Dissecting Tendermint

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

Dissecting Tendermint

Yackolley Amoussou-Guenou[1,2], Antonella Del Pozzo[1], Maria Potop-Butucaru[2], and Sara Tucci-Piergiovanni[1]

[1] CEA LIST, PC 174, Gif-sur-Yvette, 91191, France
[2] Sorbonne Université, CNRS, LIP6, Paris, France

Abstract. In this paper we analyze Tendermint proposed in [10], one of the most popular blockchains based on PBFT Consensus. Our methodology consists in identifying the algorithmic principles of Tendermint necessary for a specific communication model. The current paper dissects Tendermint under two system communication models: synchronous and eventually synchronous communication models. This methodology allowed to identify bugs [6] in preliminary versions of the protocol ([21], [10]) and to prove its correctness under the most adversarial conditions: an eventually synchronous communication model and Byzantine faults.

Keywords: BFT Consensus · Blockchain · Tendermint

1 Introduction

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. 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 efficiency is targeted. Moreover proof of work solutions guarantee the existence of an unique chain only with high probability which is the major drawback for using blockchains in industrial applications. That is, forks even though they are rare do still happen with an impact on the consistency guarantees offered by the system and Consensus algorithms play an important role to prevent inconsistencies. Therefore, alternatives to proof-of-work have been recently considered and 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.

Since Blockchain is an append-only 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 employs the proof-of-work mechanism [16]. 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 last block then processes may have blockchains with different suffix as long as the conflict is unsolved. In [7] the authors proved that consensus [22] is necessary in order to avoid forks. Consensus can be informally described as follows: given a set of processes proposing each of them a value then, after a finite amount of time, all processes agree on the same value, chosen among the proposed ones.

In this paper we analyze Tendermint proposed in [10] as one of the most promising but not fully analyzed protocols so far that implements Byzantine fault tolerant Consensus. Tendermint targets an eventual synchronous system [15], 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 considered system model: synchronous round-based model in presence of Byzantine faults; and eventual synchronous communication model in presence of Byzantine faults. For each type of model we provide the corresponding algorithm (a variant of Tendermint [10]). Finally, we provide a proved correct protocol specification of [10] in the eventual synchronous setting in presence of Byzantine faults and computed its complexity. This methodology allowed to identify bugs [6] in the preliminary versions of the protocol ([21], [10]) that now have been solved.

2 Model

The system is composed of an infinite set Π of sequential processes, namely $\Pi = \{p_1, \dots\}$; i is called the *index* of p_i. *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* [4], 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 [20] that is in charge of implementing the selection of V.

Time assumptions on communication. The processes communicate by exchanging messages through an eventually synchronous network [15]. *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 [23] 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 $f < n/3$ and more precisely we consider $n = 3f + 1$.

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 [8].

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 exits from the current round and starts the next round. Each round has a finite 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 duration of the Delivery phase, and so of the round is δ. We assume that processes have no access to a global clock but have access to local clocks, these clocks might not be synchronized with each other but are allowed to have bounded clock skew.

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[12]**, a decided value is valid, it satisfies the predefined predicate denoted valid().

3 Tendermint Algorithms

Tendermint BFT Consensus protocol [21, 26, 10] is a variant of PBFT consensus at the core layer under the Tendermint blockchain.

The algorithm follows the rotating coordinator paradigm i.e., for each new block to be appended there is a proposer, chosen among the validators, that proposes a block. If the block is not decided then a new proposer is selected and so on, until a block is decided by all the correct processes and the consensus terminates.

Basic principles of the protocol. Each block in the blockchain is characterized by its height h, which is the distance in terms of blocks from the genesis block, which is at height 0. For each new height, the two protocols (Algorithm 2 and Algorithm 4) share a common algorithmic structure, they proceed in *epoch*s, and each epoch e consists in three rounds: the *PRE-PROPOSE* round; the *PROPOSE* round; and the *VOTE* round. During the PRE-PROPOSE round, the proposer pre-proposes a value v to all the other validators. During the PROPOSE round, if a validator accepts v then it proposes such value. If a validator receives *enough* proposals for the same value v then it votes for v during the VOTE round. Finally, if a validator receives *enough* votes for v, it decides on v. In this case, *enough* means at least $2f+1$ occurrences of the same value from $2f+1$ different processes and from each process only the first value delivered for each round is considered, (cf. Algorithm 1).

If the proposer is correct then it pre-proposes the same value to all the $2f + 1$ correct processes. All the $2f + 1$ correct processes propose such value, it follows that all the $2f + 1$ correct processes vote for such value and decide for it. If the proposer is Byzantine faulty it can pre-propose different values to different correct processes, creating a partition in the proposal value set collected by validators. Depending on what the remaining Byzantine faulty processes do, some correct processes may decide on a value v and some other may not [3], then a new epoch starts. To do not violate the agreement property, processes that have not decided yet in the previous epoch must only decide for v, for this reason processes, before vote for some value v, lock on that value, i.e., it will refuse to propose a further pre-proposed value different than v.

Information from one epoch to the next. $lockedValue$ and $validValue$ variables[4] carry the potentially decided value from one epoch to the next one. The $lockedValue$ idea is the following. If one correct process decides on v, it means that it collected $2f + 1$ votes for v during the VOTE phase, since there are at most f Byzantine faulty processes thus there are at least $f + 1$ correct processes that voted for v and those processes must not vote for any other different value

[3] Since there are $3f + 1$ validators, there cannot be two different values that collect $2f + 1$ distinct votes in the same epoch.

[4] $validValue$ was not present in the previous version of Tendermint, that was suffering from the Live Lock bug [1].

than v. For this reason if a process delivers $2f + 1$ proposals for v during the PROPOSE round it sets $lockedValue$ to v. Since each new pre-proposed value v' is proposed if v' is equal to $lockedValue$ or $validValue$ (not true for at lest $f + 1$ correct processes that set $lockedValue$ to v), then there can be at most $2f$ possible proposals for v' that are not enough to lock and vote for v', i.e., it is not possible to decide for any value different than v. On the other side, if no correct process decided yet, Byzantine faulty processes may force different correct processes to lock on different values. Let us consider a scenario where the proposer is Byzantine faulty and proposes v to $f + 1$ correct processes and then f Byzantine processes make $x \leq f$ of them lock on v and a similar scenario can happen with another value v' so that we can have different correct processes, let us say $y \leq f$ locked on a different value. If any new new pre-proposal is checked only against the $lockedValue$ then a correct process locked on a value v refuses (does not propose) all values different from v, it means that when some correct process is locked, the proposer needs to propose some of the value on which the correct processes are locked on, but such value, in order to be accepted cannot be checked only against the $lockedValue$ because we may never have enough correct processes proposing such value. For this reason processes keep track of the $validValue$ and by construction of the algorithm all correct processes have the same $validValue$ at the end of the epoch (in the synchronous period). Such value is then used to set the value to pre-propose and it is further used along with $lockedValue$ to accept or not a pre-proposed value.

Messages syntax. When p_i broadcasts a message $\langle TAG, h, e, m\rangle$, where m contains a value, we say that p_i proposes or votes m if TAG=`PROPOSE` or TAG=`VOTE` respectively. When p_i broadcasts $\langle$`PRE-PROPOSE`$, h, e, m, e'\rangle$, where e' is an epoch and h is the height. We also say that p_i pre-proposes m with an epoch e'.
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. $lockedValue$ stores a value which is potentially decided by some other process. If process p delivers more than $2f + 1$ proposes for the same value v during its PROPOSE round, it sets its $lockedValue_p$ to v. $validValue$ stores a value which is potentially decided by some other process. If the process p delivers 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 $lockedValue$. The latter two variables are used as follows: if p is the next proposer then p pre-proposes $validValid$ if different from nil. Otherwise, if p is a validator, it checks the new pre-proposal against $lockedValue$ and $validValid$ if those are different from nil.
Functions. We denote as $Value$ the set containing all blocks, and as $MemPool$ the set containing all the transactions.
- proposer : $Height \times Epoch \rightarrow V \subseteq \Pi_\rho$ is a deterministic function which gives

the proposer out of the validators set 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 : $MemPool \rightarrow Value$ is an application dependent predicate which gives a valid value.
- id : $Value \rightarrow Hash\_of\_Value$ 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.
Everything defined above is common to the two algorithms. In each section we specify the data structures relative to a specific version of the algorithm.
- $2f + 1 : PROPOSE^* \cup VOTE^* \rightarrow Bool$: checks if there are at least $2f + 1$ proposals/vote in the given set.

Algorithm 1 Messages management for process p

```
1: upon ⟨TYPE, h, e, message⟩ from process q do
2:   if ∄c : (⟨TYPE, h, e, c⟩, q) ∈ messagesSet then
3:     messagesSet ← messagesSet ∪ (⟨TYPE, h, e, message⟩, q)
```

Byzantine Synchronous System. This section presents the Algorithm 2 and Algorithm 1 that solve Consensus in a synchronous model in presence of Byzantine faulty processes.

Detailed description of the algorithm. In Algorithms 1 - 3 we describe the algorithm to solve the Consensus as defined in Section 2 in a synchronous system in presence of Byzantine failures. The algorithm proceeds in 3 rounds for any given epoch e at height h:
- Round PRE-PROPOSE (lines 8 - 25, Algorithm 2): 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 and if to accept it with respect to the values in $validValue$ and $lockedValue$, and if the pre-proposal is accepted and valid, q sets its proposal $proposal_q$ to the pre-proposal, otherwise it sets it to nil.
- Round PROPOSE (lines 1 - 13, Algorithm 3): 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

Algorithm 2 Simplified Algorithm part 1 for height h executed at process p

```
1: Initialization:
2:    e_i := 0                                                    /* current epoch number */
3:    decision_i := nil
4:    lockedValue_i := nil; validValue_i := nil
5:    proposal_i := getValue()   /* This variable stocks the value the process will (pre-)propose */
6:    v_i := nil                          /* Local variable stocking the pre-preposal if delivered */
7:    vote_i := nil

8: Round PRE-PROPOSE(e_i):
9:    Send phase:
10:      if decision_i ≠ nil then
11:        ∀x ∈ Vote_|decision, broadcast x /* x is on the form ⟨VOTE, h, epochOfDecision, ∗⟩, and
           epochOfDecision if the epoch number where the decided block where pre-proposed */
12:      if proposer(h, e_i) = p then
13:        broadcast ⟨PRE − PROPOSE, h, e_i, proposal_i⟩ to all processes
14:   Delivery phase:
15:      while (timerPrePropose not expires) do
16:        if ∃v_j, e_j : sendByProposer(h, e_i, v_j) then
17:          v_i ← v_j                                          /* v_j is the value of the proposal */
18:   Compute phase:
19:      if valid(v_i) ∧ validValue_i = nil then
20:        proposal_i ← id(v_i)
21:      else
22:        if !valid(v_i) ∨ v_i ∉ {lockedValue_i, validValue_i} then
23:          proposal_i ← nil
24:        else
25:          proposal_i ← id(v_i)                  /* Note that id is only defined on valid value */
```

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.

- Round VOTE (lines 14 - 31, Algorithm 3): 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 it 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 Simplified Algorithm part 2 for height h executed at process p

```
1: Round PROPOSE(e_i):
2:   Send phase:
3:     if proposal_i ≠ nil then
4:       broadcast ⟨PROPOSE, h, e_i, proposal_i⟩ to all processes
5:   Delivery phase:
6:     while (timerPropose not expires) do{}
7:   Compute phase:
8:     if ∃v' : 2f + 1⟨PROPOSE, h, e_i, id(v')⟩ ∧ sendByProposer(h, e_i, v') then
9:       lockedValue_i ← v'
10:      validValue_i ← v'
11:      vote_i ← id(v')
12:    else
13:      vote_i ← nil

14: Round VOTE(e_i):
15:   Send phase:
16:     if vote_i ≠ nil then
17:       broadcast ⟨VOTE, h, e_i, vote_i⟩
18:     ∀x ∈ Propose_|e_i, broadcast x   /* x is on the form ⟨PROPOSE, h, e_i, ∗⟩; Propose_|e_i is the
        set of PROPOSE messages delivered corresponding to the phase PROPOSE(e_i) */
19:   Delivery phase:
20:     while (timerVote not expires) do{}
21:   Compute phase:
22:     if ∃v_d, e_d : 2f + 1⟨VOTE, h, e_d, id(v_d)⟩ ∧ sendByProposer(h, e_d, v_d) ∧ decision_i = nil then
23:       validValue_i ← v_d
24:     if 2f + 1⟨VOTE, h, r, id(v_d)⟩ ∧ sendByProposer(h, e_i, v_d) ∧ valid(v_d) ∧ decision_i = nil then
25:       decision_i = v_d
26:     else
27:       e_i ← e_i + 1
28:       if validValue_i ≠ nil then
29:         proposal_i ← validValue_i
30:       else
31:         proposal_i ← getValue()
```

Algorithm 4 Tendermint Consensus part 1 for height h executed at process p

```
1: Initialization:
2:   e_i := 0                                                    /* Current epoch number */
3:   decision_i := nil
4:   lockedValue_i := nil; validValue_i := nil
5:   lockedEpoch_i := −1; validEpoch_i := −1
6:   proposal_i := getValue()   /* This variable stocks the value the process will (pre-)propose */
7:   v_i := nil                          /* Local variable stocking the pre-preposal if delivered */
8:   validEpoch_j := nil                   /* Local variable stocking the proposer's validEpoch */
9:   vote_i := nil                      /* This variable stock the value the process will vote for */
10:  timeoutPrePropose := Δ_Pre-propose; timeoutPropose := Δ_Propose; timeoutVote := Δ_Vote

11: Round PRE-PROPOSE:
12:   Send phase:
13:     if decision_i ≠ nil then
14:       ∀x ∈ Vote_|decision, broadcast x   /* x is on the form ⟨VOTE, h, epochOfDecision, ∗⟩ */
15:     if proposer(h, e_i) = p_i then
16:       broadcast ⟨PRE − PROPOSE, h, e_i, proposal_i, validEpoch_i⟩
17:   Delivery phase:
18:     set timerPrePropose to timeoutPrePropose
19:     while (timerPrePropose not expires) ∧ ¬(∃v_j, e_j : sendByProposer(h, e_i, v_j, e_j)) do
20:       if ∃v_j, e_j : sendByProposer(h, e_i, v_j, e_j) then
21:         v_i ← v_j                                          /* v_j is the value of the proposal */
22:         validEpoch_j ← e_j                    /* e_j is the validEpoch sent by the proposer */
23:     if ¬(∃v, epochProp : sendByProposer(h, e_i, v, epochProp)) then
24:       timeoutPrePropose ← timeoutPrePropose + 1
25:   Compute phase:
26:     if 2f + 1 ⟨PROPOSE, h, validEpoch_j, id(v_i)⟩ ∧ validEpoch_j ≥
        lockedEpoch_i ∧ validEpoch_j < e_i ∧ valid(v_i) then
27:       proposal_i ← id(v_i)        /* Note that the function id is only applied on value value. */
28:     else
29:       if !valid(v_i) ∨ (lockedEpoch_i > e ∧ lockedValue_i ≠ v_i) then
30:         proposal_i ← nil                          /* Note that valid(nil) is set to false */
31:       if valid(v_i) ∧ (lockedEpoch_i = −1 ∨ lockedValue_i = v_i) then
32:         proposal_i ← id(v_i)
```

Algorithm 5 Tendermint Consensus part 2 for height h executed at process p

```
1: Round PROPOSE:
2:   Send phase:
3:     if proposal_i ≠ nil then
4:       broadcast ⟨PROPOSE, h, e_i, proposal_i⟩
5:     broadcast ⟨HeartBeat, PROPOSE, h, e_i⟩
6:   Delivery phase:
7:     set timerPropose to timeoutPropose
8:     while (timerPropose not expires) ∧ ¬(2f + 1⟨HeartBeat, PROPOSE, h, e_i⟩) do{}   /* Note
       that the HeartBeat messages should be from different processes */
9:     if ¬(2f + 1⟨HeartBeat, PROPOSE, h, e_i⟩) then
10:      timeoutPropose ← timeoutPropose + 1
11:  Compute phase:
12:    if ∃v' : 2f + 1⟨PROPOSE, h, e_i, id(v')⟩ ∧ sendByProposer(h, e_i, v') then
13:      lockedValue_i ← v'
14:      lockedEpoch_i ← e_i
15:      validValue_i ← v'
16:      validEpoch_i ← e_i
17:      vote_i ← id(v')
18:    else
19:      vote_i ← nil

20: Round VOTE:
21:   Send phase:
22:     if vote_i ≠ nil then
23:       broadcast ⟨VOTE, h, e_i, vote_i⟩
24:     ∀x ∈ Propose_{|e_i}, broadcast x
25:     broadcast ⟨HeartBeat, VOTE, h, e_i⟩
26:   Delivery phase:
27:     set timerVote to timeoutVote
28:     while (timerVote not expires) ∧ ¬(2f + 1⟨HeartBeat, VOTE, h, e_i⟩) do{}
29:     if ¬(2f + 1⟨HeartBeat, VOTE, h, e_i⟩) then
30:       timeoutVote ← timeoutVote + 1
31:   Compute phase:
32:     if ∃v'' : 2f + 1⟨PROPOSE, h, e_i, id(v'')⟩ ∧ sendByProposer(h, e_i, v'') then
33:       validValue_i ← v''
34:       validEpoch_i ← e_i
35:     if ∃v_d, e_d : 2f + 1⟨VOTE, h, e_d, id(v_d)⟩ ∧ sendByProposer(h, e_d, v_d) ∧ decision_i = nil then
36:       decision_i = v_d
37:     else
38:       e_i ← e_i + 1
39:       v_i ← nil
40:       if validValue_i ≠ nil then
41:         proposal_i ← validValue_i
42:       else
43:         proposal_i ← getValue()
```

Byzantine Eventual Synchronous System. This section presents the Algorithm 1 and Algorithms 4 - 5 that solve Consensus in an eventually synchronous model in presence of Byzantine faulty processes. This algorithm has been reported in [10] with the bugs fixed in [25].

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 do not violate 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 generated during a previous epoch.

Moreover, a round duration management mechanism needs to be introduced, i.e. increasing timeouts. In the previous algorithm, rounds were lasting δ, the known message delay. In an eventually synchronous system such approach is not feasible, since during the asynchronous period messages may take unbounded delay before being delivered. It follows that, since there are at most f Byzantine faulty processes, when a process delivers messages from $n - f$ different processes it can terminate the delivery phase, but such phase may last an unbounded time. On the contrary, in the PRE-PROPOSE round only the proposer is sending a message, and generally messages may take lot of time before being delivered, for such reasons timeouts needs to be used in order to manage the rounds duration and adapted to be message delays, such that once the system enters in the synchronous period, rounds last enough for messages send during the round to be delivered before the end of it.

Detailed description of the algorithm.

The algorithm proceeds in 3 rounds for any given epoch e at height h. The description is mainly the same as before, thus in the following we underline just the differences:

- Round PRE-PROPOSE (lines 11 - 32, Algorithm 4): The description of this round is mainly the same as before. We highlight the fact that a correct process p takes into account also $lockedEpoch_p$ in order to accept a pre-proposed value.
- Round PROPOSE (lines 1 - 19, Algorithm 5): When a correct process p updates $lockedValue_p$ (resp. $validValue_p$), it also update $lockedEpoch_p$ (resp. $validEpoch_p$) to the current epoch.
- Round VOTE (lines 20 - 43, Algorithm 5): 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 when a correct process p updates $validValue_p$, it also update $validEpoch_p$ to the current epoch.

We recall that 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. Those messages can be values or heartbeats, in the case in which a correct process has not a value to propose or vote.

Complexity. Let us consider the following worst case scenario after the asynchronous period (i.e., after τ), in which in the first f epochs, $e_{i+1}, \ldots, e_{i+f}$, there are f Byzantine proposers that make lock only one correct process at each epoch on f different values with different $lockedEpoch$, $e_{i+1}, \ldots, e_{i+f}$. Let p_j be the last correct process that locked, and let v such value with $lockedEpoch = e_{i+f}$. Then all the other correct processes have $validValue$ set to v and $validEpoch$ set to e_{i+f}. This happens thanks to the fact that when a correct process locks on a value then at the end of the epoch all correct process sets their $validValue$ to that value. The algorithm terminates when a pre-proposal is proposed and voted by more than $2f$ correct processes. i.e, when the pre-proposed value has $validEpoch$ greater equal than the process $lockedEpoch$. Thus, during the pe-

riod of synchrony, the first correct proposer that proposes leads the algorithm to terminate in $f + 1$ rounds. Let us consider the case in which there f correct processes locked on f different values with different *lockedEpoch* before τ. Let us assume that p_j is the last correct process that locked on a value v, thus it has the highest *lockedEpoch* but not all the correct processes have the $validValue$ set to v (due to the asynchronous communication). Let us consider that after τ the first f proposers are Byzantines and stay silent. The following proposers are correct but their pre-propose value might not be accepted by enough correct processes as long as p_j, which the highest *validEpoch* and *lockedEpoch* proposes. Which eventually happens due to the round robin selection function. Thus, the protocol terminates in a number of epochs proportional to the number of validators, while the optimum for the worst case scenario is $f + 1$ [17]. Considering that at each epoch, all processes broadcast messages, it follows that during one epoch the protocol uses $O(n^2)$ messages, thus in the worst case scenario the message complexity is $O(n^3)$.

Correctness Proof of Tendermint Algorithm in a Byzantine Eventual Synchronous Setting. In this section, we prove the correctness of Algorithm 4 - 5 (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 36), it checks before if that value is valid (line 35). 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 35 - 36), 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 35). If there is already a value decided ($decision_p[h_p] \neq nil$), there is no decision (lines 37 - 43). No correct process decides twice. Moreover, note that a correct process exit the algorithm, the epoch after it has decide (line 13, Algorithm 4). $\square_{Lemma\ 2}$

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. In Algorithm 5, a correct process proposes (line 4) and votes only once during the corresponding round (line 23), and at the end of the VOTE round, a process either changes epoch or height (lines 36 & 38). $\square_{Lemma\ 3}$

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 12 - 14, Algorithm 5). 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 26 - 32, Algorithm 4 & 4, Algorithm 5), 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 26 - 32, , Algorithm 4 & 4, Algorithm 5), 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 12 - 14), otherwise, it does not change $lockedValue_q$ nor $lockedEpoch_q$ (line 19, Algorithm 5). 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, since when a correct process decides, it sends back all votes it delivered than makes it decide (line 13, Algorithm 4).

If a correct process q does not decide before delivering these votes, when delivering them, it will decide v (lines 35 - 36). 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 35 - 36). 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 12 - 23, Algorithm 5). 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 12 - 14, Algorithm 5). When p votes, it sends all proposals delivered during PROPOSE round (line 23, Algorithm 5), 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 32 - 34, Algorithm 5). $\square_{Lemma\ 7}$

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 6, Algorithm 4 & 43, Algorithm 5), and $validValue_p$ is always valid (lines 12 & 32, Algorithm 5). 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 26 - 32, Algorithm 4). During the PROPOSE round, q proposes v (line 4, Algorithm 5), 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 7, Algorithm 5).
  Correct processes will set $vote$ to v (lines 12 - 4, Algorithm 5), will vote v, and will deliver these votes, so at least $2f + 1$ of votes (lines 23 & 26,

Algorithm 5). 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 35 - 36, Algorithm 5).

- 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 30, Algorithm 4) and will propose *nil* (line 4, Algorithm 5).
  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 19 & 23 - 26 & 37 - 43, Algorithm 5).
  - 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 23, Algorithm 5). Without Byzantine processes, there will be less than $2f + 1$ vote for v, no correct process will decide (lines 35 - 36, Algorithm 5) 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 35 - 36, Algorithm 5), 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 Related work

The Consensus problem, as proved in [22], cannot be solved in presence of f Byzantine faulty processes if the overall number of processes n is less than $3f+1$ in a synchronous message-passing system (where the message delivery delay is upper bounded). Moreover, as proved in the seminal FLP paper [18], Consensus cannot be solved 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. In between those impossibility results, it is still possible to solve Consensus in an asynchronous setting, either adding randomness [9] (which also proved the impossibility result for $n \leq 3f$ for any asynchronous solution) or partial synchrony as in Dwork et al. [15] (DLS) where BFT Consensus is solved an eventual synchronous message-passing system (there is a time after which there is an upper bound on the message delivery delay). DLS preserves safety during the asynchronous period and the termination only after τ, when the message transfer delay becomes bounded. Unfortunately such protocol is not practical, it has a message complexity of $O(n^4)$ per epoch and $O(n)$ epochs before deciding. Finally, Castro and Liskov proposed the PBFT [11], a protocol that optimizes the performances of the previous solution. Indeed, such protocol is leader-based, so that if the leader is correct the complexity boils down to $O(n^2)$. Otherwise, a view change mechanism takes place, to change the leader and continue the computation. Such mechanism implies that when a leader is suspected to be faulty, all processes have to collect enough evidences for the view-change, that is, the view-change message contains $2f + 1$ signed messages and those messages are sent from $2f + 1$ processes. Those messages are then sent to all processes, so that the view-change costs $O(n^3)$ in terms of message complexity. Since the protocol terminates when there is a correct leader, then in the worst case scenario it has a message complexity of $O(n^4)$. The view-change is used to avoid that in case of faulty leader, if some correct process decides on a value v, the other correct processes cannot decide on a value $v' \neq v$ when the new leader takes place. Tendermint reduces the message complexity in case of worst case scenario to $O(n^3)$ thanks to the $lockedValue$. That is, processes instead of exchanging all the messages they already delivered, they locally keep track of potentially decided values to preserve the safety, in this way reducing the message complexity. In the same spirit, HotStuff [2] incurs in the same message complexity, sharing with Tendermint a linear proposer replacement.

In the blockchain context, there exist PBFT based blockchain propositions (e.g., [3][19][14] based on PBFT) and real implementations as Hyperledger [5] based on BFT-SMaRt [24]. Moreover we have also examples of Blockchain solution that are based on new BFT Consensus algorithms as RedBelly based on DBFT Consensus algorithm [13] that presents a message complexity of $O(n^3)$.

5 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 considered. We believe that this methodology can contribute in making consensus algorithms more understandable for developers and practitioners.

References

1. Livelock scenario. `https://github.com/tendermint/tendermint/wiki/0.7-Livelock-Scenario`, accessed: 2019-03-14
2. Abraham, I., Gueta, G., Malkhi, D.: Hot-stuff the linear, optimal-resilience, one-message BFT devil. CoRR **abs/1803.05069** (2018), `http://arxiv.org/abs/1803.05069`
3. Abraham, I., Malkhi, D., Nayak, K., Ren, L., Spiegelman, A.: Solidus: An incentive-compatible cryptocurrency based on permissionless byzantine consensus. CoRR, abs/1612.02916 (2016)
4. Aguilera, M.K.: A pleasant stroll through the land of infinitely many creatures. ACM Sigact News **35**(2), 36–59 (2004)
5. at al., E.A.: Hyperledger fabric: a distributed operating system for permissioned blockchains. In: Proceedings of the Thirteenth EuroSys Conference, EuroSys 2018, Porto, Portugal, April 23-26, 2018. pp. 30:1–30:15 (2018)
6. Amoussou-Guenou, Y., Pozzo, A.D., Potop-Butucaru, M., Tucci Piergiovanni, S.: Correctness and fairness of tendermint-core blockchains. CoRR **abs/1805.08429** (2018)
7. Anceaume, E., Pozzo, A.D., Ludinard, R., Potop-Butucaru, M., Tucci Piergiovanni, S.: Blockchain abstract data type. CoRR **abs/1802.09877** (2018)
8. Baldoni, R., Bertier, M., Raynal, M., Tucci-Piergiovanni, S.: Looking for a definition of dynamic distributed systems. In: International Conference on Parallel Computing Technologies. pp. 1–14. Springer (2007)
9. Ben-Or, M.: Another advantage of free choice (extended abstract): Completely asynchronous agreement protocols. In: Proceedings of the second annual ACM symposium on Principles of distributed computing. pp. 27–30. ACM (1983)
10. Buchman, E., Kwon, J., Milosevic, Z.: The latest gossip on bft consensus. arXiv preprint arXiv:1807.04938 (2018)
11. Castro, M., Liskov, B.: Practical Byzantine Fault Tolerance. In: Proceedings of the Symposium on Operating Systems Design and Implementation (OSDI) (1999)
12. Crain, T., Gramoli, V., Larrea, M., Raynal, M.: (Leader/Randomization/Signature)-free Byzantine Consensus for Consortium Blockchains. `http://csrg.redbellyblockchain.io/doc/ConsensusRedBellyBlockchain.pdf` (visited on 2018-05-22) (2017)
13. Crain, T., Gramoli, V., Larrea, M., Raynal, M.: Dbft: Efficient byzantine consensus with a weak coordinator and its application to consortium blockchains. arXiv preprint arXiv:1702.03068 (2017)
14. Decker, C., Seidel, J., Wattenhofer, R.: Bitcoin Meets Strong Consistency. In: Proceedings of the 17th International Conference on Distributed Computing and Networking Conference (ICDCN) (2016)

15. Dwork, C., Lynch, N.A., Stockmeyer, L.J.: Consensus in the presence of partial synchrony. J. ACM **35**(2), 288–323 (1988)
16. Dwork, C., Naor, M.: 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. pp. 139–147 (1992)
17. Fischer, M.J., Lynch, N.A.: A lower bound for the time to assure interactive consistency. Information processing letters **14**(4), 183–186 (1982)
18. Fischer, M.J., Lynch, N.A., Paterson, M.S.: Impossibility of distributed consensus with one faulty process. Journal of the ACM (JACM) **32**(2), 374–382 (1985)
19. Kokoris-Kogias, E., Jovanovic, P., Gailly, N., Khoffi, I., Gasser, L., Ford, B.: Enhancing Bitcoin Security and Performance with Strong Consistency via Collective Signing. In: Proceedings of the 25th USENIX Security Symposium (2016)
20. Kwon, J., Buchman, E.: Cosmos: A Network of Distributed Ledgers. `https://cosmos.network/resources/whitepaper` (visited on 2018-05-22)
21. Kwon, J., Buchman, E.: Tendermint. `https://tendermint.readthedocs.io/en/master/specification.html` (visited on 2018-05-22)
22. Lamport, L., Shostak, R., Pease, M.: The byzantine generals problem. ACM Transactions on Programming Languages and Systems **4**(3), 382–401 (Jul 1982)
23. Pease, M., Shostak, R., Lamport, L.: Reaching agreement in the presence of faults. Journal of the ACM **27**(2), 228–234 (Apr 1980)
24. Sousa, J., Alchieri, E., Bessani, A.: State machine replication for the masses with bft-smart (2013)
25. Tendermint: Tendermint: correctness issues. `https://github.com/tendermint/spec/issues` (visited on 2018-09-24)
26. Tendermint: Tendermint: Tendermint Core (BFT Consensus) in Go. `https://github.com/tendermint/tendermint/blob/e88f74bb9bb9edb9c311f256037fcca217b45ab6/consensus/state.go` (visited on 2018-05-22)