether during its PROPOSE round or its VOTE round, it sets its $validValue$ to v. $validValid$ is the last value that a process delivered at least $2f + 1$ times, and can be different than $lockedValue$.

Those variables are used to ensure the Agreement condition. Byzantine processes can send different information to different processes then some correct processes (but not all) may deliver $2f + 1$ occurrences for the same value. Therefore such value can be a potential decided value for some correct process p. To prevent the violation of the agreement property, there is a need to keep track locally of such value. Such that if p is the next proposer pre-proposes such value. Otherwise p checks the new pre-proposal against such value.

Detailed description of the algorithm. In Algorithm 2 we describe the algorithm to solve the Consensus as defined in Section 2 in a synchronous system, and where Byzantine processes can exhibit even asymmetric behaviour. The algorithm proceeds in 3 rounds for any given epoch e at height h:

Algorithm 2 Simplified Tendermint in a Synchronous model

```
1: Initialization:
2:    h_p := 0                                   /* current height, or consensus instance currently executed */
3:    e_p := 0                                   /* current epoch number */
4:    decision_p[] := nil
5:    lockedValue_p := nil
6:    validValue_p := nil
7:    proposal_p := getValue()
8:    vote_p := nil

9: Round PRE-PROPOSE(e_p):
10:   Send phase:
11:     if proposer(h_p, e_p) = p then
12:       broadcast ⟨PRE − PROPOSE, h_p, e_p, proposal_p⟩ to all processes
13:   Delivery phase:
14:     delivery ⟨PRE − PROPOSE, h_p, e_p, v⟩ from proposer(h_p, e_p)
15:   Compute phase:
16:     if valid(v) ∧ validValue_p = nil then
17:       proposal_p ← id(v)
18:     else
19:       if !valid(v) ∨ v ∉ {lockedValue_p, validValue_p} then
20:         proposal_p ← nil
21:       else
22:         proposal_p ← id(v)

23: Round PROPOSE(e_p):
24:   Send phase:
25:     broadcast ⟨PROPOSE, h_p, e_p, proposal_p⟩ to all processes
26:   Delivery phase:
27:     delivery ⟨PROPOSE, h_p, e_p, ∗⟩ from all processes
28:   Compute phase:
29:     if 2f + 1⟨PROPOSE, h_p, e_p, id(v')⟩ ∧ sendByProposer(h_p, e_p, v') ∧ valid(v') then
30:       lockedValue_p ← v'
31:       validValue_p ← v'
32:       vote_p ← id(v')
33:     else
34:       vote_p ← nil

35: Round VOTE(e_p):
36:   Send phase:
37:     broadcast ⟨VOTE, h_p, e_p, vote_p⟩ and ⟨PROPOSE, h_p, e_p, ∗⟩ delivered to all processes
38:   Delivery phase:
39:     delivery ⟨VOTE, h_p, e_p, ∗⟩ and ⟨PROPOSE, h_p, e_p, ∗⟩ delivered from all processes
40:   Compute phase:
41:     if 2f + 1⟨PROPOSE, h_p, e_p, id(v'')⟩ ∧ sendByProposer(h_p, e_p, v'') ∧ valid(v'') then
42:       validValue_p ← v''
43:     if 2f + 1 ⟨VOTE, h_p, r, id(v'')⟩ ∧ sendByProposer(h_p, e_p, v'') ∧ valid(v'') ∧ decision_p[h_p] = nil then
44:       decision_p[h_p] = v''
45:       h_p ← h_p + 1
46:       reset lockedValue_p, validValue to init values and empty message log
47:     else
48:       e_p ← e_p + 1
49:       if validValue_p ≠ nil then
50:         proposal_p ← validValue_p
51:       else
52:         proposal_p ← getValue()
```

1. Round PRE-PROPOSE (lines 9 - 22): If the process p is the proposer of the epoch, it pre-proposes its proposal value, otherwise, it waits for the proposal from the proposer. The proposal value of the proposer is its $validValue$ if $validValue \neq nil$.

   If a process q delivers the pre-proposal from the proposer of the epoch, q checks the validity of

the pre-proposal with respect to its state, and if the pre-proposal is valid, q sets its proposal $proposal_q$ to the pre-proposal, otherwise it sets it to *nil*.

2. Round PROPOSE (lines 23 - 34): During the PROPOSE round, each process broadcasts its proposal, and collects the proposals sent by the other processes. After the Delivery phase of the round propose, process p has a set of proposals, and checks if a value v, pre-proposed by the proposer, was proposed by at least $2f + 1$ different processes, if it is the case, and the value is valid, then p sets $vote_p, validValue$ and $lockedValue$ to v, otherwise it sets $vote_p$ to nil.

3. Round VOTE (lines 35 - 52): In the round VOTE, a correct process p votes $vote_p$ and broadcasts all the proposals it delivered during the current epoch. Then p collects all the messages that were broadcast. First p checks if its now has delivered at least $2f+1$ of proposal for a value v' pre-proposed by the proposer of the epoch, in that case, it sets $validValue_p$ to that value then it checks if a value v' pre-proposed by the proposer of the current epoch is valid and has at least $2f + 1$ votes, if it is the case, then p decides v' and goes to the next height; otherwise it increases the epoch number and update the value of $proposal_p$ with respect to $validValue_p$.

Algorithm 3 Tendermint Consensus Algorithm

```
1: Initialization:
2:   h_p := 0                                   /* current height, or consensus instance currently executed */
3:   e_p := 0                                   /* current epoch number */
4:   decision_p[] := nil
5:   lockedValue_p := nil , validValue_p := nil
6:   lockedEpoch_p := −1 , validEpoch_p := −1
7:   proposal_p := getValue()
8:   vote_p := nil

9: Round PRE-PROPOSE(e_p):
10:  Send phase:
11:    if proposer(h_p, e_p) = p then
12:      broadcast ⟨PRE − PROPOSE, h_p, e_p, proposal_p, validEpoch_p⟩ to all processes
13:  Delivery phase:
14:    delivery ⟨PRE − PROPOSE, h_p, e_p, v, e⟩ from proposer(h_p, e_p)
15:  Compute phase:
16:    if f + 1⟨∗, h_p, epoch, ∗, ∗⟩ ∧ epoch > e_p then
17:      e_p ← epoch
18:      goto PRE-PROPOSE(e_p)
19:    if 2f + 1 ⟨PROPOSE, h_p, e, id(v)⟩ ∧ e ≥ lockedEpoch_p ∧ e < e_p ∧ valid(v) then
20:      proposal_p ← id(v)
21:    else
22:      if !valid(v) ∨ (lockedEpoch_p > e ∧ lockedValue_p ≠ v) then
23:        proposal_p ← nil
24:      else if valid(v) ∧ (lockedEpoch_p = −1 ∨ lockedValue_p = v) then
25:        proposal_p ← id(v)

26: Round PROPOSE(e_p):
27:  Send phase:
28:    broadcast ⟨PROPOSE, h_p, e_p, proposal_p, validEpoch_p⟩ to all processes
29:  Delivery phase:
30:    delivery ⟨PROPOSE, h_p, e_p, ∗⟩ from all processes
31:  Compute phase:
32:    if f + 1⟨∗, h_p, epoch, ∗, ∗⟩ ∧ epoch > e_p then
33:      e_p ← epoch
34:      goto PRE-PROPOSE(e_p)
35:    if 2f + 1⟨PROPOSE, h_p, e_p, id(v′)⟩ ∧ sendByProposer(h_p, e_p, v′) ∧ valid(v′) then
36:      lockedValue_p ← v′
37:      lockedEpoch_p ← e_p
38:      vote_p ← id(v′)
39:      validValue_p ← v′
40:      validEpoch_p ← e_p
41:    else
42:      vote_p ← nil

43: Round VOTE(e_p):
44:  Send phase:
45:    broadcast ⟨VOTE, h_p, e_p, vote_p⟩ and ⟨PROPOSE, h_p, e_p, ∗⟩ delivered to all processes
46:  Delivery phase:
47:    delivery ⟨VOTE, h_p, e_p, ∗⟩ and ⟨PROPOSE, h_p, e_p, ∗⟩ delivered from all processes
48:  Compute phase:
49:    if f + 1⟨∗, h_p, epoch, ∗, ∗⟩ ∧ epoch > e_p then
50:      e_p ← epoch
51:      goto PRE-PROPOSE(e_p)
52:    if 2f + 1⟨PROPOSE, h_p, e_p, id(v′′)⟩ ∧ sendByProposer(h_p, e_p, v′′) ∧ valid(v′′) then
53:      validValue_p ← v′′
54:      validEpoch_p ← e_p
55:    if 2f + 1⟨VOTE, h_p, e′, id(v′′)⟩ ∧ sendByProposer(h_p, e′, v′′) ∧ valid(v′′) ∧ decision_p[h_p] = nil then
56:      decision_p[h_p] = v′′
57:      h_p ← h_p + 1
58:      reset lockedValue_p, lockedEpoch_p, validValue, validEpoch_p to init values and empty message log
59:    else
60:      e_p ← e_p + 1
61:      if validValue_p ≠ nil then
62:        proposal_p ← validValue_p
63:      else
64:        proposal_p ← getValue()
```

Byzantine Eventual Synchronous System Using the formalism in Section 2, we present the Tendermint BFT Consensus algorithm [7] in an eventual synchronous setting, where Byzantine processes may exhibit an asymmetric behaviour.

To achieve the consensus in this setting two additional variables need to be used, (i) $lockedEpoch_p$ is an integer representing the last epoch where process p updated its $lockedValue$, and (ii) $validEpoch_p$ is an integer which represents the last epoch where process p updates $validValue_p$.

These two new variables are used to guarantee the agreement property during the asynchronous period. During such period different epochs may overlap at different processes, then it is needed to keep track of the relative epoch when a process locks in order to not accept "outdated" information, i.e., generated during a previous epoch.

Detailed description of the algorithm.

In Algorithm 3 we describe the algorithm to solve the Consensus as defined in Section 2 in an eventually synchronous system, and where Byzantine processes can exhibit asymmetric behaviour. This algorithm has been reported in [7] with the bugs fixed in [23]. The algorithm proceeds in 3 rounds for any given epoch e at height h:

1. Round PRE-PROPOSE (lines 9 - 25): The description of this round is the same as in the round-based case with asymmetric Byzantine. We highlight the fact that a "valid" for a correct process p with respect to the state in the eventual synchronous case is different than in the round-base case, since the variable $lockedEpoch_p$ is now taken into account.

2. Round PROPOSE (lines 26 - 42): The difference with the Algorithm 2 is that: when a correct process p updates $lockedValue_p$ (resp. $validValue_p$), it also update $lockedEpoch_p$ (resp. $validEpoch_p$) to the current epoch.

3. Round VOTE (lines 43 - 64): The differences with the Algorithm 2 are the following: (i) if a correct process p delivered at least $f + 1$ same type of messages from an epoch higher than the current one, p moves directly to the PRE-PROPOSE round of that epoch; and (ii) when a correct process p updates $validValue_p$, it also update $validEpoch_p$ to the current epoch.

We recall that for the evental synchronous setting, each process has a time-out for each round. If during a round process p does not deliver at least $2f + 1$ messages sent during that round (or the pre-proposal for the PRE-PROPOSE round), the corresponding time-out is increased.

Complexity. In the worst case scenario, the algorithm terminates when a pre-proposal can be proposed by more than $2f + 1$ correct processes, which eventually happens due to the round robin selection function. Thus, during the synchrony period, the protocol terminates in at most $2f$ epochs, while the optimum is f [14]. At each epoch, all processes broadcast messages, it follows that during one epoch the protocol uses $O(n^2)$ messages, then in the worst case scenario the message complexity is $O(fn^2)$.

Correctness Proof of Tendermint Algorithm in a Byzantine Eventual Synchronous Setting In this section, we prove the correctness of Algorithm 3 (Tendermint) in an eventual synchronous system. We recall that there are $3f+1$ processes, and less than f Byzantine processes in the system.

Lemma 1 (Validity) *In an eventual synchronous system, Tendermint verifies the following property: A decided value satisfies the predefined predicate denoted as* valid().

Proof The proof follows by construction. When a correct process decides a value (line 56), it checks before if that value is valid (line 55). So a correct process only decides a valid value. $\square_{Lemma\ 1}$

Lemma 2 (Integrity) *In an eventual synchronous system, Tendermint verifies the following property: No correct process decides twice.*

Proof The proof follows by construction. Before deciding (lines 55 - 58), a correct process p checks if there is not already a value decided ($decision_p[h_p] = nil$) for the current height ($i.e.$ line 55). If there is already a value decided ($decision_p[h_p] \neq nil$), there is no decision (lines 59 - 64). No correct process decides twice. $\square_{Lemma\ 2}$

Lemma 3 *In an eventual synchronous system, Tendermint verifies the following property: Correct processes only propose and vote once per epoch.*

Proof We prove this lemma by construction. A correct process proposes (line 28) and votes only once during the corresponding round (line 45), and at the end of the VOTE round, a process either changes epoch or height (lines 57 & 60). $\square_{Lemma\ 3}$

Lemma 4 *In an eventual synchronous system, Tendermint verifies the following property: At most one value can be proposed at least $2f + 1$ times per epoch, and at most one value can be voted at least $2f + 1$ times by epoch.*

Proof We prove this lemma by contradiction. Let v, v' such that $v \neq v'$. Since there are $3f + 1$ processes in the system, if v or v' gets at least $2f + 1$ proposals (resp. votes), it means that at least $f + 1$ processes propose (vote) for both v and v'. By assumption there are less than f Byzantine in the system, at least 1 correct process proposes (votes) both for v and v', which contradicts Lemma 3. It means that two different values cannot be proposed (resp. voted) at least $2f + 1$ times during the same epoch. $\square_{Lemma\ 4}$

Lemma 5 *Let v be a value, e an epoch, and $L^{v,e} = \{q : q$ $correct \wedge lockedValue_q = v \wedge lockedEpoch_q = e$ at the end of epoch $e\}$. In an eventual synchronous system, Tendermint verifies the following property: If $|L^{v,e}| \geq f + 1$ then no correct process p will have $lockedValue_p \neq v \wedge lockedEpoch_p \geq e$, at the end of each epoch $e' > e$, moreover a process in $L^{v,e}$ only proposes v or nil for each epoch $e' > e$.*

Proof Let v be a value, e an epoch, and $L^{v,e} = \{q : q$ correct$\wedge lockedValue_q = v \wedge lockedEpoch_q = e$ at the end of epoch $e\}$, we assume that $|L^{v,e}| \geq f + 1$. We prove the theorem by induction:

- Initialization: At the end of epoch e, by assumption, we have that $|L^{v,e}| \geq f + 1$. Since a correct process p ($p \in L^{v,e}$) updates $lockedValue_p$ to v during epoch e, it means that p delivered $2f + 1$ proposals for the value v (lines 35 - 37). By Lemma 4, at most one value can have at least $2f + 1$ proposals during epoch e, and since v has at least $2f + 1$ proposes, no process q update $lockedValue_q$ to a value $v' \neq v$ during epoch e. At the end of e, $lockedValue_q \neq v \vee lockedEpoch_q < e$.

- Induction: Let $a \geq 1$, we assume that $\forall p \in L^{v,e}$, $lockedValue_p = v$ at the end of each epoch between e and $e+a$, we also assume that if a value was proposed at least $2f+1$ times during these epochs it was either v or *nil*. We prove that at the end of epoch $e+a+1$, no correct process q will have $lockedValue_q = v' \wedge lockedEpoch_q = e+a+1$ with $(v' \neq v)$.

  Let $p \in L^{v,e}$, p delivers a pre-proposal for v, then p will set $proposal_p$ to v, and will propose v since $lockedValue_p = v$ (lines 19 - 25 & 28), in any other case, if p does not deliver a pre-proposal, or delivers a pre-proposal for a value $v' \neq v$, it will set $proposal_p$ to *nil* and will propose *nil* (lines 19 - 25 & 28), since $valid(nil) = \mathsf{false}$ and by assumption, there is no $e' \in \{e, \ldots, e+a\}$ where there were at least $2f+1$ proposals for a value $v' \neq v$, and $lockedEpoch_p \geq e$. All processes in $L^{v,e}$ will then propose v or *nil* during epoch $e+a+1$. By Lemma 3, correct processes only propose once per epoch, at least $f+1$ processes (in $L^{v,e}$) propose v or *nil* and messages cannot be forged, the only values that can get at least $2f+1$ proposals for the epoch $e+a+1$ are v and *nil*. If a correct process q delivers at least $2f+1$ proposals for v, it sets $lockedValue_q$ to v and $lockedEpoch_q$ to $e+a+1$ (lines 35 - 37), otherwise it does not change $lockedValue_q$ nor $lockedEpoch_q$ (line 42). At the end of epoch $e+a+1$, there is no correct process q such that $lockedValue_q \neq v \wedge lockedEpoch_q = e+a+1$. Moreover, processes in $L^{v,e}$, only propose v or *nil* during epoch $e+a+1$.

We proved that if $|L^{v,e}| \geq f+1$, no correct process p will have $lockedValue_p \neq v \wedge lockedEpoch_p \geq e$, moreover a process in $L^{v,e}$ only proposes v or *nil* for each epoch $e' > e$.

$\square_{Lemma\ 5}$

Lemma 6 (Agreement) *In an eventual synchronous system, Tendermint verifies the following property: If there is a correct process that decides a value v, then eventually all the correct processes decide v.*

Proof Let p be a correct process. We assume that p is the first correct process that decides, and we assume that it decides value v during epoch e. To decide, p delivered at least $2f+1$ votes for v for epoch e. Since there are less that f Byzantine processes, and by Lemma 3 correct processes can only vote once per epoch, so at least $f+1$ correct processes voted for v during epoch e, so we have $|L^{v,e}| = |\{q : q \text{ correct} \wedge lockedValue_q = v \wedge lockedEpoch_q = e \text{ at the end of epoch } e\}| \geq f+1$. By Lemma 5 processes in $L^{v,e}$ only propose v or *nil* during each epoch after e, and no correct process q will have $lockedValue_p \neq v \wedge lockedEpoch_p \geq e$. Thanks to the best effort broadcast guarantees, all correct processes will eventually deliver the $2f+1$ votes for v from epoch e.

If a correct process q does not decide before delivering these votes, when delivering them, it will decide v (lines 55 - 56). Otherwise, it means that q decides before delivering the votes from epoch e.

By contradiction, we assume that q decides a value $v' \neq v$ during an epoch $e' > e$, so q delivered at least $2f+1$ votes for v' during epoch e' (lines 55 - 56). Since a correct process only votes once by Lemma 3, there are less that f Byzantine processes and the messages are unforgeable, at least $f+1$ correct processes vote for v'. A correct process votes a non-*nil* value if that value was proposed at least $2f+1$ times during the current epoch (lines 35 - 45). By Lemma 3 a correct process only proposes once, there are less that f Byzantine processes and the messages are unforgeable, so at least $f+1$ correct processes proposed v' during e'. Since $e' > e$ and $|L^{v,e}| \geq f+1$, by Lemma 5 there are at least $f+1$ processes that propose v or *nil* during epoch e'. Even if all the $2f$ processes

remaining proposes v', there cannot be $2f+1$ proposals for v', which is a contradiction. So q cannot decide $v' \neq v$ after epoch e and we assume that e is the first epoch where a correct process decides. $\square_{Lemma\ 6}$

Lemma 7 *In an eventual synchronous system, if there is an epoch after which when a correct process broadcasts a message during a round r, it is delivered by all correct processes during the same round r, Tendermint verifies the following property: If a correct process p updates $lockedValue_p$ to a value v during epoch e, then at the end of the epoch e, all correct processes have $validValue = v$ and $validEpoch = e$.*

Proof We prove this lemma by construction.

Let e be the epoch after which when a correct process broadcasts a message during a round r, it is delivered by all correct processes during the same round r. Let p be a correct process, we assume that at the end of epoch $e' \geq e$, p has $lockedValue_p = v$ and $lockedEpoch_p = e'$, it means that p delivered at least $2f + 1$ proposals for v during epoch e' (lines 35 - 37). When p votes, it sends all proposals delivered during PROPOSE round (line 45), and all the correct processes will deliver these proposals for v. Let q be a correct process, since q will deliver at least $2f + 1$ proposals for v and epoch e' during the VOTE round, it will set $validValue_q = v$ and $validEpoch_q = e'$ (lines 52 - 54). $\square_{Lemma\ 7}$

Lemma 8 (Termination) *In an eventual synchronous system, Tendermint verifies the following property: Every correct process eventually decides some value.*

Proof By construction, if a correct process does not deliver more than $2f + 1$ messages (or 1 from the proposesr in the PRE-PROPOSE round) from different processes during the corresponding round, it increases the duration of its rounds, so eventually during the synchronous period of the system all the correct processes will deliver the pre-proposal, proposals and votes from correct processes respectively during the PRE-PROPOSE, PROPOSE and the VOTE round. Let e be the first epoch after that time.

If a correct process decides before e, by Lemma 6 all correct processes decide which ends the proof. Otherwise at the beginning of epoch e, no correct process decides yet. Let p be the proposer of e. We assume that p is correct and pre-propose v, v is valid since $getValue()$ always return a valid value (lines 7, 64), and $validValue_p$ is always valid (lines 35 & 52). We have 2 cases:

- Case 1: At the beginning of epoch e, $|\{q : q \text{ correct} \wedge (lockedEpoch_q \leq validEpoch_p \vee lockedValue_q = v)\}| \geq 2f + 1$.

  Let q be a correct process such that $lockedEpoch_q \leq validEpoch_p \vee lockedValue_q = v$, after the delivery of the pre-proposal v from p, q will update $proposal_q$ to v (lines 19 - 25). During the PROPOSE round, q proposes v (line 28), and since there are at least $2f+1$ similar correct processes they will all propose v, and all correct processes will deliver at least $2f+1$ proposals for v (line 30).

  Correct processes will set *vote* to v (lines 35 - 28), will vote v, and will deliver these votes, so at least $2f + 1$ of votes (lines 45 & 47). Since we assume that no correct processes decide yet, and since they deliver at least $2f + 1$ votes for v, they will decide v (lines 55 - 56).

- Case 2: At the beginning of epoch e, $|\{q : q \text{ correct} \wedge (lockedEpoch_q \leq validEpoch_p \vee lockedValue_q = v)\}| < 2f + 1$.

  Let q be a correct process such that $lockedEpoch_q > validEpoch_p \wedge lockedValue_q \neq v$, when p will make the pre-proposal, q will set $proposal_q$ to *nil* (line 23) and will propose *nil* (line 28).

  By counting only the propose value of the correct processes, no value will have at least $2f + 1$ proposals for v. There are two cases:

  - No correct process delivers at least $2f + 1$ proposals for v during the PROPOSE round, so they will all set their *vote* to *nil*, vote *nil* and go to the next epoch without changing their state (lines 42 & 45 - 47 & 59 - 64).
  - If there are some correct processes that delivers at least $2f + 1$ proposals for v during the PROPOSE round, which means that some Byzantine processes send proposals for v to those processes.

    As in the previous case, they will vote for v, and since there are $2f + 1$ of them, all correct processes will decide v. Otherwise, there are less than $2f + 1$ correct processes that deliver at least $2f + 1$ proposals for v. Only them will vote for v (line 45). Without Byzantine processes, there will be less than $2f + 1$ vote for v, no correct process will decide (lines 55 - 58) and they will go to the next epoch, if Byzantine processes send votes for v to a correct process such as it delivers at least $2f + 1$ votes for v during VOTE round, then it will decide (lines 55 - 56), and by Lemma 6 all correct processes will eventually decide.

    Let q_1 be one of the correct processes that delivers at least $2f + 1$ proposals for v during PROPOSE round, it means that at $lockedValue_{q_1} = v$ and $lockedEpoch_{q_1} = e$, by Lemma 7 at the end of epoch e, all correct processes will have $validValue = v$ $validEpoch = e$.

  If there is no decision, either no correct process changes its state, otherwise all correct processes change their state and have the same $validValue$ and $validRound$, eventually a proposer of an epoch will satisfy the case 1, and that ends the proof.

If p the proposer of epoch e is Byzantine and more than $2f + 1$ correct processes delivered the same message during PRE-PROPOSE round, and the pre-proposal is valid, the situation is like p was correct. Otherwise, there are not enough correct processes that delivered the pre-proposal, or if the pre-proposal is not valid, then there will be less than $2f + 1$ correct processes that will propose that value, which is similar to the case 2.

Since the proposer is selected in a round robin fashion, a correct process will eventually be the proposer, and a correct process will decide. $\square_{Lemma\ 8}$

Theorem 1 *In an eventual synchronous system, Tendermint implements the Consensus.*

Proof The proof follows directly from Lemmas 1, 2, 6 and 8.

$\square_{Theorem\ 1}$

4 Conclusion

The contribution of this work is twofold. First, it analyzes the Tendermint consensus protocol and provides detailed proof of its correctness. Second, it dissects such protocol to link all the algorithmic techniques employed to the system model to cope with. We believe that this methodology can contribute in making consensus algorithms more understandable for developers and practitioners.

References

[1] Ittai Abraham, Dahlia Malkhi, Kartik Nayak, Ling Ren, and Alexander Spiegelman. Solidus: An incentive-compatible cryptocurrency based on permissionless byzantine consensus. *CoRR, abs/1612.02916*, 2016.

[2] Marcos K Aguilera. A pleasant stroll through the land of infinitely many creatures. *ACM Sigact News*, 35(2):36–59, 2004.

[3] Yackolley Amoussou-Guenou, Antonella Del Pozzo, Maria Potop-Butucaru, and Sara Tucci Piergiovanni. Correctness and fairness of tendermint-core blockchains. *CoRR*, abs/1805.08429, 2018.

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

[5] Elli Androulaki at al. 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.

[6] 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.

[7] Ethan Buchman, Jae Kwon, and Zarko Milosevic. The latest gossip on bft consensus. *arXiv preprint arXiv:1807.04938*, 2018.

[8] M. Castro and B. Liskov. Practical Byzantine Fault Tolerance. In *Proceedings of the Symposium on Operating Systems Design and Implementation (OSDI)*, 1999.

[9] 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.

[10] Tyler Crain, Vincent Gramoli, Mikel Larrea, and Michel Raynal. Dbft: Efficient byzantine consensus with a weak coordinator and its application to consortium blockchains. *arXiv preprint arXiv:1702.03068*, 2017.

[11] 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.

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

[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] 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.

[15] Michael J Fischer, Nancy A Lynch, and Michael S Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM (JACM)*, 32(2):374–382, 1985.

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

[17] 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.

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

[19] Jae Kwon and Ethan Buchman. Tendermint. `https://tendermint.readthedocs.io/en/master/specification.html` (visited on 2018-05-22).

[20] L. Lamport, R. Shostak, and M. Pease. The byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, July 1982.

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

[22] João Sousa, Eduardo Alchieri, and Alysson Bessani. State machine replication for the masses with bft-smart. 2013.

[23] Tendermint. Tendermint: correctness issues. `https://github.com/tendermint/spec/issues` (visited on 2018-09-24).

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