



HAL
open science

Formally Documenting Tenderbake

Sylvain Conchon, Alexandrina Korneva, Çagdas Bozman, Mohamed Iguernlala, Alain Mebsout

► **To cite this version:**

Sylvain Conchon, Alexandrina Korneva, Çagdas Bozman, Mohamed Iguernlala, Alain Mebsout. Formally Documenting Tenderbake. Open Access Series in Informatics, In press, 10.4230/OA-SIcs.FMBC.2021.5 . hal-03398884

HAL Id: hal-03398884

<https://hal.science/hal-03398884>

Submitted on 23 Oct 2021

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

1 Formally Documenting Tenderbake

2 **Sylvain Conchon**

3 Nomadic Labs, Paris, France

4 **Alexandrina Korneva**

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

6 **Çagdas Bozman**

7 Functori, Paris, France

8 **Mohamed Iguernlala**

9 Functori, Paris, France

10 **Alain Mebsout**

11 Functori, Paris, France

12 — Abstract —

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

21 **2012 ACM Subject Classification** Software and its engineering → Software organization and
22 properties

23 **Keywords and phrases** Consensus algorithm, Tezos blockchain, TLA+

24 **Digital Object Identifier** 10.4230/OASICS.FMBC.2021.5

25 **Category** Short Paper

26 **1** Introduction

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

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

42 To achieve such a finality result, Tenderbake implements a three-phase PBFT protocol:
43 a *proposal* phase where a single participant (called *baker*) proposes a new block, and two



© Sylvain Conchon and Alexandrina Korneva and Çagdas Bozman and Mohamed Iguernlala and Alain Mebsout;

licensed under Creative Commons License CC-BY 4.0

3rd International Workshop on Formal Methods for Blockchains (FMBC 2021).

Editors: Bruno Bernardo and Diego Marmosler; Article No. 5; pp. 5:1–5:9



Open Access Series in Informatics

Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

44 successive *voting* phases (called *preendorsement* and *endorsement*) at the end of which a
 45 quorum of votes must be reached on the proposed block. If a consensus is reached, each
 46 participant adds the proposed block locally to their blockchain and a new instance of the
 47 algorithm can then start for the next block (referred to as the *next level* in Tezos). However,
 48 this idyllic scenario can fail for many reasons. For example, Byzantine participants can inject
 49 fake blocks or fake votes. The consensus can also fail even in the absence of participant
 50 failure because blocks and votes, which are sent as messages, can be arbitrarily delayed or
 51 lost by the network. In this case, a new round of proposals/votes is launched, possibly with
 52 a new block issued by another participant.

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

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

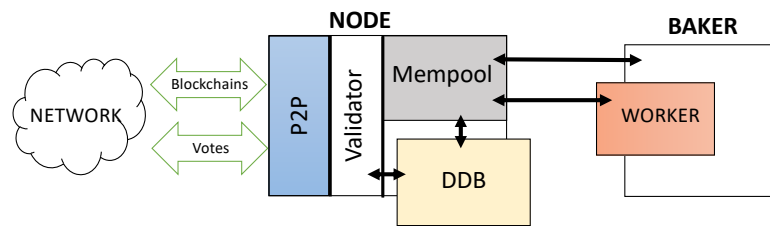
68 The main goal of our work is to provide a formal executable documentation of Tenderbake
 69 that will serve as a basis for extracting complex test scenarios and invariants that the
 70 Tenderbake implementation must satisfy. So far, our TLA⁺ automaton has proven useful for
 71 reasoning and exchanging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

73 **2** Tezos Architecture

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

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

89 In this architecture, shown in Figure 1, bakers are not directly visible on the network.



■ **Figure 1** Tezos general architecture

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

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

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

113 **3 Tenderbake Automaton**

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

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

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

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

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

141 In addition to the two head blocks stored in CH_i , a baker maintains his current consensus
 142 round in RND_i . For safety reasons, a baker must also keep track of the block he voted
 143 for, in variable LOCKED_i and for which a preendorsement voting quorum was observed.
 144 To guarantee progression, a record ELECT_i of the form $\{ b; q; \}$ is used to store the first
 145 observed endorsement quorum (in q) for the head block (in b). Finally, in order to speed up
 146 the convergence of the algorithm, a record PQC_i of the form $\{ p; r; q; \}$ is used to keep track
 147 of the preendorsement quorum q with the highest round r , associated to the block payload p .
 148 The initial state for a baker is given in figure 7. Bakers are locked on and have elected the
 149 genesis block G in order to force the progression to go through proposals at level 1.

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

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

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

171 To start a new round for a baker i , our oracle executes *non-deterministically* the guard/ac-

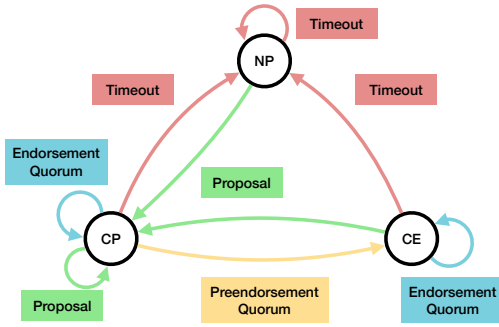


Figure 2 Tenderbake automaton

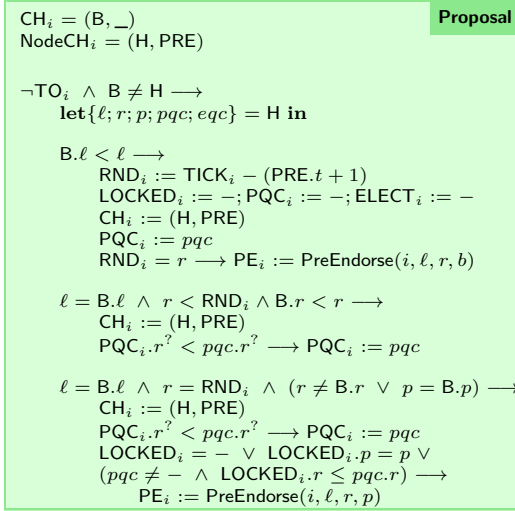


Figure 3 Receiving a proposal

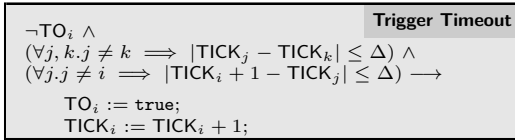


Figure 4 Trigger timeout oracle

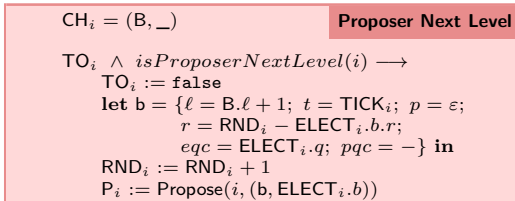
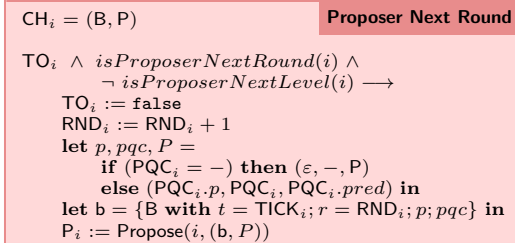
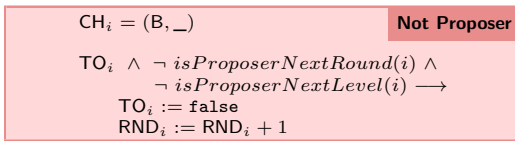


Figure 5 A baker's possible actions once timeout has been reset

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
$CH_i = (G, G)$	$\text{NodeCH}_i = (G, G)$
$\text{RND}_i = 0$	$M_i = \emptyset$
$\text{TICK}_i = 1$	$P_i = -$
$\text{LOCKED}_i = \text{Genesis}$	$E_i = -$
$\text{PQC}_i = \{p = []; q = \emptyset; r = 0\}$	$\text{PE}_i = -$
$\text{ELECT}_i = \{b = G; q = \emptyset\}$	
$\text{TO}_i = \text{true}$	

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

Figure 7 Initial states

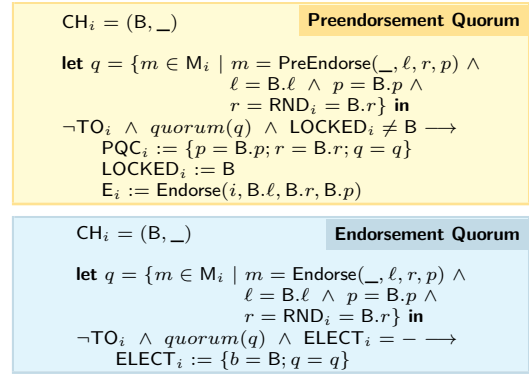


Figure 8 Preendorsement and endorsement quorums

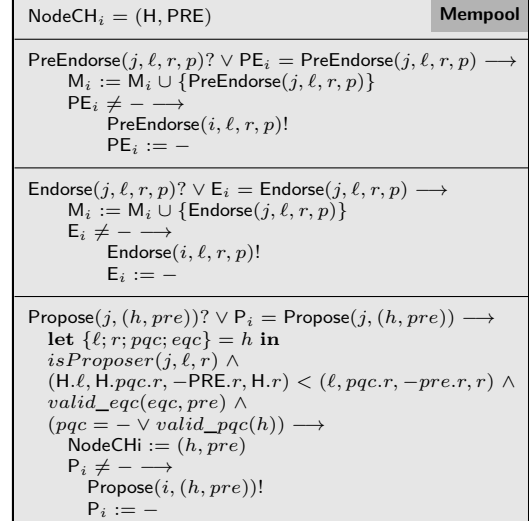


Figure 9 Mempool transitions

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

$$\text{valid_eqc}(eqc, pre) \triangleq \text{quorum}(eqc) \wedge$$

$$\forall \text{Endorse}(i, \ell, r, p) \in eqc, (\ell, r, p) = pre.(\ell, r, p)$$

$$\text{valid_pqc}(b) \triangleq \text{quorum}(b.pqc.q) \wedge$$

$$\forall \text{PreEndorse}(i, \ell, r, p) \in b.pqc.q, (\ell, p) = b.(\ell, p) \wedge r < b.r$$

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

$$\text{isProposerNextRound}(i) \triangleq \text{let } (B, P) = CH_i \text{ in}$$

$$\text{isProposer}(i, B.\ell, \text{RND}_i + P.r - \text{PQC}_i.pred.r + 1)$$

$$\text{isProposerNextLevel}(i) \triangleq \text{ELECT}_i \neq - \wedge$$

$$\text{isProposer}(i, B.\ell + 1, \text{RND}_i - \text{ELECT}_i.b.r)$$

172 tion command in figure 4 as soon as (1) the baker i has no timeout to handle (2) the
 173 differences between any two bakers' counted rounds does not exceed Δ , before and after
 174 execution of the transition.

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

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

190 Figure 5 contains the possible behaviors (or transitions) of a baker after a timeout. In
 191 NOT THE PROPOSER, the baker first checks that he *is not* the proposer for the next round
 192 $RND_i + 1$ of the current level $B.\ell$ (see Def. of *isProposerNextRound*). Then, either there
 193 is no block stored in $ELECT_i$ (denoted by $ELECT_i = -$), meaning the baker did not obtain
 194 a quorum for his head block, or the baker is not the proposer for the next level (see Def.
 195 of *isProposerNextLevel*). In the latter case, instead of $IsProposer(i, B.\ell + 1, 0)$, the baker
 196 checks for the round $RND_i - ELECT_i.b.r$ of the next level $B.\ell + 1$. This expression takes
 197 into account the difference between the baker's current round RND_i and the round during
 198 which the baker obtained a quorum for his head block (stored in the $ELECT_i$ variable). Thus,
 199 for instance, if a baker obtains a quorum at round $RND_i = r$, and if he is the proposer
 200 for the next level at the end of that round r , then the baker checks indeed the first round
 201 $RND_i - r = 0$ of the next level. The actions associated to this transition consist only of
 202 resetting the TO_i variable and incrementing the counters $TICK_i$ and RND_i . In PROPOSER
 203 OF NEXT ROUND, the baker communicates a proposal $Propose(i, (b, P))$ for the next round
 204 to the Mempool through the variable P_i . The block b is built using the content of the head
 205 block B with new timestamp and round information. The payload of this new proposal is
 206 either a fresh value (denoted by ε) or the payload of the block stored in the baker's PQC_i
 207 variable, if it exists. The preendorsement quorum certificate of this new block is either empty
 208 or the one stored in PQC_i . In PROPOSER OF NEXT LEVEL, the baker must have a block
 209 stored in $ELECT_i$ and he must also be the proposer of the round $RND_i - ELECT_i.b.r$ in the
 210 next level $B.\ell + 1$. The new proposal contains a fresh payload, an endorsement quorum for
 211 its block predecessor taken from $ELECT_i.q$ and a timestamp equal to $TICK_i$.

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

219 Furthermore, when the Mempool receives a proposal, either through a message or a shared
 220 variable, it first verifies that the proposed block is actually *better* than its current head. If
 221 it is indeed better, the Mempool simply updates its version of the blockchain. Otherwise,
 222 it is ignored. The notion of a *better chain* is an important part of a consensus algorithm,
 223 corresponding to a total ordering between blocks. In Tezos, this ordering is based on a notion
 224 of *fitness*, which amounts to comparing, in a lexicographic order, the following quadruples
 225 $(H.l, H.pqc.r, -PRE.r, H.r) < (l, pqc.r, -pre.r, r)$, where H and PRE are the first two head
 226 blocks of $NodeCH_i$, while h and pre are the blocks received in a $Propose(j, (h, pre))$ message.
 227 Moreover, in addition to fitness, the Mempool ensures the information contained in the *eqc*
 228 and *pqc* fields is valid. Last, if this better proposal has been received through a shared
 229 variable, the Mempool broadcasts it to the other participants. Figure 9 shows transitions
 230 of the Mempool that handle PreEndorse and Endorse votes (received either by messages or
 231 through the shared variables PE_i and E_i). These messages are simply stored in M_i ¹.

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

- 237 1. The chain stored in the Mempool is *strictly* longer than the one stored in the baker.
- 238 2. Both chains have the same length and the proposal's round is equal to the current baker's
 239 round RND_i . The baker also checks that he is not about to vote twice in the same round,
 240 except for the same payload. Moreover, the baker only preendorses in this case if:
 - 241 a. he has never endorsed (locked) a previous proposal in the same level, or
 - 242 b. he is locked to some block payload p_0 at some round r_0 , but the current proposal's
 243 payload is equal to p_0 , or the current proposal got a PQC at some round $r_1 > r_0$.

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

247 **Quorums.** The last two transitions are described in Figure 8. As mentioned above, the
 248 Mempool keeps a set M_i of all the messages it has received. If the number of preendorse
 249 messages for the head block B stored in CH_i is enough for a quorum, then a baker can
 250 execute the Preendorsement Quorum transition to update PQC_i with his current head and the
 251 calculated quorum, change $LOCKED_i$ to B, since this is the block he is about to endorse, and
 252 communicate an $Endorse(i, B.l, B.r, B.p)$ message to the Mempool. An endorsement quorum
 253 transition is possible in states CE and CP. The baker observes endorsement quorums only
 254 when his $ELECT_i$ variable is not set. In that case, if enough endorsement messages exist in
 255 the Mempool for his head block, the baker records that block and its quorum in $ELECT_i$.

256 4 TLA⁺

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

260 **The Baker and the Mempool.** We define a constant set BAKERS of all bakers in the
 261 network. A variable BakerState maps each baker to their state (i.e. the internal variables

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.

262 from section 3), represented as a record structure. We stray from the types in section 3 by
 263 using n -tuples instead of records to represent LOCKED_i , ELECT_i , and PQC_i . $\text{BakerState}[i]$
 264 represents the state of baker i . To model the phases of the algorithm, we add an internal
 265 variable STATE_i for each baker. Initially, each baker starts off in the following state, where
 266 sequences are delimited by $\langle \rangle$, and Genesis is the genesis block:

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

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

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

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

$$\text{EndQuorum}(i) \triangleq \wedge \text{BakerState}[i].\text{timeout} = \text{FALSE} \\ \wedge \text{BakerState}[i].\text{elect} = \langle \rangle \\ \wedge \text{BakerState}[i].\text{state} = \text{"cp"} \vee \text{BakerState}[i].\text{state} = \text{"ce"} \\ \wedge \text{CollectEnd}(i) \\ \wedge \text{BakerState}' = [\text{BakerState} \text{ EXCEPT} \\ \quad ![i].\text{elect} = \langle \text{BakerState}[i].\text{chain}[1].\text{round}, \\ \quad \quad \quad \text{BakerState}[i].\text{chain}[1].\text{contents}, \\ \quad \quad \quad \text{BakerState}[i].\text{chain}[1].\text{time} \rangle, \\ \quad ![i].\text{state} = \text{BakerState}[i].\text{state}] \\ \wedge \text{UNCHANGED Mempool}$$

281 Baker i can execute this step iff (i) he is synchronized, (ii) he is in state cp or ce, and (iii)
 282 $\text{CollectEnd}(i)$ is true. CollectEnd (for “collecting endorsements”) counts all of the Endorse
 283 messages for i ’s current head in Mempool.msgs and checks whether it is enough for a quorum.
 284 If these three conditions are satisfied, baker i modifies ELECT_i and transitions to phase NP
 285 of the algorithm. Every other transition in Figure 2 is defined in a similar way.

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

293 5 Conclusion

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

301 — **References** —

- 302 **1** Yackolley Amoussou-Guenou, Antonella Del Pozzo, Maria Potop-Butucaru, and Sara Tucci
303 Piergiovanni. Correctness of tendermint-core blockchains. In *22nd International Conference on*
304 *Principles of Distributed Systems, OPODIS 2018, December 17-19, 2018, Hong Kong, China*,
305 volume 125 of *LIPICs*, pages 16:1–16:16, 2018.
- 306 **2** Lăcrămioara Astefănoaei, Pierre Chambart, Antonella Del Pozzo, Thibault Rieutord, Sara
307 Tucci Piergiovanni, and Eugen Zalinescu. Tenderbake - a solution to dynamic repeated
308 consensus for blockchains. In *Fourth International Symposium on Foundations and Applications*
309 *of Blockchain*, 2021.
- 310 **3** Sean Braithwaite, Ethan Buchman, Igor Konnov, Zarko Milosevic, Iliana Stoilkovska, Josef
311 Widder, and Anca Zamfir. Formal specification and model checking of the tendermint
312 blockchain synchronization protocol (short paper). In *2nd Workshop on Formal Methods for*
313 *Blockchains, FMBC@CAV 2020, July 20-21, 2020, Los Angeles, California, USA (Virtual*
314 *Conference)*, volume 84 of *OASICs*, pages 10:1–10:8, 2020.
- 315 **4** Miguel Castro, Barbara Liskov, et al. Practical byzantine fault tolerance. In *OSDI*, volume 99,
316 pages 173–186, 1999.
- 317 **5** LM Goodman. Tezos—a self-amending crypto-ledger white paper. *URL: https://www.tezos.com/static/papers/white_paper.pdf*, 2014.
- 318 **6** Jae Kwon and Ethan Buchman. Cosmos whitepaper, 2019.
- 319