Formally Documenting Tenderbake

Sylvain Conchon
Nomadic Labs, Paris, France

Alexandrina Korneva
Université Paris-Saclay, ENS Paris-Saclay, CNRS, LMF, 91190, Gif-sur-Yvette, France

Çagdas Bozman
Functori, Paris, France

Mohamed Iguernlala
Functori, Paris, France

Alain Mebsout
Functori, Paris, France

Abstract

In this paper, we propose a formal documentation of Tenderbake, the new Tezos consensus algorithm, slated to replace the current Emmy family algorithms. The algorithm is broken down to its essentials and represented as an automaton. The automaton models the various aspects of the algorithm: (i) the individual participant, referred to as a baker, (ii) how bakers communicate over the network (the mempool) and (iii) the overall network the bakers operate in. We also present a TLA+ implementation, which has proven to be useful for reasoning about this automaton and refining our documentation. The main goal of this work is to serve as a formal foundation for extracting intricate test scenarios and verifying invariants that Tenderbake's implementation should satisfy.

2012 ACM Subject Classification Software and its engineering → Formal methods

Keywords and phrases Consensus algorithm, Tezos blockchain, TLA+

Digital Object Identifier 10.4230/OASIcs.FMBC.2021.4

Category Short Paper

Supplementary Material *Model (Source Code)*: `https://www.lri.fr/~conchon/tenderbake/Tenderbake.tla`

1 Introduction

Tenderbake is a new consensus algorithm designed by Nomadic Labs for the Tezos blockchain [5]. Tenderbake participates in the blockchain protocol to ensure that all peers reach agreement on the state of the distributed ledger. Essentially, the algorithm ensures that all participants record the same blocks, in the same order, in their local copy of the blockchain.

Like Tezos's current Emmy family protocols, Tenderbake is a Byzantine Fault-Tolerant (BFT) algorithm that can tolerate (a limited number of) malicious machine failures on an aynchronous network. The main advantage of Tenderbake is related to block finality, *i.e.*, the point at which the parties involved can consider the consensus on adding a block to be complete. More precisely, this is the moment when it becomes impossible to go back or modify a block that has been added to the blockchain. Unlike the probabilistic finality of Emmy algorithms, where the probability that a block will eventually belong to the blockchain increases with the number of blocks added in front of it, Tenderbake allows for an almost immediate finality: a block is considered to belong to the chain when only two blocks are added after it. This new consensus algorithm technology is inspired by PBFT (practical Byzantine Fault-Tolerant) protocols [4] like Tendermint [1, 3] in the Cosmos project [6].

To achieve such a finality result, Tenderbake implements a three-phase PBFT protocol: a *proposal* phase where a single participant (called *baker*) proposes a new block, and two successive *voting* phases (called *preendorsement* and *endorsement*) at the end of which a quorum of votes must be reached on the proposed block. If a consensus is reached, each participant adds the proposed block locally to their blockchain and a new instance of the algorithm can then start for the next block (referred to as the *next level* in Tezos). However, this idyllic scenario can fail for many reasons. For example, Byzantine participants can inject fake blocks or fake votes. The consensus can also fail even in the absence of participant failure because blocks and votes, which are sent as messages, can be arbitrarily delayed or lost by the network. In this case, a new round of proposals/votes is launched, possibly with a new block issued by another participant.

Tenderbake implements several mechanisms to avoid Byzantine attacks or asynchrony-related problems to guarantee the correctness of the consensus. For instance, a synchronization mechanism is required for each participant to decide that a round of proposals/votes is over. For this purpose, Tenderbake implements a partially synchronous system, where participants synchronize without exchanging messages, by exploiting their internal clocks and the information stored in the blockchain. As another example, cryptographic certificates about the (pre)endorsing majority are injected into blocks to prevent Byzantine attacks.

Designing and implementing a consensus algorithm like Tenderbake is notoriously challenging. While a very precise proof-and-paper description of this algorithm has been given in [2], we propose in this paper a TLA$^+$ modeling of Tenderbake. To do this, we break down the algorithm to its essentials and represent bakers' roles as an automaton. We also abstract the notion of time, but retain a synchronization mechanism that allows the drift of participants' clocks to be simulated. We do not sacrifice any of the more subtle features of Tenderbake's implementation, like how the protocol is handled by both the mempool (a more sophisticated gossip layer) and the bakers themselves.

The main goal of our work is to provide a formal executable documentation of Tenderbake that will serve as a basis for extracting complex test scenarios and invariants that the Tenderbake implementation must satisfy. So far, our TLA$^+$ automaton has proven useful for reasoning and exchanging with the developers of the actual implementation. The TLA$^+$ model is available at `https://www.lri.fr/~conchon/tenderbake/`.

2 Tezos Architecture

Tezos forms a Peer-to-peer network in which peers, called *nodes*, are interconnected and communicate by message passing. Nodes implement the core algorithms and data structures of the blockchain. They are composed of a Peer-to-peer layer (P2P), validators (which use the rules of the economic protocol to check blocks and operations), a distributed database (DDB), and a specific data structure for pending operations, called the *Mempool*.

Nodes continuously run a gossip protocol to communicate and exchange blockchains (complete or just head blocks) with each other. Each node maintains in the Mempool the best version of the blockchain that it has received. Nodes do not communicate plain messages directly, but only a hash value of them. When a node receives a hash, it checks if this value is already stored in its DDB before saving it. The role of the DDB is to maintain a correspondance between hash keys and the plain values associated with them. For that, as a parallel task, the DDB fetches data (of which only the hash is known) from the node's peers, and, conversely, responds to similar peers' requests by providing them with the requested data. When the DDB gets a response, it transmits to the Mempool the plaintext values that correspond to blocks, transactions, or votes.

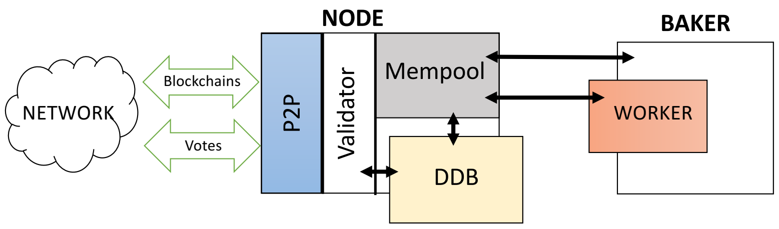

Figure 1 Tezos general architecture.

In this architecture, shown in Figure 1, bakers are not directly visible on the network. For security reasons, they only communicate with each other through the nodes they are connected to (which we refer as *the node of the baker*). The role of a baker is to produce proposal blocks and to vote for the head blocks of the blockchain stored in its node's Mempool. For that, a baker gets the first two blocks of the blockchain from the Mempool (via a Remote Procedure Call mechanism – RPC) and it implements the consensus rounds of Tenderbake to decide whether to vote on the current head or not. A baker is also composed of a worker running in parallel, whose role consists of getting the votes from the Mempool (via RPC) and checking for potential quorums.

This modular, secure and highly parallel architecture raises several issues when implementing a PBFT algorithm like Tenderbake. First, while a Baker is voting on a specific blockchain head, the Mempool can receive a new proposal and decide to change its head. This means that everything needs to be resynchronized for the baker and the worker to vote or get a quorum on the current head. Secondly, Tezos has been designed to be agnostic to the consensus algorithm used to produce blocks. As a consequence, the rules of the Tenderbake algorithm are abstract, so it is important to make sure that the Mempool has access to all necessary information needed to choose the best blockchain. Last, bakers combine timestamp information stored in the blocks and their current clock to know how long before a round timeout is triggered. Since each baker has their own clock, this can lead to clock drift, to which the protocol must be resistant.

Finally, the communication mechanism between components involves RPC (Worker/Mempool and Baker/Mempool) and streams of events (Worker/Baker). To simplify our modeling, we approximate these communications through a shared memory mechanism and leave the modeling of a communication layer closer to the implementation to future work.

3 Tenderbake Automaton

In this section, we describe the Tenderbake consensus formally, for a set of participants BAKERS. Contrary to the implementation in Tezos, where participants change at each level, we assume that this set is fixed. Each individual participant (baker) runs the same automaton. We explain how this automaton is implemented in TLA$^+$ in Section 4.

The automaton is given in Figure 2. It represents the evolution of a baker's state and the actions performed by this baker in the three possible consensus phases. In the rest of this section, we give a description of the local state maintained by an arbitrary baker i and we detail the transitions of this automaton using a rudimentary guarded command language.

Notations. By convention, the internal variables of the baker i are denoted by capital letters associated with an index i. Thus, X_i represents the internal variable X of i. We use lowercase letters for parameters. Certain variables are *option variables*, meaning that

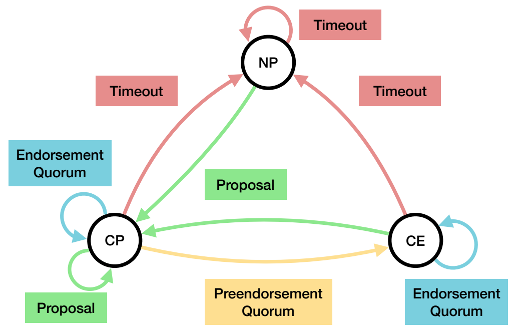

Figure 2 Tenderbake automaton.

```
Trigger Timeout
$\neg \mathsf{TO}_i \wedge$
$(\forall j,k. j \neq k \implies |\mathsf{TICK}_j - \mathsf{TICK}_k| \leq \Delta) \wedge$
$(\forall j. j \neq i \implies |\mathsf{TICK}_i + 1 - \mathsf{TICK}_j| \leq \Delta) \longrightarrow$
    $\mathsf{TO}_i := \mathtt{true};$
    $\mathsf{TICK}_i := \mathsf{TICK}_i + 1;$
```

Figure 4 Trigger timeout oracle.

```
Proposal
$\mathsf{CH}_i = (\mathsf{B}, \_)$
$\mathsf{NodeCH}_i = (\mathsf{H}, \mathsf{PRE})$

$\neg \mathsf{TO}_i \;\wedge\; \mathsf{B} \neq \mathsf{H} \longrightarrow$
    $\mathbf{let}\{\ell; r; p; pqc; eqc\} = \mathsf{H}\ \mathbf{in}$

    $\mathsf{B}.\ell < \ell \longrightarrow$
        $\mathsf{RND}_i := \mathsf{TICK}_i - (\mathsf{PRE}.t + 1)$
        $\mathsf{LOCKED}_i := -; \mathsf{PQC}_i := -; \mathsf{ELECT}_i := -$
        $\mathsf{CH}_i := (\mathsf{H}, \mathsf{PRE})$
        $\mathsf{PQC}_i := pqc$
        $\mathsf{RND}_i = r \longrightarrow \mathsf{PE}_i := \mathsf{PreEndorse}(i, \ell, r, b)$

    $\ell = \mathsf{B}.\ell \;\wedge\; r < \mathsf{RND}_i \wedge \mathsf{B}.r < r \longrightarrow$
        $\mathsf{CH}_i := (\mathsf{H}, \mathsf{PRE})$
        $\mathsf{PQC}_i.r^? < pqc.r^? \longrightarrow \mathsf{PQC}_i := pqc$

    $\ell = \mathsf{B}.\ell \;\wedge\; r = \mathsf{RND}_i \;\wedge\; (r \neq \mathsf{B}.r \;\vee\; p = \mathsf{B}.p) \longrightarrow$
        $\mathsf{CH}_i := (\mathsf{H}, \mathsf{PRE})$
        $\mathsf{PQC}_i.r^? < pqc.r^? \longrightarrow \mathsf{PQC}_i := pqc$
        $\mathsf{LOCKED}_i = - \;\vee\; \mathsf{LOCKED}_i.p = p \;\vee$
        $(pqc \neq - \;\wedge\; \mathsf{LOCKED}_i.r \leq pqc.r) \longrightarrow$
            $\mathsf{PE}_i := \mathsf{PreEndorse}(i, \ell, r, p)$
```

Figure 3 Receiving a proposal.

```
Not Proposer
$\mathsf{CH}_i = (\mathsf{B}, \_)$

$\mathsf{TO}_i \;\wedge\; \neg\, isProposerNextRound(i) \wedge$
        $\neg\, isProposerNextLevel(i) \longrightarrow$
    $\mathsf{TO}_i := \mathtt{false}$
    $\mathsf{RND}_i := \mathsf{RND}_i + 1$
```

```
Proposer Next Round
$\mathsf{CH}_i = (\mathsf{B}, \mathsf{P})$

$\mathsf{TO}_i \;\wedge\; isProposerNextRound(i) \wedge$
        $\neg\, isProposerNextLevel(i) \longrightarrow$
    $\mathsf{TO}_i := \mathtt{false}$
    $\mathsf{RND}_i := \mathsf{RND}_i + 1$
    $\mathbf{let}\ p, pqc, P =$
        $\mathbf{if}\ (\mathsf{PQC}_i = -)\ \mathbf{then}\ (\varepsilon, -, \mathsf{P})$
        $\mathbf{else}\ (\mathsf{PQC}_i.p, \mathsf{PQC}_i, \mathsf{PQC}_i.pred)\ \mathbf{in}$
    $\mathbf{let}\ \mathsf{b} = \{\mathsf{B}\ \mathbf{with}\ t = \mathsf{TICK}_i; r = \mathsf{RND}_i; p; pqc\}\ \mathbf{in}$
    $\mathsf{P}_i := \mathsf{Propose}(i, (\mathsf{b}, P))$
```

```
Proposer Next Level
$\mathsf{CH}_i = (\mathsf{B}, \_)$

$\mathsf{TO}_i \;\wedge\; isProposerNextLevel(i) \longrightarrow$
    $\mathsf{TO}_i := \mathtt{false}$
    $\mathbf{let}\ \mathsf{b} = \{\ell = \mathsf{B}.\ell + 1;\ t = \mathsf{TICK}_i;\ p = \varepsilon;$
            $r = \mathsf{RND}_i - \mathsf{ELECT}_i.b.r;$
            $eqc = \mathsf{ELECT}_i.q;\ pqc = -\}\ \mathbf{in}$
    $\mathsf{RND}_i := \mathsf{RND}_i + 1$
    $\mathsf{P}_i := \mathsf{Propose}(i, (\mathsf{b}, \mathsf{ELECT}_i.b))$
```

Figure 5 A baker's possible actions once `timeout` has been reset.

they can have a value or not. Not having a value is denoted by the symbol `-`. When comparing variables, $X^?$ means that X is an option variable and can therefore be empty. By convention, empty variables are (strictly) less than non-empty variables. We stick to conventional message passing notation where $m(x_1, \ldots, x_k)?$ stands for the reception of a message m with parameters $x_1, \ldots, x_k$, and $m(v_1, \ldots, v_k)!$ is the asynchronous broadcast of m with $v_1, \ldots, v_k$ as arguments. Note that when a baker broadcasts a message, he does not send it to himself.

Baker's state. As shown in Figure 2, our automaton has three distinct states, which correspond to the possible phases of the consensus algorithm: NP for *Non Proposer*, CP for *Collecting Preendorsements*, and CE for *Collecting Endorsements*. In addition to this control flow information, a baker i maintains a copy of the blockchain in a variable CH_i. Since only the two head blocks of the blockchain are needed for the consensus algorithm, CH_i contains a pair of blocks (B, P), where B is the head block of the blockchain and P its predecessor. A block is represented by a record $\{\ \ell;\ r;\ t;\ p;\ eqc;\ pqc\ \}$, where each component is accessible via the standard record access notation (*e.g.* $\mathsf{B}.r$). The role of each of these components is summarized in Figure 6.

In addition to the two head blocks stored in CH_i, a baker maintains his current consensus round in RND_i. For safety reasons, a baker must also keep track of the block he voted for, in variable LOCKED_i and for which a preendorsement voting quorum was observed.

ℓ	level of the block in the blockchain;
r	consensus round during which the block was proposed;
t	timestamp of when the block was proposed;
p	block's payload - *i.e.* contents without consensus operations;
pqc	preendorsing majority certificate with the round when it was observed;
eqc	endorsing majority certificate for the previous block.

Figure 6 Block structure.

Initial state for Baker i			**Init. state for Mempool**		
CH_i	$=$	(G, G)	NodeCH_i	$=$	(G, G)
RND_i	$=$	0	M_i	$=$	$\emptyset$
TICK_i	$=$	1	P_i	$=$	$-$
LOCKED_i	$=$	$Genesis$	E_i	$=$	$-$
PQC_i	$=$	$\{p = []; q = \emptyset; r = 0\}$	PE_i	$=$	$-$
ELECT_i	$=$	$\{b = G; q = \emptyset\}$			
TO_i	$=$	$\mathtt{true}$			

where $G = \{\ell = 0; r = 0; t = 0; p = []; pqc = -; eqc = \emptyset\}$

Figure 7 Initial states.

```
NodeCH_i = (H, PRE)                                         Mempool

PreEndorse(j, ℓ, r, p)? ∨ PE_i = PreEndorse(j, ℓ, r, p) ⟶
    M_i := M_i ∪ {PreEndorse(j, ℓ, r, p)}
    PE_i ≠ − ⟶
        PreEndorse(i, ℓ, r, p)!
        PE_i := −

Endorse(j, ℓ, r, p)? ∨ E_i = Endorse(j, ℓ, r, p) ⟶
    M_i := M_i ∪ {Endorse(j, ℓ, r, p)}
    E_i ≠ − ⟶
        Endorse(i, ℓ, r, p)!
        E_i := −

Propose(j, (h, pre))? ∨ P_i = Propose(j, (h, pre)) ⟶
  let {ℓ; r; pqc; eqc} = h in
  isProposer(j, ℓ, r) ∧
  (H.ℓ, H.pqc.r, −PRE.r, H.r) < (ℓ, pqc.r, −pre.r, r) ∧
  valid_eqc(eqc, pre) ∧
  (pqc = − ∨ valid_pqc(h)) ⟶
    NodeCHi := (h, pre)
    P_i ≠ − ⟶
      Propose(i, (h, pre))!
      P_i := −
```

Figure 9 Mempool transitions.

```
CH_i = (B, _)                                   Preendorsement Quorum

let q = {m ∈ M_i | m = PreEndorse(_, ℓ, r, p) ∧
                   ℓ = B.ℓ ∧ p = B.p ∧
                   r = RND_i = B.r} in
¬TO_i ∧ quorum(q) ∧ LOCKED_i ≠ B ⟶
    PQC_i := {p = B.p; r = B.r; q = q}
    LOCKED_i := B
    E_i := Endorse(i, B.ℓ, B.r, B.p)
```

```
CH_i = (B, _)                                   Endorsement Quorum

let q = {m ∈ M_i | m = Endorse(_, ℓ, r, p) ∧
                   ℓ = B.ℓ ∧ p = B.p ∧
                   r = RND_i = B.r} in
¬TO_i ∧ quorum(q) ∧ ELECT_i = − ⟶
    ELECT_i := {b = B; q = q}
```

Figure 8 Preendorsement and endorsement quorums.

$quorum(x) \triangleq |x| > \frac{2\times|\mathsf{BAKERS}|}{3}$

$valid\_eqc(eqc, pre) \triangleq quorum(eqc) \wedge$
$\forall \mathsf{Endorse}(i, \ell, r, p) \in eqc,\ (\ell, r, p) = pre.(\ell, r, p)$

$valid\_pqc(b) \triangleq quorum(b.pqc.q) \wedge$
$\forall \mathsf{PreEndorse}(i, \ell, r, p) \in b.pqc.q,\ (\ell, p) = b.(\ell, p) \wedge r < b.r$

$isProposer(i, \ell, r) \triangleq ((\ell + r) \bmod |\mathsf{BAKERS}|) + 1 = i$

$isProposerNextRound(i) \triangleq \mathbf{let}\ (\mathsf{B}, \mathsf{P}) = \mathsf{CH}_i\ \mathbf{in}$
$isProposer(i, \mathsf{B}.\ell, \mathsf{RND}_i + \mathsf{P}.r - \mathsf{PQC}_i.pred.r + 1)$

$isProposerNextLevel(i) \triangleq \mathsf{ELECT}_i \neq - \wedge$
$isProposer(i, \mathsf{B}.\ell + 1, \mathsf{RND}_i - \mathsf{ELECT}_i.b.r)$

Figure 10 Definitions of predicates.

To guarantee progression, a record ELECT_i of the form $\{\ b;\ q;\ \}$ is used to store the first observed endorsement quorum (in q) for the head block (in b). Finally, in order to speed up the convergence of the algorithm, a record PQC_i of the form $\{\ p;\ r;\ q;\ \}$ is used to keep track of the preendorsement quorum q with the highest round r, associated to the block payload p. The initial state for a baker is given in Figure 7. Bakers are locked on and have elected the genesis block G in order to force the progression to go through proposals at level 1.

Time and clocks. Tenderbake runs on the notion of rounds and time. As mentioned in Section 1, the ideal consensus scenario is not always attainable. This is where the concept of rounds comes in. Bakers have a predefined number of seconds to decide on a block. Once that time is up, and if an agreement has not been reached, a timeout event is triggered, and the bakers have to drop what they were doing and start a new round. In Tenderbake, this is achieved with clocks and real-time. By combining timestamp information stored in the blocks and their current clock, bakers can calculate both their current round in the consensus and the time remaining before a timeout is triggered. The protocol is also resistant (to some extent) to a possible clock drift between bakers.

Our model accounts for this clock/real-time mechanism in an abstract way. To do this, we first simplify the problem by considering that all rounds have the same duration. Then, we get rid of local clocks by replacing them with local counters that contain the number of timeouts a baker has received. Finally, we use a global mechanism (the *oracle*, depicted in Figure 4) to notify a baker when a round ends. Although it may seem too simplistic, our mechanism allows us to account for the problems related to time in Tenderbake, in particular the one related to clock drift.

To implement our abstract synchronization mechanism, we assign two local variables to each baker: a boolean TO_i, for *timeout*, used by the oracle to communicate the end of a round to the baker, and an integer TICK_i to count the number of rounds elapsed since the blockchain was started. We also use a constant Δ to set the maximum offset on the number of ticks (*i.e.* rounds) between bakers.

To start a new round for a baker i, our oracle executes *non-deterministically* the guard/action command in Figure 4 as soon as (1) the baker i has no timeout to handle (2) the differences between any two bakers' counted rounds does not exceed Δ, before and after execution of the transition.

The command's action sets the *timeout* variable of the baker i to `true` and increments its tick counter. This transition guarantees that no two bakers can drift for more than Δ rounds but allows each one to proceed independently. After this transition, the baker must handle its timeout and move according to one of the three cases described in the next paragraph. In Tenderbake, we use $\Delta = 1$, which means that internal clocks of the machines on which bakers run are only allowed to drift by an amount that would result in a difference of at most one round.

Timeout transitions. As shown in Figure 2, a baker is forced to move to state NP when the oracle resets his TO_i variable. This is when the baker can start a new round if no consensus was reached during the current round, or a new level, if the baker has collected a quorum of endorsements for his current head block. The actions bakers are allowed to perform on timeouts depend on their right to propose a new block for the next round (in the same level), or for the earliest possible round of the next level in which the baker can propose. We abstract this authorization with a predicate $\mathsf{IsProposer}(i, \ell, r)$ which is true when baker i is the proposer at level ℓ and round r.

Figure 5 contains the possible behaviors (or transitions) of a baker after a timeout. In Not the Proposer, the baker first checks that he *is not* the proposer for the next round $\mathsf{RND}_i + 1$ of the current level $\mathsf{B}.\ell$ (see Def. of *isProposerNextRound*). Then, either there is no block stored in ELECT_i (denoted by $\mathsf{ELECT}_i = \text{-}$), meaning the baker did not obtain a quorum for his head block, or the baker is not the proposer for the next level (see Def. of *isProposerNextLevel*). In the latter case, instead of $\mathsf{IsProposer}(i, \mathsf{B}.\ell + 1, 0)$, the baker checks for the round $\mathsf{RND}_i - \mathsf{ELECT}_i.b.r$ of the next level $\mathsf{B}.\ell + 1$. This expression takes into account the difference between the baker's current round RND_i and the round during which the baker obtained a quorum for his head block (stored in the ELECT_i variable). Thus, for instance, if a baker obtains a quorum at round $\mathsf{RND}_i = r$, and if he is the proposer for the next level at the end of that round r, then the baker checks indeed the first round $\mathsf{RND}_i - r = 0$ of the next level. The actions associated to this transition consist only of resetting the TO_i variable and incrementing the counters TICK_i and RND_i. In Proposer of next round, the baker communicates a proposal $\mathsf{Propose}(i, (b, P))$ for the next round to the Mempool through the variable P_i. The block b is built using the content of the head block B with new timestamp and round information. The payload of this new proposal is either a fresh value (denoted by ε) or the payload of the block stored in the baker's PQC_i

variable, if it exists. The preendorsement quorum certificate of this new block is either empty or the one stored in PQC_i. In PROPOSER OF NEXT LEVEL, the baker must have a block stored in ELECT_i and he must also be the proposer of the round $\mathsf{RND}_i - \mathsf{ELECT}_i.b.r$ in the next level $\mathsf{B}.\ell + 1$. The new proposal contains a fresh payload, an endorsement quorum for its block predecessor taken from $\mathsf{ELECT}_i.q$ and a timestamp equal to TICK_i.

The Mempool. While a Mempool typically serves as a gossip layer, simply passing on messages between bakers, Tenderbake's Mempool is more sophisticated. For instance, the Mempool keeps a local variable NodeCH_i, its own copy of the blockchain, the most up-to-date version that it has "seen" come through. Since the consensus in Tenderbake depends on the last two blocks, NodeCH_i contains only the head of the blockchain and its predecessor in our model. In addition to these blocks, the Mempool also maintains a set M_i of all of the votes ($\mathsf{PreEndorse}$ or $\mathsf{Endorse}$ messages) that it receives from all bakers.

Furthermore, when the Mempool receives a proposal, either through a message or a shared variable, it first verifies that the proposed block is actually *better* than its current head. If it is indeed better, the Mempool simply updates its version of the blockchain. Otherwise, it is ignored. The notion of a *better chain* is an important part of a consensus algorithm, corresponding to a total ordering between blocks. In Tezos, this ordering is based on a notion of *fitness*, which amounts to comparing, in a lexicographic order, the following quadruples $(\mathsf{H}.\ell, \mathsf{H}.pqc.r, -\mathsf{PRE}.r, \mathsf{H}.r) < (l, pqc.r, -pre.r, r)$, where H and PRE are the first two head blocks of NodeCH_i, while h and pre are the blocks received in a $\mathsf{Propose}(j, (h, pre))$ message. Moreover, in addition to fitness, the Mempool ensures the information contained in the *eqc* and *pqc* fields is valid. Last, if this better proposal has been received through a shared variable, the Mempool broadcasts it to the other participants. Figure 9 shows transitions of the Mempool that handle $\mathsf{PreEndorse}$ and $\mathsf{Endorse}$ votes (received either by messages or through the shared variables PE_i and E_i). These messages are simply stored in M_i[1].

Proposal transition. As seen in Figure 2, a baker can handle a new proposal in any state. We give in Figure 3 the $\mathsf{Proposal}$ transition that a baker can execute as soon as he is running a new round and when the head block B in CH_i is different from the one in the Mempool. In that case, a baker determines if he can vote (preendorse) for the new head stored in the Mempool. There are only two possibilities for a baker to preendorse a proposal:

1. The chain stored in the Mempool is *strictly* longer than the one stored in the baker.
2. Both chains have the same length and the proposal's round is equal to the current baker's round RND_i. The baker also checks that he is not about to vote twice in the same round, except for the same payload. Moreover, the baker only preendorses in this case if:
   a. he has never endorsed (locked) a previous proposal in the same level, or
   b. he is locked to some block payload p0 at some round r0, but the current proposal's payload is equal to p0, or the current proposal got a PQC at some round r1 > r0.

In (1), a baker synchronizes the value of its current round RND_i in the new level. It also checks, before preendorsing, that the block H, while at a higher level, does not correspond to an old proposal.

Quorums. The last two transitions are described in Figure 8. As mentioned above, the Mempool keeps a set M_i of all the messages it has received. If the number of preendorse messages for the head block B stored in CH_i is enough for a quorum, then a baker can

[1] Although we could wipe the contents of M_i at each new round startup, we decided not to do it explicitly to be able to explore different mempool cleaning strategies in practice.

execute the Preendorsement Quorum transition to update PQC_i with his current head and the calculated quorum, change LOCKED_i to B, since this is the block he is about to endorse, and communicate an $\mathsf{Endorse}(i, \mathsf{B}.\ell, \mathsf{B}.r, \mathsf{B}.p)$ message to the Mempool. An endorsement quorum transition is possible in states CE and CP. The baker observes endorsement quorums only when his ELECT_i variable is not set. In that case, if enough endorsement messages exist in the Mempool for his head block, the baker records that block and its quorum in ELECT_i.

4 TLA$^+$

In this section we discuss how we go from the previous automaton to its TLA$^+$ implementation. The automaton makes it fairly straightforward to convert to TLA$^+$ by simply representing the baker, the Mempool, the possible actions, and the synchronization mechanism.

The Baker and the Mempool. We define a constant set BAKERS of all bakers in the network. A variable BakerState maps each baker to their state (i.e. the internal variables from Section 3), represented as a record structure. We stray from the types in Section 3 by using n-tuples instead of records to represent LOCKED_i, ELECT_i, and PQC_i. BakerState[i] represents the state of baker i. To model the phases of the algorithm, we add an internal variable STATE_i for each baker. Initially, each baker starts off in the following state, where sequences are delimited by $\langle\ \rangle$, and *Genesis* is the genesis block:

$$\begin{aligned} \mathit{InitialState} \triangleq\ & [\mathit{state} \mapsto \text{"np"}, \mathit{pqc} \mapsto \langle\rangle, \mathit{ch} \mapsto \langle \mathit{Genesis}, \mathit{Genesis}\rangle, \mathit{rnd} \mapsto 0, \\ & \mathit{locked} \mapsto \langle\rangle, \mathit{elect} \mapsto \langle \mathit{Genesis}, \{\}\rangle, \mathit{timeout} \mapsto \text{TRUE}, \mathit{tick} \mapsto 0] \end{aligned}$$

The Mempool is a record with the fields - nodeCH, for its local blockchain (the first two blocks), msgs, the set of Endorse and PreEndorse messages it has received, and the fields propose, endorse, preendorse for the variables P_i, E_i, PE_i. It starts off with an empty set of msgs and two *Genesis* blocks.

Synchronization. As mentioned in Section 3, we introduce an oracle transition which allows bakers to progress individually with timeouts (TO_i) while maintaining synchronization, *i.e.* by being at most Δ rounds apart. We do the same thing in our TLA$^+$ implementation: TO_i is the first enabling condition of each timeout step definition.

Actions. Bakers and the Mempool are impacted by the various actions on the network. Each of these are defined individually in TLA$^+$. For example, the Endorsement Quorum step in Figure. 2, enabled in CP or CE, is defined as follows:

$$\begin{aligned} \mathit{EndQuorum}(i) \triangleq\ & \land\ \mathit{BakerState}[i].\mathit{timeout} = \text{FALSE} \\ & \land\ \mathit{BakerState}[i].\mathit{elect} = \langle\rangle \\ & \land\ \mathit{BakerState}[i].\mathit{state} = \text{"cp"} \lor \mathit{BakerState}[i].\mathit{state} = \text{"ce"} \\ & \land\ \mathit{CollectEnd}(i) \\ & \land\ \mathit{BakerState}' = [\mathit{BakerState}\ \text{EXCEPT} \\ & \qquad\qquad !{[i]}.\mathit{elect} = \langle \mathit{BakerState}[i].\mathit{chain}[1].\mathit{round}, \\ & \qquad\qquad\qquad\qquad \mathit{BakerState}[i].\mathit{chain}[1].\mathit{contents}, \\ & \qquad\qquad\qquad\qquad \mathit{BakerState}[i].\mathit{chain}[1].\mathit{time}\rangle, \\ & \qquad\qquad !{[i]}.\mathit{state} = \ \mathit{BakerState}[i].\mathit{state}] \\ & \land\ \text{UNCHANGED}\ \mathit{Mempool} \end{aligned}$$

Baker i can execute this step iff (i) he is synchronized, (ii) he is in state cp or ce, and (iii) *CollectEnd(i)* is true. *CollectEnd* (for "collecting endorsements") counts all of the Endorse messages for i's current head in Mempool.*msgs* and checks whether it is enough for a quorum. If these three conditions are satisfied, baker i modifies ELECT_i and transitions to phase NP of the algorithm. Every other transition in Figure 2 is defined in a similar way.

Test scenarios. While the automaton made writing our TLA$^+$ specification easier, the spec itself has, in return, proven extremely useful in debugging the automaton. Sometimes a deadlock would be reached when it should not have been, leading us to review Tenderbake's code, and fixing things we overlooked in our model. The main advantage is, however, being able to run various test scenarios. We can easily modify our spec to account for various clock drifts or Byzantine bakers.

5 Conclusion

In this paper we proposed a TLA$^+$ model of Tenderbake, along with an automaton detailing the key parts of Tenderbake. This method simplifies the problem by abstracting the notion of time, while retaining Tenderbake's more nuanced features, such as its more elaborate Mempool. Our method gives us a formalized and executable Tenderbake documentation. This serves as the foundation for running specific test scenarios and verifying properties Tenderbake needs to satisfy. An immediate line of future work is to define those properties and check them with the TLC model checker.

References

1 Yackolley Amoussou-Guenou, Antonella Del Pozzo, Maria Potop-Butucaru, and Sara Tucci Piergiovanni. Correctness of tendermint-core blockchains. In *22nd International Conference on Principles of Distributed Systems, OPODIS 2018, December 17–19, 2018, Hong Kong, China*, volume 125 of *LIPIcs*, pages 16:1–16:16, 2018.

2 Lăcrămioara Astefănoaei, Pierre Chambart, Antonella Del Pozzo, Thibault Rieutord, Sara Tucci Piergiovanni, and Eugen Zalinescu. Tenderbake – A solution to dynamic repeated consensus for blockchains. In *Fourth International Symposium on Foundations and Applications of Blockchain*, 2021.

3 Sean Braithwaite, Ethan Buchman, Igor Konnov, Zarko Milosevic, Ilina Stoilkovska, Josef Widder, and Anca Zamfir. Formal specification and model checking of the tendermint blockchain synchronization protocol (short paper). In *2nd Workshop on Formal Methods for Blockchains, FMBC@CAV 2020, July 20–21, 2020, Los Angeles, California, USA (Virtual Conference)*, volume 84 of *OASIcs*, pages 10:1–10:8, 2020.

4 Miguel Castro, Barbara Liskov, et al. Practical byzantine fault tolerance. In *OSDI*, volume 99, pages 173–186, 1999.

5 LM Goodman. Tezos – A self-amending crypto-ledger white paper. *URL: https://www. tezos. com/static/papers/white paper. pdf*, 2014.

6 Jae Kwon and Ethan Buchman. Cosmos whitepaper, 2019. URL: `https://cosmos.network/resources/whitepaper`.