Tenderbake – A Solution to Dynamic Repeated Consensus for Blockchains

Lăcrămioara Aştefănoaei
Nomadic Labs, Paris, France

Pierre Chambart
Nomadic Labs, Paris, France

Antonella Del Pozzo
Université Paris-Saclay, CEA, List, F-91120, Palaiseau, France

Thibault Rieutord
Université Paris-Saclay, CEA, List, F-91120, Palaiseau, France

Sara Tucci-Piergiovanni
Université Paris-Saclay, CEA, List, F-91120, Palaiseau, France

Eugen Zălinescu
Nomadic Labs, Paris, France

Abstract

First-generation blockchains provide probabilistic finality: a block can be revoked, albeit the probability decreases as the block "sinks" deeper into the chain. Recent proposals revisited committee-based BFT consensus to provide deterministic finality: as soon as a block is validated, it is never revoked. A distinguishing characteristic of these second-generation blockchains over classical BFT protocols is that committees change over time as the participation and the blockchain state evolve. In this paper, we push forward in this direction by proposing a formalization of the Dynamic Repeated Consensus problem and by providing generic procedures to solve it in the context of blockchains.

Our approach is modular in that one can plug in different synchronizers and single-shot consensus. To offer a complete solution, we provide a concrete instantiation, called Tenderbake, and present a blockchain synchronizer and a single-shot consensus algorithm, working in a Byzantine and partially synchronous system model with eventually synchronous clocks. In contrast to recent proposals, our methodology is driven by the need to *bound* the message buffers. This is essential in preventing spamming and run-time memory errors. Moreover, Tenderbake processes can synchronize with each other *without* exchanging messages, leveraging instead the information stored in the blockchain.

2012 ACM Subject Classification Theory of computation → Distributed algorithms

Keywords and phrases Blockchain, BFT-Consensus, Dynamic Repeated Consensus

Digital Object Identifier 10.4230/OASIcs.FAB.2021.1

Related Version *Full Version*: `https://arxiv.org/abs/2001.11965`

Acknowledgements We thank Philippe Bidinger for feedback on a previous version of this paper.

1 Introduction

Besides raising public interest, blockchains have also recently gained traction in the scientific community. The underlying technology combines advances in several domains, most notably from distributed computing, cryptography, and economics, in order to provide novel solutions for achieving trust in decentralized and dynamic environments.

Our work has been initially motivated by Tezos [18, 1], a blockchain platform that distinguishes itself through its self-amendment mechanism: protocol changes are proposed and voted upon. This feature makes Tezos especially appealing as a testbed for experimenting

© Lăcrămioara Aştefănoaei, Pierre Chambart, Antonella Del Pozzo, Thibault Rieutord, Sara Tucci-Piergiovanni, and Eugen Zălinescu;
licensed under Creative Commons License CC-BY 4.0

with different consensus algorithms to understand their strengths and suitability in the blockchain context. Tezos relies upon a consensus mechanism build on top of a liquid proof-of-stake system, meaning that block production and voting rights are given to participants in proportion to their stake and that participants can delegate their rights to other stakeholders. As Nakamoto consensus [21, 17], Tezos' current consensus algorithm [24] achieves only probabilistic finality assuming an attacker with at most half of the total stake, and relying on a synchrony assumption.

The initial goal of this work was to strengthen the resilience of Tezos through a BFT consensus protocol to achieve deterministic finality while relaxing the synchrony assumption. We had two general requirements that we found were missing in the existing BFT consensus protocols. First, for security reasons, message buffers need to be bounded: assuming unbounded buffers may lead to memory errors, which can be caused either accidentally or maliciously, through spamming for instance. Second, as previously observed [2], plugging a classical BFT consensus protocol in a blockchain setting with a proof-of-stake boils down to solve a form of repeated consensus [13], where each consensus instance (i) produces a block, i.e., the decided value, and (ii) runs among a committee of processes which are selected based on their stake. To be applicable to open blockchains, committees need to be dynamic and change frequently. Frequent committee changes is fundamental in blockchains for mainly two reasons: (i) it is not desirable to let a committee be responsible for producing blocks for too long, for neither fairness nor security; (ii) participants' stake may change frequently.

Dynamic Repeated Consensus. Typically, repeated consensus is solved with state machine replication (SMR) implementations. We, instead, propose to use a novel formalism, dynamic repeated consensus (DRC) to take into account that, in the context of *open* blockchains, participants in consensus change. To this end, we propose that the selection of participants is based upon information readily available in the blockchains.

To solve DRC, we follow the methodology initially presented in [14] and revived more recently in [29, 22, 23]: we decouple the logic for synchronizing the processes in consensus instances from the consensus logic itself. Thus, our solution uses two main generic ingredients: a synchronizer and a single-shot consensus skeleton. Our approach is modular in that one can plug in different synchronizers and single-shot consensus algorithms. Our solution works in a partially synchronous model where the bound on the message delay is *unknown*, and the communication is *lossy* before the global stabilization time (GST). We note that losing messages is a consequence of processes having bounded memory: if a message is received when the buffers are full, then it is dropped.

Blockchain-based Synchronizer. The need for and the benefits of decoupling the synchronizer from the consensus logic have already been pointed out in [29, 22, 23, 6]. Indeed, such separation of concerns allows reusability and simpler proofs. We continue this line of work and propose a synchronizer for DRC which does not exchange messages. Instead, it relies upon local clocks while leveraging information already stored in the blockchain. Our solution allows buffers to be bounded and guarantees that correct processes in the synchronous period are always in the same round, except for negligeable periods of time due to clock drifts. Thus, processes can discard all the messages not associated with their current or next round. This is similar to the communication-closed round model [10, 16] and in contrast to most existing solutions, which, in principle, need to store messages for an unbounded number of rounds.

Consensus algorithm. To complete our DRC solution, Tenderbake, we also present a single-shot consensus algorithm. Single-shot Tenderbake is inspired by Tendermint [7, 3], in turn inspired by PBFT [8] and DLS [14]. We improve Tendermint in two aspects: i) we remove the reliable broadcast requirement during the asynchronous period, and ii) we provide faster termination. Tendermint terminates once processes synchronize in the same round after GST, in the worst case, in n rounds, where n is the size of the committee. Single-shot Tenderbake terminates in $f+2$ rounds, where f is the upper bound on the number of Byzantine processes. Tenderbake departs from its closest relatives Tendermint and HotStuff [29] in that it is driven by a bounded-buffers design leveraging a synchronizer that paces protocol phases on timeouts only. However, the price for this is that Tenderbake is not optimistic responsive as HotStuff, which makes progress at the speed of the network and terminates in $f+1$ rounds, at the cost of an additional phase. As a last difference, we note that, contrary to recent pipelined algorithms [29, 9], Tenderbake lends itself better to open blockchains. Pipelined algorithms focus more on performance, however pipelining imposes restrictions on how much and how frequently committees can change [9].

We are not aware of any existing approach providing a complete, generic DRC formalization. However, several references exist for particular aspects which we touch upon. For instance, repeated consensus with bounded buffers has been studied in [13, 27] but in system models which assume crash failures only. Working solutions for implementing dynamic committees are (mostly partially) documented in [11, 20, 19, 26, 28, 25, 5]. The differences with respect to the closest relatives of single-shot Tenderbake have been discussed above.

Outline. The paper is organized as follows: Section 2 defines the system model; Section 3 formalizes the DRC problem and proposes a generic solution; Section 4 proposes a synchronizer leveraging blockchain's immutability; Sections 5 - 6 present the single-shot consensus skeleton and respectively single-shot Tenderbake, as an example of an instantiation; Section 7 discusses message complexity and gives some intuition on the upper bound on the recovery time after GST; Section 8 concludes. Appendix B contains the detailed correctness proofs of Tenderbake.

2 System Model

We consider a message-passing distributed system composed of a possibly infinite set Π of processes. Processes have access to digital signing and hashing algorithms. We assume that cryptography is perfect: digital signatures cannot be forged, and there are no hash collisions. Each process has an associated public/private key pair for signing and processes can be identified by their public keys.

Execution model. Processes repeatedly run consensus instances to decide *output* values. New output values are appended to a *chain* that processes maintain locally. Consensus instances run in *phases*. The execution of a phase consists in broadcasting some messages (possibly none), retrieving messages, and updating the process state. At the end of a phase a correct process exits the current phase and starts the next phase. We consider that message sending and state updating are instantaneous, because their execution times are negligible in comparison to message transmission delays. This means that the duration of a phase is given by the amount of time dedicated to message retrieval.

Partial synchrony. We assume a partially synchronous system, where after some unknown time τ (the global stabilization time, GST) the system becomes synchronous and channels reliable, that is, there is a finite *unknown* bound δ on the message transfer delay. Before τ the system is asynchronous and channels are lossy.

We assume that processes have access to local clocks and that after τ these clocks are loosely synchronized: at any time after τ, the difference between the real time and the local clock of a process is bounded by some constant ρ, which, as δ, is a priori unknown.

Fault model. Processes can be *correct* or *faulty*. Correct processes follow the protocol, while faulty ones exhibit Byzantine behavior by arbitrarily deviating from the protocol.

Communication primitives. We assume the presence of two communication primitives built on top of point-to-point channels, where exchanged messages are authenticated. The first primitive is a best-effort broadcast primitive used by processes participating in a consensus instance and the second is a pull primitive which can be used by any process.

Broadcasting messages is done by invoking the primitive broadcast. This primitive provides the following guarantees: (i) integrity, meaning that each message is delivered at most once and only if some process previously broadcast it; (ii) validity, meaning that after τ if a correct process broadcasts a message m at time t, then every correct process receives m by time $t + \delta$. For simplicity, we assume that processes also send messages to themselves. Processes are notified of the reception of a message with a NewMessage event.

The pullChain primitive is used by a process to retrieve output values from other processes. This primitive guarantees that, if invoked by a process p at some time $t > \tau$, then p will eventually receive all the output values that correct processes had before t. We note that the pull primitive can be implemented in such a way that the caller does not need to pull all output values, but only the ones that it misses. Furthermore, output values can be grouped and thus received as a chain of values. Processes are notified of the reception of a chain with a NewChain event.

3 Dynamic Repeated Consensus

3.1 Problem definition

Originally, repeated consensus was defined as an infinite sequence of consensus instances executed by the *same* set of processes, with processes having to agree on an infinitely growing sequence of decision values [13]. Dynamic repeated consensus, instead, considers that each consensus instance is executed by a potentially different set of n processes where n is a parameter of the problem. More precisely, given the i-th consensus instance, only n processes $\Pi_i \subseteq \Pi$ participate in the consensus instance proposing values and deciding a unique value v_i. Processes in $\Pi - \Pi_i$ can only adopt v_i. Therefore output values can be either directly decided or adopted. We assume that every correct process agrees a priori on a value v_0.

To know the committee, each process has access to a deterministic selection function committee that returns a sequence of processes based on previous output values. More precisely, the committee Π_i is given by committee$([v_0])$ for $i \leq k$ and by committee$(\bar{v}_p[..(i-k)])$ for $i > k$, where $k > 0$ is a problem parameter, $\bar{v}_p$ denotes the sequence of output values of process p, and $\bar{s}[..j]$ denotes the prefix of length $j + 1$ of the sequence $\bar{s}$. Each process calls committee with its own decided values; however since decided values are agreed upon, committee returns the same sequence when called by different correct processes. We note that the sets Π_i are potentially unrelated to each other, and any pair of subsequent committees may differ. However, we assume that in each committee, less than a third of the members are faulty. For convenience, we consider the worst case: $n = 3f + 1$, and each committee contains exactly f faulty processes.

Dynamic repeated consensus, as repeated consensus, needs to satisfy three properties: agreement, validity, and progress. Agreement and progress have the same formulation for both problems. However, validity needs to reflect the dynamic aspect of committees. To this end, we define validity employing two predicates. The first one is isLegitimateValue. When given as input a value v_i, isLegitimateValue(v_i) returns true if the value has been proposed by a legitimate process, e.g., a process in Π_i. The second predicate is isConsistentValue. When given as input two consecutive output values v_i, v_{i-1}, isConsistentValue(v_i, v_{i-1}) returns true if v_i is consistent with v_{i-1}. This predicate takes into account the fact that an output value depends on the previous one, as commonly assumed in blockchains. For instance, when output values are blocks containing transactions, a valid block must include the identifier or hash of the previous block, and transactions must not conflict with those already decided. For conciseness, we define isValidValue(v_i, v_{i-1}) as a predicate that returns true if both isLegitimateValue(v_i) and isConsistentValue(v_i, v_{i-1}) return true for $i > 0$. Note that the use of an application-defined predicate for stating validity already appears in [2, 12].

An algorithm that solves the *Dynamic Repeated Consensus* problem must satisfy the following three properties:

- (**agreement**) At any time, if $\bar{v}_p$ and $\bar{v}_q$ are the sequences of output values of two correct processes p and q, then $\bar{v}_p$ is a prefix of $\bar{v}_q$ or $\bar{v}_q$ is a prefix of $\bar{v}_p$.
- (**validity**) At any time, if $\bar{v}_p$ is the sequence of output values of a correct process p, then the predicate isValidValue($\bar{v}_p[i]$,$\bar{v}_p[i-1]$) is satisfied for any $i > 0$.
- (**progress**) For any time t, there is a later $t' > t$ such that the sequence of output values of a correct process at time t is a strict prefix of the sequence of output values at time t'.

We use $\bar{s}[i]$ to denote the $(i+1)$-th element of the sequence $\bar{s}$.

3.2 A DRC solution for blockchains

3.2.1 Preliminaries

A *blockchain* is a sequence of linked blocks. The *head* of a blockchain is the last block in the sequence. The block *level* is its position in the sequence, with the first block having level 0. We call this block *genesis*. A block has a *header* and a *content*. The content typically consists of a sequence of transactions; it is application-specific and therefore we do not model it further. The block header includes the level of the block and the hash of the previous block, among other fields detailed later.

In a nutshell, the proposed DRC algorithm works as follows. At each level, for a block b which is proposed to be appended to the blockchain, processes run a single-shot consensus algorithm to agree on the tuple (u, h), where u is the content of b and h is the hash of the predecessor of b. Therefore, we consider that the output values in $\bar{v}$ from the DRC definition in Section 3.1 are the agreed upon tuples (u, h).

Intuitively, the block content is what needs to be agreed upon at a given level. Thanks to block hashes, the agreement obtained during a single-shot consensus instance implies agreement on the whole blockchain, *except for its head*, for which there might not yet be agreement on the other fields of the header besides the predecessor hash. The possible "disagreement" comes from processes taking a decision at possibly different times and thus on different proposed blocks which, however, share the same content. Agreement on the head is obtained implicitly at the next level. For clarity, we refer to a block as being *committed* if it is not the head of the blockchain of a correct process.

```
1   proc runDRC()
2     schedule onTimeoutPull() to be executed after I
3     updateState([genesis], ∅)
4     while true
5       initConsensusInstance()
6       (chain, certificate) := runConsensusInstance()
7       updateState(chain, certificate)

8   proc updateState(chain, certificate)
9     # NB: tail of blockchain_p is a prefix of chain
10    blockchain_p := chain
11    headCertificate_p := certificate
12    ℓ_p := length(chain)
13    h_p := hash(blockchain_p[ℓ_p − 1])

14  proc onTimeoutPull()
15    pullChain
16    schedule onTimeoutPull() to be executed after I

17  proc handleEvents()
18    while not stopEventHandler() do
19      upon NewMessage(msg)
20        handleConsensusMessage(msg)
21      upon NewChain(chain, proposalOrCertificate)
22        certificate := getCertificate(proposalOrCertificate)
23        if validChain(chain, certificate) then
24          if length(chain) > ℓ_p then
25            return (chain, certificate)
26          else if length(chain) = ℓ_p ∧ betterHead(chain, proposalOrCertificate) then
27                updateState(chain, certificate)
```

Figure 1 DRC entry point and auxiliary procedures.

In order for processes to validate a chain independently of the current consensus instance, a certificate is included in the block header to justify the decision on the previous block. A *certificate* is a quorum of signatures which serves as a justification that the content of the predecessor block was agreed upon by the "right" committee. To effectively check certificates, the public keys of committee members are stored in the blockchain.

3.2.2 A DRC algorithm

Fig. 1 presents the pseudocode of a *generic* procedure to solve DRC in the context of blockchains. It is generic in that it can run with *any* single-shot consensus algorithm.
We first enumerate the state variables at any correct process p. Namely, the state of p:

- $blockchain_p$, its local copy of the blockchain;
- ℓ_p, the level at which p runs a consensus instance, which equals the blockchain's length;
- h_p, the hash of the head of $blockchain_p$, that is, of the block at level $\ell_p - 1$;
- $headCertificate_p$, the certificate which justifies the head of $blockchain_p$.

In the pseudocode, all these state variables are considered global, while variables local to a procedure are those that do not have a subscript.

Next, we refine the answer to pullChain requests, in that we consider that the pullChain primitive retrieves more than just output values. Concretely, when a correct process p at level ℓ_p answers a pullChain request, it returns a tuple $(blockchain_p, proposalOrCertificate)$ where $blockchain_p$ is its local chain and $proposalOrCertificate$ is either: (1) the block that p considers as the current proposal at level ℓ_p or; (2) in absence of a proposal, $headCertificate_p$. Here, by *proposal* we mean a proposed block.

We now proceed to describing the entry point of the DRC algorithm, that is, the procedure runDRC in Fig. 1. Processes need not start DRC at the same time. When executing runDRC, a process starts by scheduling calls to pullChain. Then, using updateState,

it initializes its local variables, namely the state variables already presented and the variables specific to the single-shot algorithm. We use the function hash to compute the hash of some input. The function length returns the length of an input sequence.

After updating its state, the process iteratively runs consensus instances and once an instance has finished, it updates its state accordingly. Normally, a consensus instance simply decides on a value, and the corresponding block is appended to the blockchain. However, a process might also be behind other processes which have already taken decisions for more than one level. In this case, as soon as the process invokes the pullChain primitive, it retrieves missed decisions and thus possibly more blocks are appended to the blockchain.

In the presence of dynamic committees, it is not enough that processes call pullChain punctually when they are behind. Indeed, assume that a process p decides at level ℓ but the others are not aware of this and have not decided, because the relevant messages were lost; also assume that p is no longer a member of the committee at level $\ell + 1$, consequently, it no longer broadcasts messages and thus the other processes cannot progress. To solve this, each process invokes pullChain regularly, every I time units, where $I > 0$ is some constant.

During the execution of a consensus instance, processes continuously handle events to update their state. The event processing loop is implemented by the handleEvents procedure in Fig. 1. The termination of the event handler is controlled by the stopEventHandler procedure, which is specific to the single-shot consensus algorithm. There are two kinds of events: message receipts, represented by the NewMessage event, and chain receipts, represented by the NewChain event. Upon receiving a new message *msg*, a process p dispatches it to the consensus instance. Upon the receipt of a new chain, p updates its state accordingly:

- If the new chain is longer, and is valid, p starts a new consensus instance for a higher level; this is because the **return** on line 25 passes the control back to the runDRC procedure in line 6.
- If the new chain has the same length but a head which is "better", in some sense that specific to the single-shot consensus algorithm, then this signals to p that it is "behind", and in this case p only updates its state while remaining at the same level. In particular, only the DRC-related state is updated, while the single-shot instance remains unchanged. A specific betterHead procedure in given in Section 6. For the moment, we note that by means of betterHead, all processes have the same reference point for synchronization.

The NewChain event has, in addition to the *chain* parameter, the *proposalOrCertificate* parameter, which serves as a justification that the head's value has indeed been agreed upon. The role of validChain(*chain*) is two-fold:

1. to check whether *chain*'s head and the certificate from *proposalOrCertificate* match; for this to be possible, we assume access to a procedure getCertificate provided at the single-shot consensus level (see Section 6.5);
2. to check whether for any level ℓ the predicate isValidValue($chain[\ell]$, $chain[\ell-1]$) is satisfied; this means that the hash field in the header of the block $chain[\ell]$ equals hash($chain[\ell-1]$) (so that the predicate isConsistentValue is satisfied), and that the value in each block is proposed by the right committee (so that the predicate isLegitimateValue is satisfied); for the latter to be possible, certificates are stored in blocks as single-shot consensus specific elements (see Section 6.2).

The DRC solution we presented is generic, one can instantiate it by providing implementations to initConsensusInstance, startConsensusInstance, getCertificate, betterHead, and stopEventHandler. We show how to concretely implement them in Sections 5 and 6.

4 A synchronizer for blockchains

We describe a synchronizer for *round*-based consensus algorithms. Round-based consensus algorithms progress in rounds, where, at each round, processes attempt to reach a decision, and if they fail, they advance to the next round to make another attempt.

In the context of round-based consensus algorithms, a standard way to achieve termination of a single consensus instance is to ensure that processes remain at the same round for a sufficiently long period of time [14, 8, 16, 10]. The synchronizer we propose realizes this by leveraging the immutability of the blockchain. One feature of our synchronizer is that it does not exchange any message, thus, it does not increase the communication complexity. Instead, it relies on rounds having the same duration for all processes. We require that rounds duration are increasing and unbounded. Concretely, the duration of a round $r > 0$ is given by $\Delta(r)$, where Δ is a function with domain $\mathbb{N} \setminus \{0\}$ such that, for any duration $d \in \mathbb{N}$, there is a round r with $\Delta(r) \geq d$. Furthermore, we assume that rounds duration are larger than the clock skew, so that rounds are not skipped in the synchrony period. Note that by using round durations, Tenderbake cannot be optimistic responsive like, for instance, [29].

▶ Remark 1. In practice, given estimates δ_{real} of the real message delay and δ_{max} of the maximum message delay δ, we would choose Δ such that: (i) $\Delta(1)$ is slightly bigger than δ_{real}, (ii) Δ increases rapidly (e.g. exponentially) till it reaches δ_{max}, and (iii) then it increases slowly (e.g. linearly) afterwards.

To determine at which round the process should be, the synchronizer relies on local clocks. Therefore, when clocks are synchronized, all processes will be at the same round. However, a prerequisite is that processes agree on the starting time of the current instance. As different processes may decide at different rounds, and therefore at different times, there is a priori no consensus about the start time of an instance. We adopt a solution based on the following observation: if the round at which a decision is taken is eventually known by all processes, then they can agree on a common global round at which a consensus instance is considered to have terminated. Indeed, a process considers that the consensus instance has ended at the smallest round at which some process has decided.

The above solution can be implemented by (1) considering that a block header stores the round at which the block is produced, and (2) using the betterHead procedure, which is called by a process p at line 27 upon receiving a new chain in response to a pullChain request. This procedure checks if some other process has already taken a decision sooner, in terms of rounds. If this is the case, betterHead signals to its caller that it is "behind" and thus that it needs to resynchronize. We postpone the concrete implementation of betterHead to Section 6.4 because it is specific to the single-shot consensus algorithm. For the moment, to illustrate the role of betterHead, Fig. 2 shows an update of the head of a process p's blockchain. Initially, the head of p's local chain is b'. Then, p sees the block b'' at level ℓ with a smaller round than b' and therefore updates the head of its local chain to b''.

Finally, we present the synchronization procedure in Fig. 3. We assume that the genesis block contains the time t_0 of its creation. To synchronize, p uses its local clock, whose value is obtained by calling now(), and the rounds of the blocks in its blockchain to find out what its current round and the time position within this round should be. Process p determines first the starting time of the current level and stores it in t. To do this, p adds the durations of all rounds for all previous levels. Once p has determined t, it finds the current round by checking incrementally, starting from round $r = 1$ whether the round r is the current round: r is the current round if there is no time left to execute a higher round. The variable t is updated to represent the time at which round r started. The difference $t' - t$ represents the offset between the beginning of the round r and the current time.

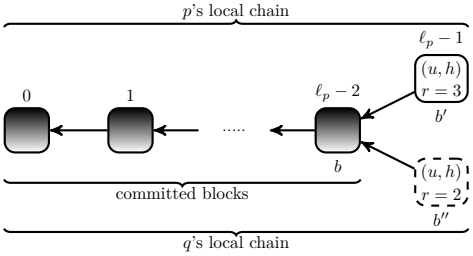

Figure 2 An update of the head of p's chain. Solid boxes represent blocks in p's chain before the update, while the dashed box represents the block that triggers the update. Block levels and labels are given above and respectively below the corresponding boxes. The hash h is that of block b.

```
28  proc synchronize()
29    t' := now()
30    t := t_0 + Σ_{ℓ=0}^{ℓ_p−1} Σ_{j=1}^{round(ℓ)} Δ(j) + Σ_{j=1}^{r'} Δ(j)
31    r := 1
32    while t + Δ(r) ≤ t' do
33      t := t + Δ(r)
34      r := r + 1
35    return (r, t' − t)
```

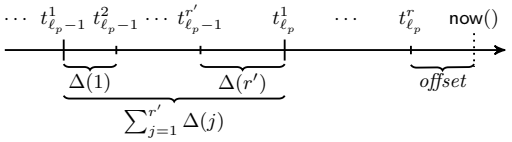

Figure 3 A round-based synchronizer and a timeline. Small/large vertical lines represent round/level boundaries, respectively.

Fig. 3 also illustrates the timeline of a process that increments its rounds using the procedure synchronize, where $t^r_{\ell_p}$ represents the starting time of the round r of level ℓ_p and r' stands for the last round of level $\ell_p - 1$. The figure also illustrates the offset $t' - t$.

5 A Single-Shot Consensus Skeleton

In this section we give a generic implementation for the procedure runConsensusInstance from Section 3.2.2. Here we make another standard assumption on the structure of the single-shot consensus algorithm, namely that each round evolves in sequential *phases*. For instance, PBFT in normal mode has 3 phases (named *pre-prepare*, *prepare*, and *commit*), Tendermint as well, DLS and Hotstuff have 4 phases, etc.

We let m denote the number of phases. As for rounds, we assume that each phase has a predetermined duration. The duration is given by the round r it belongs to, and it is denoted $\Delta'(r)$. For simplicity, we assume that $\Delta(r) = m \cdot \Delta'(r)$. We also refine the assumption on round durations, and also require that phase durations are larger than the clock skew, so that phases are not skipped in the synchrony period, i.e. $\Delta'(1) > 2\rho$.

To synchronize correctly, a process also needs to update its phase (not only its round) and to know its time position within a phase. These can be readily determined from the round and the round offset returned by synchronize. The procedure getNextPhase, presented in Fig. 4, performs this task. For the pseudocode, we consider that each phase has a label identifying it and we use *phases* to denote the sequence of phase labels.

The entry point of a single-shot consensus instance is runConsensusInstance, given in Fig. 4. As part of its state, a process p also maintains its current round r_p. A process p starts by calling synchronize in an attempt to (re)synchronize with other processes. We recall that this is just an attempt and not a guarantee because clocks are not necessarily synchronized before τ. If synchronize returns that p should be at a round in the past with respect to p's

```
36  proc runConsensusInstance()
37    (round, roundOffset) = synchronize()
38    if r_p > round then  # p is ''ahead''
39      runConsensusInstance()
40    else # p is ''behind''
41    (phase, phaseOffset) :=
42                    getNextPhase(round, roundOffset)
43      r_p := round
44      set runEventHandler timer to
45                                  Δ'(r_p) − phaseOffset
46      if p ∈ committeeAtLevel(ℓ_p) then
47        goto phase
48      else
49        goto phase-observer
```

```
50  proc getNextPhase(round, roundOffset)
51    i := roundOffset / Δ'(round)
52    phase := phases[i]
53    phaseOffset := roundOffset − i · Δ'(round)
54    return (phase, phaseOffset)

55  proc advance(decisionOption)
56    match decisionOption with
57      | Some (block, blockCertificate) →
58          c := blockchain_p ++ block
59          return (c, blockCertificate)
60      | None → # no decision
61          r_p := r_p + 1
62          filterMessages()
63          runConsensusInstance()
```

Figure 4 Entry point and progress procedures for generic single-shot consensus.

current round, then p invokes (indirectly) the synchronizer again. This active waiting loop ensures that p is ready to continue its execution as soon as it is not "ahead" anymore. We note that a jump backward to a previous round or phase may jeopardize safety. When p is "behind", it first uses the procedure getNextPhase to obtain the phase at which it should be. Next, it updates its round and the timer used to time the execution of the event handler. Concretely, through this timer, the generic procedure stopEventHandler is implemented as follows:

```
64  proc stopEventHandler()
65    return true iff timer runEventHandler expired
```

We recall this procedure is used by handleEvents at line 18 in Fig. 1.

After setting *runEventHandler*, p checks whether it is part of the committee for level ℓ_p. To this end, we assume having access to a committeeAtLevel function, which returns the committee at some given level ℓ. This function corresponds to committee$(\bar{v}_p[..(\ell - k)])$ (Section 3.1), where $\bar{v}_p$ is the sequence of output values of the caller process p. Finally, p executes the single-shot consensus algorithm according to its role and to the phase returned by getNextPhase. The determined phase is executed by means of an unconditional jump to corresponding phase label. The two **goto** statements in Fig. 4 are intentionally symmetric for committee and non-committee members to keep all processes in sync. This has the advantage of not introducing delays when they eventually become part of the committee.

Fig. 4 also shows the advance procedure, which is used by processes to handle the progress of the current consensus instance by either returning the control to runDRC when a decision can been taken; or otherwise increasing the round. In this former case, advance first prepares the updated blockchain, appending the block corresponding to the decision to its current blockchain; runDRC will then update the state accordingly, for instance increasing the level. The procedure advance has one parameter, which is optional, represented in the pseudocode as a value of an optional type (with values of the form Some x if the parameter is present or None if it is not). The parameter is present when the current consensus instance has taken a decision. In this case, the parameter is a tuple consisting of a block containing the decided value and of a certificate justifying the decision. Otherwise, when no decision is taken, the process increases its round and filters its message buffer by removing messages no longer necessary. The filtering procedure filterMessages is specific to the consensus instance.

We conclude by presenting in Fig. 5 the pseudocode capturing the behavior of the processes which are not part of a committee for a given level. We call such processes *observers*. Contrary to committee members, observers are passive in the sense that they only receive (but not send) messages and update their state accordingly.

```
66  phases[1]-observer phase:
67    handleEvents()
68  ⋮
69  phases[m − 1]-observer phase:
70    handleEvents()

71  phases[m]-observer phase:
72    handleEvents()
73    advance(getDecision())
```

Figure 5 Generic single-shot algorithm for an observer.

This observer behavior serves two purposes:

i) to keep the blockchain at each observer up to date;

ii) to check at the end of the round whether a decision was taken, and if so, whether the observer becomes a committee member at the next level.

To achieve i), the observer checks if it can adopt a proposed value. It does so by invoking the handleEvents and advance procedures, where the parameter to advance is obtained using the procedure getDecision, which is specific to the single-shot consensus algorithm. Concerning ii), when the corresponding check (line 46) is successful, the observer switches roles and acts as a committee member. We note that line 46 is reached when the observer end its round and calls advance, which in turn calls runConsensusInstance at the end.

As for the DRC solution in Section 3.2.2, the methods presented in this section are generic. One can instantiate them by providing implementations to the filterMessages and getDecision procedures. We show such concrete implementations in the next section.

6 Single-shot Tenderbake

To show the specific phase behavior of a committee member, we first introduce some terminology inspired by Tezos. Tenderbake committee members are called *bakers*. At each round, a value is proposed by the proposer whose turn comes in a round-robin fashion. Tenderbake has three types of phases: PROPOSE, PREENDORSE, and ENDORSE, each with a corresponding type of message: Propose for proposals, Preendorse for preendorsements, and Endorse for endorsements. A fourth type of message, Preendorsements, is for the retransmission of preendorsements. A baker *proposes*, *preendorses*, and *endorses* a value v (at some level and with some round) when the baker broadcasts a message of the corresponding type. Only one value per round can be proposed or (pre)endorsed. A set of at least $2f + 1$ (pre)endorsements with the same level and round and for the same value is called a *(pre)endorsement quorum certificate* (QC).

We consider that Propose messages are blocks. This is a design choice that has the advantage that values do not have to be sent again once decided.

Within a consensus instance, if a baker p receives a preendorsement QC for a value v and round r, then p keeps track of v as an *endorsable value* and of r as an *endorsable round*. Similarly, if a baker p receives a preendorsement QC for a value v and round r during the ENDORSE phase of the round r, then p locks on the value v, and it keeps track of v as a *locked value* and of r as a *locked round*. Note that the locked round stores the most recent round at which p endorsed a value, while the endorsable round stores the most recent round that p is aware of at which bakers may have endorsed a value.

The execution of a round works as follows. During the PROPOSE phase, the designated proposer proposes a value v, which can be newly generated or an endorsable value from a previous round r. During the PREENDORSE phase, a baker preendorses v if it is not locked or if it is locked on a value at a previous round than r; in particular, it does not preendorse

v if it is locked and v is newly generated. If a baker does not preendorse v, then it sends a Preendorsements message with the preendorsement QC that justifies its more recent locked round. During the ENDORSE phase, if bakers receive a preendorsement QC for v, they lock on it and endorse it. If bakers receive an endorsement QC for v, they *decide* v.

Tenderbake inherits from classical BFT solutions the two voting phases per round and the locking mechanism. Tracking endorsable values is inherited from [7]. Tenderbake distinguishes itself in a few aspects which we detail next.

Preendorsement QCs. For safety, bakers accept endorsable values only from higher rounds than their locked round. Assume a correct baker p locks and all other correct bakers locked at smaller rounds. Assume also that the messages from p are lost. To prevent p from not making progress, it is enough to include the preendorsement QC that made p lock in Endorse and Propose messages. In this way, bakers can update their endorsable values and rounds accordingly and propose values that can be accepted by any correct locked baker. Tendermint does not need such QCs as it assumes reliable communication in the asynchronous period.

The Preendorsements message. For faster termination of a consensus instance, when a baker refuses a proposal because it is locked on a higher round than the endorsable round of the proposed value, it broadcasts a Preendorsements message. This message contains a preendorsement QC justifying its higher locked round. During the next round, bakers use this QC to set their endorsable value to the one with the highest round. The consensus instance terminates with the first correct proposer. Thus, in the worst-case scenario, when the first f bakers are Byzantine, Tenderbake terminates in $f+2$ rounds after τ, assuming that processes have achieved round synchronization and that the round durations are sufficiently large.

Endorsement QCs. For processes to be able to check that blocks received by calling pullChain are already agreed upon, each block comes with an endorsement QC for the block at the previous level. Furthermore, for the same reason, in response to a pull request, a process also attaches the endorsement QC that justifies the value in the head of the blockchain.

6.1 Process state and initialization

In addition to the variables mentioned in Section 3.2.2, a process p running Tenderbake maintains its current round r_p as well as:

- $lockedValue_p$ and $lockedRound_p$ to keep track respectively of the value on which p is locked and the round during which p locked on it,
- $endorsableValue_p$ to keep track of the proposed value with a preendorsement QC (with the highest round), which can therefore be considered endorsable,
- $endorsableRound_p$ and $preendorsementQC_p$ to store the round and the preendorsement QC corresponding to an endorsable value;
- $headCertificate_p$ to store the endorsement QC for p's last decided value.

The variable $headCertificate_p$ (introduced in Section 3) is empty at level 1 (Fig. 1, line 3). The state of a process is initialized by the procedure initConsensusInstance:

```
74  proc initConsensusInstance()
75    r_p := 1
76    lockedValue_p := ⊥; lockedRound_p := 0
77    endorsableValue_p := ⊥; endorsableRound_p := 0
78    preendorsementQC_p := ∅
79    messages_p := ∅
```

where, by abuse of notation, we use $x := \perp$ to denote that x has become undefined.

6.2 Messages and blocks

We write messages using the following syntax: $type_p(\ell, r, h, payload)$, where *type* is Propose, Preendorse, Endorse, or Preendorsements, p is the process that sent the message, ℓ and r are the level and the round during which the message is generated, h is the block hash at level $\ell - 1$, and *payload* is the type specific content of the message.

The payload (eQC, u, eR, pQC) of a Propose message contains the endorsement quorum eQC that justifies the block at the previous level and the proposed value u to be agreed on. The payload also contains, in case u is a previously proposed value, the corresponding endorsable round and the preendorsement QC that justifies u. If the proposed value is new, then eR is 0 and pQC is the empty set.

Given a Propose$(\ell, r, h, (eQC, u, eR, pQC))$ message, the corresponding block has contents u, while the remaining fields, notably the hash h, are part of the block header.

The payload of a Preendorse message consists of the value to be agreed upon while the payload of an Endorse message consists of an endorsed value. The payload of a Preendorsements message consists of a preendorsement QC justifying some endorsable value and round.

6.3 Message management

The message management is designed such that message buffers are bounded. We prove this in Lemma 3 and we give some more intuition in Appendix A. In this section, we only focus on the elements needed to understand single-shot Tenderbake, namely the handleConsensusMessage and some helper procedures. The procedure handleConsensusMessage is depicted in Fig. 6. A process p adds (line 84) to its message buffer valid messages *msg* but only from the current and next round. Messages from the next round are needed in order to cater for the possible clock drift. Moreover, if a preendorsement QC is observed for a higher round than the current $endorsableRound_p$, then p updates $endorsableValue_p$, $endorsableRound_p$, and $preendorsementQC_p$ using the procedure updateEndorsable (line 85). Finally, as an optimization, if the received message is from either a higher level or from the same level but with a different hash, then p attempts to resynchronize by calling pullChain (line 87).

The procedure filterMessages() removes messages not for the current round (see Appendix A). The helper procedures used in Fig. 6 are described as follows:

- proposedValue() returns the current proposed value of the block at level ℓ;
- valueQC(qc) and roundQC(qc) return the value and respectively the round from a qc;
- pQC(*msg*) returns the preendorsement QC from a Propose or Preendorsements message *msg*; if the Propose message does not contain a preendorsement QC (because what is proposed is a new value), then pQC returns the empty set;
- proposal(), preendorsements(), and endorsements() return the proposal, preendorsements, and respectively the endorsements contained in *messages*.

6.4 Tenderbake main loop

Fig. 7 gives the execution of one round of Tenderbake by baker p, when the round's three phases are executed in sequence. We recall that the pseudocode has the same structure as that for observers, as described in Section 5. Each phase consists of a conditional broadcast followed by a call to handleEvents (described in Section 3.2.2). In addition, the ENDORSE phase calls advance (described in Section 3.2.2). In the PROPOSE phase, p checks if it is the proposer for the current level ℓ_p and round r_p (line 102). If so, p proposes:

```
80  proc handleConsensusMessage(msg)
81    let type_q(ℓ, r, h, payload) = msg
82    if ℓ = ℓ_p ∧ h = h_p ∧ (r = r_p ∨ r = r_p + 1) then
83      if isValidMessage(msg)
84        messages_p := messages_p ∪ {msg}
85        updateEndorsable(msg)
86    if (ℓ = ℓ_p ∧ h ≠ h_p) ∨ ℓ > ℓ_p then
87      pullChain

88  proc updateEndorsable(msg)
89    if |preendorsements()| ≥ 2f + 1 then
90      endorsableValue_p := proposedValue()
91      endorsableRound_p := r_p
92      preendorsementQC_p := preendorsements()
93    else if type(msg) ∈ {Propose, Preendorsements} then
94      pQC := pQC(msg)
95      if pQC ≠ ∅ ∧ roundQC(pQC) > endorsableRound_p then
96        endorsableValue_p := valueQC(pQC)
97        endorsableRound_p := roundQC(pQC)
98        preendorsementQC_p := pQC

99  proc filterMessages()
100   messages_p := messages_p \ {type(ℓ, r, h, payload) ∈ messages_p | r ≠ r_p}
```

Figure 6 Message management in Tenderbrake.

- either a new value u, returned by the procedure newValue; here it is assumed that u is consistent with respect to the value u' contained in the last block of the blockchain of the process that calls this procedure; that is, isConsistentValue(v, v') holds (see Section 3.1), where v, v' are the output values corresponding to u, u';
- or its $endorsableValue_p$ if defined; in this case, p includes in the payload of its proposal the corresponding endorsable round and the preendorsement QC that justifies it.

The payload also includes the endorsement QC to justify the decision for the previous level.

In the PREENDORSE phase, p checks if the value u from the Propose message received from the current proposer is preendorsable (lines 110-111). Namely, it checks whether one of the following conditions are satisfied:

- p is unlocked ($lockedRound_p = 0$, thus the second disjunction at line 111 is true); or
- p is locked (i.e. $lockedRound_p > 0$), u was already proposed during some previous round (i.e. $0 < eR < r_p$), and:
  - p is already locked on u itself (thus the first disjunction at line 111 is true); or
  - p is locked on $u' \neq u$ and its locked round is smaller than the endorsable round associated to u.

In the second case, there is a preendorsement QC for u and round eR, thanks to the validity check on the Propose message. If the condition holds, then p preendorses u. If p cannot preendorse u as it is locked on some value $u' \neq u$ with a higher locked round than eR, then p broadcasts the preendorsement QC that justifies v'. If received on time, this information allows the next proposer to choose a value that passes the checks at all correct bakers.

In the ENDORSE phase, p checks if it received a preendorsement QC for the proposed value u. If yes, p updates its *lockedValue* and *endorsableValue* and broadcasts its Endorse message, along with all the Preendorse messages for u (lines 117-120). Note also that in this case p has already updated its endorsable value to u and its endorsable round to r_p while executing handleEvents.

Finally, at the end of this last phase, which is also the end of the round, bakers call advance with a parameter that signals whether a decision can be taken or not. This parameter is obtained using getDecision, implemented is as follows:

```
101  PROPOSE phase:
102    if proposer(ℓ_p, r_p) = p then
103      u := if endorsableValue_p ≠ ⊥ then endorsableValue_p
104           else newValue()
105      payload := (headCertificate_p, u, endorsableRound_p, preendorsementQC_p)
106      broadcast Propose_p(ℓ_p, r_p, h_p, payload)
107    handleEvents()

108  PREENDORSE phase:
109    if ∃q, eQC, u, eR, pQC :
110        Propose_q(ℓ_p, r_p, h_p, (eQC, u, eR, pQC)) ∈ messages_p ∧
111        (lockedValue_p = u ∨ lockedRound_p < eR < r_p) then
112      broadcast Preendorse_p(ℓ_p, r_p, h_p, hash(u))
113    else if lockedValue_p ≠ ⊥ then
114      broadcast Preendorsements(ℓ_p, r_p, h_p, preendorsementQC_p)
115    handleEvents()

116  ENDORSE phase:
117    if |preendorsements()| ≥ 2f + 1 then
118      u := proposedValue()
119      lockedValue_p := u;  lockedRound_p := r_p
120      broadcast Endorse_p(ℓ_p, r_p, h_p, hash(u))
121      broadcast preendorsementQC_p
122    handleEvents()
123    advance(getDecision())
```

Figure 7 Single-shot Tenderbake for baker p.

```
124  proc getDecision()
125    if |endorsements()| ≥ 2f + 1 then
126      return Some (proposal(), endorsements())
127    else
128      return None
```

6.5 The betterHead procedure

The role of betterHead is to make processes agree on the same blockchain head; recall that they already agree on the head contents, but not necessarily on the head's header. Agreeing on the same blockchain head has in turn two roles:

- allowing agreement on the round at which a decision was taken at the previous level, which is one of the ingredients for processes to synchronize at the current level, as explained in Section 4.
- allowing agreement to take place at the current level; recall that at the current level agreement needs to be reached also on the hash of the block at the predecessor level, that is, on the hash of the head of a process' blockchain.

To reach these two goals, as suggested in Section 4, processes adopt the head with the smallest round. However, there is a caveat: if this would be the only check done by betterHead, processes might end up with a head on top of which no proposal will be accepted in case they have seen an endorsable value: indeed, the hash component of such a value may not match the new head. To avoid this situation, a process first performs an additional check in case they have seen an endorsable value. When *proposalOrCertificate* is a proposal, the check is similar to the check for preendorsing (line 111): the endorsable round of process p is smaller than the one in the received proposal (line 133). When *proposalOrCertificate* is a certificate we simply required that the process has not seen an endorsable value. The betterHead procedure implementing these checks is given next.

```
129  proc betterHead(chain, proposalOrCertificate)
130    let ⟨_, r, . . . ; ·⟩ = head(chain)
131    match proposalOrCertificate with
132    | Propose(_, _, _, (_, _, eR, _)) →
133      return endorsableRound_p < eR ∨ (endorsableRound_p = eR ∧ r < round(ℓ_p − 1))
134    | _ → # proposalOrCertificate is a certificate
135      return endorsableRound_p = 0 ∧ r < round(ℓ_p − 1)
```

In the pseudocode, $\langle \ldots ; \ldots \rangle$ denotes a block, with the part before the semicolon representing the block's header and the part after it its contents. The procedure head($chain$) returns the head of $chain$. Also, recall that round(ℓ) returns the round contained in the header of the block at level ℓ in the caller's blockchain.
As for the implementation of getCertificate, it is a simple match on *proposalOrCertificate*:

```
136  proc getCertificate(proposalOrCertificate)
137    match proposalOrCertificate with
138      | Propose_q(_, _, _, (eQC, _, _, _)) → return eQC
139      | eQC → return eQC
```

7 Correctness and Complexity

The following theorem states that Tenderbake provides a solution to DRC. Its proof can be found in Appendix B.

▶ **Theorem 2.** *Tenderbake satisfies validity, agreement, and progress.*

Bounded memory. We assume that all values referred to by global or local variables of a process p are stored in volatile memory, except for the variable $blockchain_p$ whose value is stored on disk. We recall that the message buffer is represented by the $messages_p$ variable. The following lemma shows that a process can use fixed-sized buffers, namely of size $4n$.

▶ **Lemma 3.** *For any correct process p, at any time, $|messages_p| \leq 4n + 2$.*

Proof. Let p be some correct process. Given that in $messages_p$ only messages from the current and next round are added (line 84), and that with each new round messages from the previous round are filtered out (line 100), $messages_p$ contains at most 2 proposals, at most $2n$ preendorsements, and at most $2n$ endorsements. ◀

The following result states that a process only uses bounded memory. We assume here that the underlying implementation of the pullChain primitive does not count towards the memory usage of a process.

▶ **Theorem 4.** *At any time, the size of the volatile memory of any correct process is in $\mathcal{O}(n)$.*

Proof. A correct process maintains a constant number of variables, and except *messages*, each variable stores a primitive value or a QC. A QC contains at most n messages and each message has a constant size. The $\mathcal{O}(n)$ bound follows from these observations, and the observation concerning the *messages* variable from the proof of Lemma 3. ◀

Message and round complexity. Each round has a message complexity of $O(nm)$ due to the n-to-m broadcast, where m is the current number of processes in the system.

Concerning round complexity, it is known that consensus, in the worst case scenario, cannot be reached in less than $f + 1$ rounds [15]. In Tenderbake, after bakers synchronize and the round durations are sufficiently long (namely, at least $\delta + 2\rho$), a decision is taken in at most $f + 2$ rounds, as already mentioned in Section 6. See Lemma 15 in Appendix B for a proof. Intuitively, f rounds are needed in case the proposers of these rounds are Byzantine. Another round is needed if there is a correct process locked on a higher round than the endorsable round of the proposed value. However, in this case, the next proposer is correct and will have updated its endorsable round, and therefore its proposed value will be accepted and decided by all correct processes.

Recovery time. Finally, we discuss the time required for bakers to synchronize after τ. A worst-case scenario analysis is in our technical report [4]. Roughly, the recovery time is the maximum time between the error that the clock can experience and the time necessary for a process to fetch the missing blocks, which is at least one round-trip time: the time to ask for the current blockchain and to get the reply. We believe that in practice the time to pull a new chain (and even to pull just the last block) is considerably bigger than the maximum error clock that a process can experience during the asynchronous period. Finally, if all processes are at the same level but not at the same round, then, as the synchronizer is called at the end of every round, all processes synchronize in at most one round.

8 Conclusion

In this paper, we proposed a formalization of dynamic repeated consensus, a general approach to solve it, and a BFT solution working with bounded buffers by leveraging a blockchain-based synchronizer. We have implemented the proposed solution in a prototype[1]. A full-fledged (based on proof-of-stake and with smart contracts) implementation is being developed[2]. Experiments with running a Tenderbake testnet are underway. A Tenderbake simulator has already been implemented[3].

Besides practical aspects such as experimenting with the testnet and the simulator, as future work, we see the following exciting directions: explore the relationship between achieving asynchronous responsiveness and providing bounded buffers; improve message size and complexity by means of aggregated or threshold signatures; mechanize the proofs; and analyze Tenderbake from an economic perspective when considering rational agents.

References

1 Victor Allombert, Mathias Bourgoin, and Julien Tesson. Introduction to the Tezos Blockchain. In *Proc. High Performance Computing and Simulation*, 2019.

2 Yackolley Amoussou-Guenou, Antonella Del Pozzo, Maria Potop-Butucaru, and Sara Tucci-Piergiovanni. Correctness of Tendermint-core blockchains. In *Proc. Principles of Distributed Systems*, 2018.

3 Yackolley Amoussou-Guenou, Antonella Del Pozzo, Maria Potop-Butucaru, and Sara Tucci Piergiovanni. Dissecting Tendermint. In *Proc. Networked Systems*, 2019.

[1] `https://gitlab.com/nomadic-labs/tezos/-/tree/tenderbake_proto`
[2] `https://gitlab.com/nomadic-labs/tezos/-/blob/tenderbake-florence`
[3] `https://gitlab.com/nomadic-labs/tenderbake-simulator`

4 Lăcrămioara Aştefănoaei, Pierre Chambart, Antonella Del Pozzo, Thibault Rieutord, Sara Tucci, and Eugen Zălinescu. Tenderbake – a solution to dynamic repeated consensus for blockchains, 2021. arXiv:2001.11965.

5 Alysson Bessani, Eduardo Alchieri, João Sousa, André Oliveira, and Fernando Pedone. From byzantine replication to blockchain: Consensus is only the beginning. In *Proc. International Conference on Dependable Systems and Networks*, 2020.

6 Manuel Bravo, Gregory V. Chockler, and Alexey Gotsman. Making byzantine consensus live. In *Proc. International Symposium on Distributed Computing*, 2020.

7 Ethan Buchman, Jae Kwon, and Zarko Milosevic. The latest gossip on BFT consensus. *CoRR*, 2018. arXiv:1807.04938.

8 Miguel Castro and Barbara Liskov. Practical Byzantine fault tolerance and proactive recovery. *ACM Trans. Comput. Syst.*, 2002.

9 T.-H. Hubert Chan, Rafael Pass, and Elaine Shi. Pala: A simple partially synchronous blockchain. *IACR Cryptol. ePrint Arch.*, 2018.

10 Bernadette Charron-Bost and André Schiper. The heard-of model: computing in distributed systems with benign faults. *Distributed Comput.*, 2009.

11 Jing Chen and Silvio Micali. Algorand: A secure and efficient distributed ledger. *Theor. Comput. Sci.*, 2019.

12 T. Crain, V. Gramoli, M. Larrea, and M. Raynal. (Leader/Randomization/Signature)-free Byzantine consensus for consortium blockchains. *CoRR*, 2017.

13 Carole Delporte-Gallet, Stéphane Devismes, Hugues Fauconnier, Franck Petit, and Sam Toueg. With finite memory consensus is easier than reliable broadcast. In *Proc. Principles of Distributed Systems*, 2008.

14 Cynthia Dwork, Nancy A. Lynch, and Larry J. Stockmeyer. Consensus in the presence of partial synchrony. *J. ACM*, 1988.

15 Michael J Fischer and Nancy A Lynch. A lower bound for the time to assure interactive consistency. *Information processing letters*, 1982.

16 Eli Gafni. Round-by-round fault detectors: Unifying synchrony and asynchrony (extended abstract). In *Proc. ACM Symposium on Principles of Distributed Computing*, 1998.

17 J. A. Garay, A. Kiayias, and N. Leonardos. The bitcoin backbone protocol: Analysis and applications. In *Proc. EUROCRYPT International Conference*, 2015.

18 L.M. Goodman. Tezos – a self-amending crypto-ledger, 2014.

19 Timo Hanke, Mahnush Movahedi, and Dominic Williams. DFINITY technology overview series, consensus system. *CoRR*, 2018.

20 Jae Kwon and Ethan Buchman. Cosmos: A Network of Distributed Ledgers.

21 S. Nakamoto. Bitcoin: A Peer-to-Peer Electronic Cash System, 2008.

22 Oded Naor, Mathieu Baudet, Dahlia Malkhi, and Alexander Spiegelman. Cogsworth: Byzantine view synchronization. *CoRR*, 2019. arXiv:1909.05204.

23 Oded Naor and Idit Keidar. Expected linear round synchronization: The missing link for linear byzantine SMR. In *Proc. International Symposium on Distributed Computing*, 2020.

24 Nomadic Labs. Analysis of Emmy$^+$. https://blog.nomadic-labs.com/analysis-of-emmy.html, 2019.

25 Rafael Pass and Elaine Shi. Rethinking large-scale consensus. *IACR Cryptol. ePrint Arch.*, 2018.

26 Roberto Saltini. Correctness analysis of IBFT. *CoRR*, 2019.

27 Omid Shahmirzadi, Sergio Mena, and André Schiper. Relaxed atomic broadcast: State-machine replication using bounded memory. In *Proc. IEEE International Symposium on Reliable Distributed Systems*, 2009.

28 The LibraBFT Team. State machine replication in the Libra blockchain, 2019.

29 Maofan Yin, Dahlia Malkhi, Michael K Reiter, Guy Golan Gueta, and Ittai Abraham. HotStuff: BFT consensus with linearity and responsiveness. In *Proc. ACM Symposium on Principles of Distributed Computing*, 2019.

A Valid messages and bounded buffers

Recall handleConsensusMessage in Section 6.3. As a necessary check for message buffers to be bounded, upon the retrieval of a new message msg, a process p first checks if the level, round, and hash in msg's header match respectively p's current level, either the current round or the next round, and the hash of the block at the previous level. If yes, p then checks that the message is valid, with the procedure isValidMessage (line 83). $\mathsf{Propose}_q(\ell,r,h,(eQC, u,eR,pQC))$ is *valid* if q is the proposer for level ℓ and round r and if eQC is an endorsement QC for level $\ell-1$ with the round, hash, and value matching those in p's blockchain. In addition, either pQC is empty and eR is 0 (i.e. u is newly proposed), or the round, value, and hash from pQC match eR, u, and h_p, respectively. Messages in eQC and pQC must be valid themselves, in particular they must be generated by bakers at levels $\ell-1$ and ℓ, respectively. These validity checks ensure that the value (u,h) satisfies the isLegitimateValue predicate from Section 3.1. The validity conditions for the other types of messages are similar, and thus omitted. We note, however, that for preendorsements and endorsements it is required that the corresponding proposal has been already received, so that it can be checked that the hash included in the payload matches the proposed value.

There are three additional aspects of handleConsensusMessage in Section 6.3 that together with the validity check, ensure that buffers are bounded: (1) only valid messages are added (line 83); (2) messages for the next round are kept (line 84) to cater for the possible clock drift; (3) messages from higher levels trigger p to ask for the sender's blockchain (line 87), because such messages "from the future" suggest that p is behind; however, the sender might be lying about being ahead. Recall that the procedure advance only calls filterMessages after a round increment (line 62). Recall also that filterMessages removes messages not matching the current round (line 100). Together with the above elements, the filtering ensures that message buffers are bounded (Lemma 3).

B Correctness proof

B.1 Validity and Agreement

▶ **Theorem 5.** *Tenderbake satisfies validity.*

Proof. The local chain of a correct process p is formed by proposals and/or chains obtained by p calling pullChain. In either case, the content of each block satisfies the predicate isValidValue by the definition of either isValidMessage or validChain. ◀

▶ **Lemma 6.** *Correct bakers preendorse and endorse at most once per round at a given level.*

Proof. Preendorse and Endorse messages are sent only during the corresponding phase (line 112 and line 120, respectively). To show that there is at most one Preendorse (resp. Endorse) per round it suffices to show that a phase is executed only once per round. Firstly, phases are executed sequentially. Secondly, non-sequential jumps happen only at line 47 (resp. at line 49) in runConsensusInstance; in turn, runConsensusInstance is called by either advance (line 63), after increasing the round; or runDRC (line 6), after increasing the level (line 7) once a decision is taken (line 59) or a longer chain is received (line 25). ◀

▶ **Lemma 7.** *At most one value can have a (pre)endorsement QC per round.*

Proof. By contraction, using Lemma 6. ◀

We say a baker p is locked on a tuple (u, h) if $lockedValue_p = u$ and $h_p = h$. We define $L^{u,h}_{\ell,r}$ as the set of *correct* bakers locked on the tuple (u, h) at level ℓ and at the end of round r. We also define $\mathsf{preendos}(\ell, r, u, h)$ as the set of preendorsements generated by *correct* processes for some level ℓ, some round r, some value u, and some hash h.

▶ **Lemma 8.** *Let ℓ be a level, r a round, u a value, and h a block hash. For any round $r' \geq r$ and any tuple $(u', h') \neq (u, h)$, if $|L^{u,h}_{\ell,r}| \geq f + 1$, then $|\mathsf{preendos}_p(\ell, r', u', h')| \leq f$.*

Proof. We reason by contradiction. Suppose that $|L^{u,h}_{\ell,r}| \geq f+1$, and let $r' \geq r$ be the smallest round for which there exists a tuple $(u', h') \neq (u, h)$ such that $|\mathsf{preendos}(\ell, r', u', h')| \geq f + 1$. As $|L^{u,h}_{\ell,r}| \geq f + 1$ and $|\mathsf{preendos}(\ell, r', u', h')| \geq f + 1$, there is at least one correct process p such that $p \in L^{u,h}_{\ell,r}$ and p preendorsed (u', h') at round r'. As $p \in L^{u,h}_{\ell,r}$, we have that p is locked on (u, h) at round r. Since p preendorsed (line 112) at round r', it means that one of the two disjunctions at line 111 holds. Note that the value of r_p at line 111 is r' in this case.

Suppose the first disjunction holds, i.e., $lockedValue_p = u'$. As a process can re-lock only in the phase ENDORSE, under the condition at line 117, this means that there is a round r'' with $r \leq r'' < r'$ and at which $|\mathsf{preendorsements}()| \geq 2f + 1$. Therefore $|\mathsf{preendos}(\ell, r'', u', h')| \geq f + 1$. This contradicts the minimality of r'.

Suppose now that the second disjunction holds, that is, $lockedRound_p < r'' < r'$ where the round r'' is the endorsable round of the proposer of u'. We note that a process cannot unlock (i.e. unset *lockedRound*), but only re-lock (i.e. set *lockedRound* to a different value). Therefore $lockedRound_p \geq r$ at round r' and from this, we obtain that $r'' > r > 0$. From the validity requirements of a propose message, we obtain that it contains a preendorsement QC for (u', h'). Thus we have that $|\mathsf{preendos}(\ell, r'', u', h')| \geq f + 1$. This contradicts the minimality of r', since $r'' < r'$. ◀

▶ **Lemma 9.** *No two correct processes have two different committed blocks at the same level in their blockchain.*

Proof. We reason by contradiction. Let ℓ be some level. Assume that two different correct processes p, p' have respectively two different committed blocks b, b' at level ℓ in their blockchain, with $b \neq b'$.

By the definition of committed blocks (Section 3), as b is a committed block at ℓ, the level of the head of p's blockchain is at least $\ell + 1$. Then, as p has a block at level $\ell + 1$ in his blockchain, p has observed an endorsement QC for $(\ell + 1, r, h, u)$ for some value u and some round r, where h is the hash of block b. Similarly, p' has observed an endorsement QC for $(\ell + 1, r', h', u')$ for some value u' and some round r', where h' is the hash of block b'. As $b \neq b'$, we have that $h \neq h'$, therefore $(u, h) \neq (u', h')$. We assume without loss of generality that $r \leq r'$. Since there are at most f Byzantine processes, and by Lemma 6 correct bakers can only endorse once per round, it follows that at least $f + 1$ correct bakers endorsed (u, h) during round r at level ℓ. Before broadcasting an endorsement for (u, h) at round r (line 120) any correct process sets its *lockedValue* to u and its *lockedRound* to r (line 119), thus $|L^{u,h}_{\ell,r}| \geq f + 1$. By Lemma 8, since $|L^{u,h}_{\ell,r}| \geq f + 1$, we also have $|\mathsf{preendos}(\ell, r'', u'', h'')| \leq f$, for any round $r'' \geq r$, and any value u'' with $(u'', h'') \neq (u, h)$. This means that a correct process cannot endorse some $(u'', h'') \neq (u, h)$ at a round $r'' \geq r$. This in turn means that there cannot be $2f + 1$ endorsements for $(u'', h'') \neq (u, h)$ with round $r'' \geq r$. This contradicts the fact that there is a QC for $(\ell + 1, r', u', h')$. ◀

▶ **Theorem 10.** *Tenderbake satisfies agreement.*

Proof. By contradiction, using Lemma 9. ◀

B.2 Progress

Let *Phases* be the set of labels PROPOSE, PREENDORSE, and ENDORSE. Let $S_p : \mathbb{N}^* \times \mathbb{N}^* \times Phases \to \mathbb{R}$ be the function such that $S_p(\ell, r, phase)$ gives the starting time of the phase *phase* of round r of process p at level ℓ. We consider that the function S_p returns the real time, not the local time of process p. Note that for different processes p and q, the function S_p and S_q may return different times for the same input, because p and q determine the starting time of their phases based on their local clocks, which may be different before τ.

We say that two correct processes p and p' are *synchronized* if $\ell_p = \ell_{p'}$, $|r_p - r_{p'}| \leq 1$, and $|S_p(\ell_q, r_q, phase_q) - S_{p'}(\ell_q, r_q, phase_q)| \leq 2\rho$, where $q \in \{p, p'\}$ is the process which is "ahead". We say that q is *ahead* of q' (or that q' is *behind* q) if $S_q(\ell_q, r_q, phase_q) \leq S_{q'}(\ell_q, r_q, phase_q)$. We say that p and q are *synchronized at level ℓ and round r* if p and q are synchronized and $\ell = \ell_p = \ell_q$ and $r = \max(r_p, r_q)$. At the beginning of r one of the processes might be at round $r - 1$. However, for at least $\Delta'(r) - 2\rho$ time, the two processes are at the same round.

Let t be the last time p called getNextPhase. We denote by $levelOffset_p = now - levelStart$, where *now* is the value returned by now when called by p at t, and *levelStart* is the sum at line 30. The next lemma states that we can use level offsets to characterize process synchronization. We omit its proof, which follows from an analysis of the synchronize and getNextPhase functions.

▶ **Lemma 11.** *After τ, two correct processes p and q are synchronized iff $|levelOffset_p - levelOffset_q| \leq 2\rho$.*

▶ **Lemma 12.** *Let p and q be two correct processes. If, after τ, they remain at the same level and the head of their blockchain has the same round, then they are eventually synchronized.*

Proof. Suppose that p and q are both at the same level ℓ and that their heads have the same round. p and q have already decided at $\ell - 1$. From the agreement property, p and q agree on the output value at level $\ell - 1$, thus they agree on all blocks up to level $\ell - 2$, and on their rounds as well. Thus, the block rounds in p's and q's blockchain are respectively the same.

Next, both p and q eventually call synchronize and getNextPhase. The round returned by synchronize is eventually larger than the current round of the process, so the process eventually exits the recursion at line 39 and calls getNextPhase.

Let p be the first to call getNextPhase and let t be the time of the call. Let $t' \geq t$ be the time when q first calls getNextPhase. We first note that *levelStart* in the definition of *levelOffset* is the same for both p and q, at both times t and t'. Let $levelOffset^*_t = t - levelStart$ and $levelOffset^*_{t'} = t' - levelStart$. We consider the values of the variable $levelOffset_p$ at t and t' and denote these by (simply) $levelOffset_p$ and $levelOffset'_p$, respectively. [4] Given the bound on clock skews, $|levelOffset_p - levelOffset^*_t| \leq \rho$ and $|levelOffset_q - levelOffset^*_{t'}| \leq \rho$. By using the inequality $|a-b| \leq |a|+|b|$, we obtain that $|levelOffset_q - levelOffset_p - (t'-t)| \leq 2\rho$, that is, $|levelOffset_q - levelOffset'_p| \leq 2\rho$. By Lemma 11, p and q are synchronized at t'. ◀

▶ **Lemma 13.** *If P is a set of correct processes that are synchronized after τ at a level ℓ and a round r with $\Delta'(r) > \delta + 2\rho$, and a process $p \in P$ sends a message at the beginning of its current phase ph, then this message is received by all processes in P by the end of their phase ph.*

[4] We note that $levelOffset'_p - levelOffset_p = t' - t$, because we assume that a process measures intervals of time precisely.

Proof. Assume that p sends its message m at time $t_p = S_p(\ell, r, ph)$. Consider a process $q \in P$, and let $t_q = S_q(\ell, r, ph)$. Process q receives m at most at time $t_p + \delta$. By the synchronization hypothesis, we have that $t_p - t_q \leq 2\rho$. Then we obtain that $t_p + \delta \leq t_q + 2\rho + \delta < t_q + \Delta'(r)$, as $t_q + \Delta'(r)$ is the time of the end of the phase ph for q. If $t_p < t_q$, then q might receive m while it is still at round $r - 1$. The message m is still available to q at round r because processes keep messages from a round one unit higher than their current round. ◀

▶ **Lemma 14.** *Let ℓ be a level and r a round with $\Delta'(r) > \delta + 2\rho$. Consider that all correct bakers are synchronized at level ℓ and round r at a time after τ. Let p be the proposer at round r. If p is correct and $endorsableRound_p \geq lockedRound_q$ for any correct baker q, then all correct bakers decide at level ℓ at the end of round r.*

Proof. From Lemma 13, we obtain that the Propose message of process p is received by all correct bakers by the beginning of their phase PREENDORSE. Let eR be the value of the endorsable round field of the Propose message. Note that $eR = endorsableRound_p$. We prove next that each correct baker sends the message Preendorse(ℓ, r, h, u), where u, h are the value and the predecessor hash proposed by p. Let q be a correct baker. If q is either unlocked or locked on u, then the condition in line 111 holds, and therefore q sends its preendorsement for (u, h). If q is locked on a value different from u then by hypothesis $lockedRound_q \leq endorsableRound_p$, therefore $lockedRound_q \leq eR$. Also, $endorsableRound_p < r$, since $endorsableRound_p$ is set during the execution of handleEvents before sending the Propose message in round r. Hence, $lockedRound_q \leq eR < r$. If $lockedRound_q = eR$ then, by quorum intersection, $lockedValue_q = u$ thus the first disjunction in line 111 holds for q. If $lockedRound_q < eR < r$ then the second disjunction in line 111 holds for q (note that $r = r_p = r_q$). Thus q sends the corresponding Preendorse message. So, we have proved that all correct bakers broadcast the Preendorse(ℓ, r, h, u) messages (line 112). By Lemma 13 all these Preendorse(ℓ, r, h, u) messages are received by all correct bakers by the beginning of the phase ENDORSE. Thus, for all of them, the condition in line 117 is true, thus all correct bakers broadcast the Endorse message for (u, h) (line 120). In the next phase, for all them, the quorum condition (line 126) holds for (u, h) so they decide (u, h). ◀

▶ **Lemma 15.** *If at some time after τ all correct bakers are synchronized at some level ℓ and round r with $\Delta'(r) > \delta + 2\rho$, then all correct bakers decide at level ℓ by the end of round $r + f + 1$.*

Proof. We first remark that, after τ, thanks to synchrony, a correct baker never skips a round, and in particular never skips its turn when it is time to propose. Let $p_0, p_1, \ldots$ be the sequence of bakers in the order in which they propose starting with round r. That is, p_i is the proposer at round $r + i$, for $i \geq 0$. Let j, k be the indexes of the first and second correct bakers in this sequence. As there are at most f Byzantine processes among $\{p_0, \ldots, p_k\} \setminus \{p_j\}$, we have $j < k \leq f + 1$. We show next that all correct bakers decide by the end of round $r + k$.

Suppose first that p_j is such that $endorsableRound_{p_j} \geq lockedRound_q$, for any correct baker q. By Lemma 14, all correct bakers decide at the end of round $r + j$.

Suppose that there is a correct baker with a locked round higher than $endorsableRound_{p_j}$. Let q be the baker with the highest locked round among all correct bakers. In the round at which p_j proposes, that is, in round $r + j$, q sends a preendorsement QC that justifies its locked round in the PREENDORSE phase (line 114). By Lemma 13, this preendorsement QC is received by all correct bakers, who update in the ENDORSE phase of round $r + j + 1$ their endorsable round to q's locked round at line 97. If between rounds $r + j + 1$ and $r + k - 1$ no correct baker updates its locked round then the proposer p_k will have at round $r + k$ that

$endorsableRound_{p_k} \geq lockedRound_q$, for any correct baker q. By Lemma 14, at the end of round $r+k$ all correct bakers decide. If instead there is a correct baker that updated its locked round before round $r+k$, then let q be the baker which updates it last, at some round $r+j'$ with $j' < k$. When q changes its locked round, q has seen a prendorsement QC for round $r+j'$. This QC is sent together with the Endorse message in the phase ENDORSE, and therefore it will be received by all correct bakers at the beginning of the next phase PROPOSE. Thus every correct baker, including p_k, sets its *endorsableRound* to $r+j'$. Because j' is maximal, no correct baker changes its locked round between rounds $r+j'+1$ and $r+k-1$. Therefore, at round $r+k$, for any baker q, we have that $lockedRound_q \leq r+j' = endorsableRound_{p_k}$. Again, by Lemma 14 we conclude that at the end of round $r+k$ all correct bakers decide. ◀

▶ **Theorem 16.** *Tenderbake satisfies progress.*

Proof. We reason by contradiction. Suppose first there is a level $\ell \geq 1$ such that no correct process decides at ℓ. Clearly, ℓ is minimal with this property. We first show that eventually all correct processes are synchronized. As ℓ is minimal, we have that there is at least one correct process that has decided at $\ell - 1$.

As processes invoke pullChain at regular intervals, all correct process will eventually be at level ℓ (that is, they will have decided at $\ell - 1$). We show next that all correct processes have the same blockchain head. Let p be a correct process that has its $headCertificate_p$ for the block with the lowest round at level $\ell - 1$. Process p eventually receives a pullChain request at some point after τ and it answers. If each correct process q has $endorsableRound_q = 0$ at the time of the receipt of p's answer, then every correct process accepts p's branch, by the definition of betterHead. Suppose however that there is a process q that has $endorsableRound_q > 0$ when it receives p's answer. In this case consider a time when round durations are so big that I and Δ are very small in comparison. More precisely, there is a time period when all pullChain requests and their answers happen during a period when correct processes update their states only in response to a NewChain event, but not in response to NewMessage events. Such a period exists because regular messages are sent only at phase boundaries. This means that the chain ending with the proposal with the highest endorsable round r will be seen by all correct processes, and these processes will have their endorsable round smaller or equal to r. They will update their blockchains to this chain (if they were on a different one). Note that if two processes have the same endorsable round then they also have the same blockchain. We have this obtained that eventually all correct processes have the same blockchain (head). We can therefore apply Lemma 12 to obtain that there is a time after τ at which all correct processes are synchronized.

Now, recall that the function Δ' has the property that there is a round r such that $\Delta'(r) > \delta + 2\rho$. As Δ' is increasing, this property holds for all subsequent rounds as well. And, given that all processes are synchronized from some time on, as proved in the previous paragraph, we obtain that the hypothesis of Lemma 15 is satisfied. Therefore all correct processes decide at ℓ, which contradicts the assumption that no correct process decides at ℓ. In other words, we have proved that, for any level ℓ, there is at least one correct process that decides at ℓ.

Finally, we show that for any level ℓ, any correct process eventually decides at ℓ. Suppose that there is a correct process p that does not decide at some level $\ell \geq 1$. From the first part of the proof we obtain that there is at least one other correct process q that eventually decides at ℓ. Process q will eventually receive p's pull request, will reply, and p will therefore receive an endorsement QC for level ℓ which enables it to decide at ℓ. This contradicts the assumption, and allows us to conclude. ◀