Byzantine Reliable Broadcast and Tendermint Consensus with trusted components

Yackolley Amoussou-Guenou[a], Lionel Beltrando[b], Maurice Herlihy[c], Maria Potop-Butucaru[b]

[a] *Université Paris-Panthéon-Assas, CRED, F-75005 Paris, France*
[b] *Sorbonne Université, CNRS, LIP6, F-75005 Paris, France*
[c] *Brown University Computer Science Dept, Providence RI 02912, USA*

Abstract

Byzantine Reliable Broadcast is one of the most popular communication primitives in distributed systems. Byzantine reliable broadcast ensures that processes agree to deliver a message from an initiator, even if some processes (possibly including the initiator) are Byzantine. In asynchronous settings, it is known since the prominent work of Bracha [1] that Byzantine reliable broadcast can be implemented deterministically if the total number of processes, denoted by n, satisfies $n \geq 3t + 1$ where t is an upper bound on the number of Byzantine processes. Here, we study Byzantine Reliable Broadcast when processes are equipped with *trusted components*, special software or hardware designed to prevent equivocation. Our contribution is threefold. First, we show that, despite common belief, when each process is equipped with a trusted component, Bracha's algorithm still needs $n \geq 3t+1$. Second, we present a novel algorithm that uses a single trusted component (at the initiator) that implements Byzantine Reliable Asynchronous Broadcast with $n \geq 2t + 1$. Lastly, building on our broadcast algorithm, we present TenderTee, a transformation of the Tendermint consensus algorithm by using trusted component, giving better Byzantine resilience. Tendertee works with $n \geq 2t + 1$, where Tendermint needed $n = 3t + 1$.

Keywords: Trusted monotonic counter, Trusted execution environment, Broadcast, Consensus

1. Introduction

Byzantine reliable broadcast and *Consensus* are two fundamental problem in fault-tolerant distributed systems. *Byzantine reliable broadcast* consists of ensuring that a correct initiator process broadcasts its value to all correct processes, even in the presence of malicious Byzantine processes. For decades, Byzantine Reliable Broadcast has been at the core of various consensus protocols, and more recently, at the core of certain blockchains. *Consensus*, introduced by Lamport, Shostak and Pease [2], is one of the fundamental problems in the area of fault-tolerant distributed computing. In the consensus problem, n nodes attempt to reach agreement on a value, despite the malicious behavior of up to t of them. One of the measures of the quality of a consensus protocol is its *resiliency*: the fraction of faulty nodes the protocol can tolerate. Since the proofs rely on a resilience bound of one third for the Byzantine consensus [2] in environments with no authentication, proved later even for models with local authentication [3], research struggled to increase the consensus resilience. The blockchains era revived the interest for the problem.

Byzantine Reliable Broadcast and Consensus have been addressed in various settings: with fixed and mobile Byzantine nodes, dynamicity or in conjunction with transient faults.

Byzantine Reliable Broadcast solutions (e.g. [4, 5, 6, 7, 8]) achieve resilience of at least $n \geq 3t + 1$ processes, where t is the maximum number of Byzantine processes. However, these solutions require strong network assumptions, such as *synchrony* (processes execute in lock-step) or *non-equivocation* (the initiator must send the same message to all processes).

More recently, *trusted execution environments* and more generally *trusted components* have emerged as a promising protection against Byzantine failures by providing cryptographic primitives that protect participants against equivocation. Trusted components are especially promising for consensus protocols such as PBFT, and therefore for many recent blockchain algorithms. For example, [9, 10] recently introduced the *TTCB wormhole*, which supports a PBFT protocol that tolerates up to half of the processes to be Byzantine, well beyond the tolerance of classical systems [11]. Nevertheless, the trusted part of these systems makes practical implementations difficult.

A2M (Attested Append-only Memory) [12] provides a small and easy-to-implement abstraction of a trusted append-only log. Each log has a unique identifier and offers methods to append and read values. A value, once added,

cannot be rewritten. A2M increases the resilience of PBFT by appending each message to the log and sending that attestation along with the message, which increases resiliency to one-half.

We are not the first to suggest that trusted environments similar to A2M can increase resilience for blockchains: see *HotStuff* [13], *Damysus* [14], and TenderTee [15]. However, none of these works focuses on the Byzantine reliable broadcast primitive. The authors of [15] conjecture that plugging A2M hardware into Bracha's protocol might increase its resilience. In this paper, we refute their conjecture by showing that A2M-Bracha has the same resilience as the original in asynchronous settings.

An alternative to A2M is the use of a *monotonic counter* implemented in a tamper-proof module. TrInc [16] is a trusted component that deals with equivocation in large systems by providing a set of monotonic counters, supported by a trusted hardware unit called a *trinket*.

More recently, [17] proposed *USIG* (Unique Sequential Identifier Generator), a service available to each process (and implemented in a tamper-proof module) that assigns each message a unique counter value, and signs that message. The service offers two functions: one that returns a certificate, and one that validates certificates. These certificates are based on a secure counter: the counter value is never duplicated, and successive counter values are successive integers. To the best of our knowledge, this kind of trusted component has never been used to increase the resilience of reliable broadcast. Here, we prove that using a trusted component can implement Byzantine Reliable Broadcast in asynchronous environments with optimal resilience.

Similarly to our work, [18] proposes a reliable broadcast algorithm tolerant with $n \geq 2t + 1$ processes where t is the number of Byzantine faults. Contrarily to us, they use failure detectors. [19] proposes an algorithm similar to ours but which only tolerates $t < n/3$ Byzantine processes, whereas our algorithm tolerates up to $t < n/2$ Byzantine faults.

Our contribution. This paper presents a study of Byzantine Reliable Broadcast and Consensus problems using trusted components. The current paper merges the contributions presented in [15] and [20]. First, we show that, despite popular belief, trusted components cannot improve the resilience of Bracha's algorithm with no modification. Instead, we propose a novel algorithm that uses a single (optimal number) trusted component to implement asynchronous Byzantine reliable broadcast with n processes, $n \geq 2t+1$ where

t is an upper bound on the number of Byzantine processes. Interestingly, this algorithm uses only one simple trusted component that provides a trusted monotonic counter. We abstract the trusted component via a distributed object called *Trusted Monotonic Counter Object*.

Furthermore, continuing the line of research proposed in [13] we enhance Tendermint [21, 22, 23] with a light version of the trusted abstraction attested append-only memory introduced in [12]. The use of this abstraction makes our protocol, TenderTee, immune to equivocation (i.e. behavior of a faulty node when it sends differents faulty messages to different nodes). TenderTee enjoys one half Byzantine resilience for both and repeated consensus.

Parallel to the design of TenderTee [15], [14] introduces *Damysus*, a BFT protocol that uses trusted environments to improve Hotstuff resilience. In this paper, the authors introduce two trusted services *Checker* and *Accumulator* that respectively increase resilience and reduce latency. It should be noted that previous BFT protocols that used trusted hardware require strong assumptions (e.g the use of expander graphs in [13]) or require important modification on the original protocol (like in [14]). Our integration of trusted components involves only minor modifications to the original protocol, and the broadcast of previous messages to ensure coherency, which helps increasing the resilience of the Tendermint protocol.

Differently from the transformer-based approach where the transformations are obtained with an important communication overhead (exponential in the case of [24] and polynomial in the case of [25]), our design uses only a constant overhead[1] with the respect of original solutions while improving both the resilience (from $n/3$ to $n/2$).

Paper organisation. The paper is organized as follows. Section 2 defines the execution model and presents the specification of the Byzantine Reliable Broadcast problem. Section 3 introduces the key component of our Byzantine Reliable Broadcast implementation, the Trusted Monotonic Counter Object. Section 4 discusses the impossibility of improving Bracha's Byzantine Reliable Broadcast resilience even when each process is equipped with a trusted component. Section 5 presents our algorithm for Byzantine Reliable Broadcast using a single Trusted Monotonic Counter Object at the initiator. Section 6 presents TenderTee, a transformation of the Tendermint consensus

[1]For this comparison, we did not take into account the cost of implementing the trusted monotonic counter abstraction nor any transformer.

algorithm using the Trusted Monotonic Counter Object. Finally, we conclude in Section 7.

2. System model and Problems Definition

We consider a set of n asynchronous sequential processes. Up to t processes can be *Byzantine*, meaning they can deviate from the given protocol. The rest are *correct* processes.

Processes communicate by exchanging messages through an asynchronous network. We make the usual assumption that there is a public key infrastructure (PKI) where public keys are distributed, each process has a (universally known) public key and a matching private key, moreover, each message is signed by its creator. Messages are not lost or spuriously generated. Each process can send messages directly to any other process, and each process can identify the sender of every message it receives.

We assume processes have access to a broadcast primitive, broadcast(m), which ensures that the message m is received by every correct process in a finite (but unknown) time. When a process initiates a broadcast instance, we call that process the *initiator*.

Nodes communicate by exchanging messages through a partially synchronous network (as defined in [26]). *Partially Synchronous* means that there is a bound δ that is finite but unknown by the participants on the message transfer delay. We do not consider asynchronous communication systems since it is impossible to solve consensus in asynchronous systems when there is at least one failure [11].

In this paper we address the Byzantine Reliable Broadcast problem and the one-shot Consensus formally defined as follows.

Definition 1 (Byzantine Reliable Broadcast problem [1])**.** *We say that an algorithm implements* Byzantine reliable broadcast *if:*

- *brb-CorrectInit: If the initiator is correct, all correct processes deliver the initiator's value.*
- *brb-ByzantineInit: If the initiator is Byzantine, then either no correct process delivers any value, or all correct processes deliver the same value.*

Definition 2 (Consensus)**.** *We say that an algorithm implements Consensus if and only if it satisfies the following properties:*

- ***Termination.*** *Every correct process eventually decides some value.*
- ***Integrity.*** *No correct process decides twice.*
- ***Agreement.*** *If there is a correct process that decides a value B, then eventually all the correct processes decide B.*
- ***Validity[27].*** *A decided value is valid, it satisfies the predefined predicate denoted* isValid().

3. Trusted Monotonic Counter Object

TEEs in general (e.g. A2M [12], TrInc [16], USIG [17]) are reputed to be powerful tools for avoiding equivocation. Although the trusted component abstraction makes protocols immune to equivocation (where the initiator sends different messages to different processes), [24] shows that non-equivocation is not enough to provide $n \geq 2t + 1$ resilience nor to support the equivalent of digital signatures.

We now define the Trusted Monotonic Counter Oracle abstraction TMC-Object the core of our novel Byzantine Reliable Broadcast protocol that supports t Byzantine failures among n processes, where $n \geq 2t + 1$, an improvement on the classical $n \geq 3t + 1$ algorithms.

In short, TMC-Object provides a non-falsifiable, verifiable, unique, monotonic, and sequential counter. In particular, TMC-Object provides each process with a read-only local variable, called trustedCounter. Whenever the TMC-Object is invoked, it returns a value for trustedCounter that is strictly greater than any previous value it returned. The difference between two successive counter values is exactly 1, so when a process p receives two messages stamped with counter values, it can detect whether there have been intermediate messages.

The TMC-Object supports the operation **get_certificate()**. A process p invokes **get_certificate(m)** with a message m. The object returns a *certificate* and a *unique identifier*. The certificate certifies that the returned identifier was created by the tamper-proof TMC-Object object for the message m. The unique identifier is essentially a reading of the monotonic counter trustedCounter, incremented whenever **get_certificate(m)** is called.

The TMC-Object object guarantees the following properties:

- **Uniqueness:** TMC-Object will never assign the same identifier to two different messages.

- **Sequentiality:** TMC-Object will always assign an identifier that is the successor of the previous one.

To send a message u certified by TMC-Object, a process p first invokes the **get_certificate()** operation of TMC-Object, which creates a certificate $\mathcal{C}_{(p,u)}$ corresponding to the value of the trustedCounter c_p,[2] then the process sends the tuple $(u, \mathcal{C}_{(p,u)}, c_p)$, which can be verified by any other process receiving the message. The verification is done using the **check_certificate()** operation of the TMC-Object, which confirms whether the given certificate was produced for the given message and has given the value of the counter. Each invocation of the **get_certificate()** operation of TMC-Object increments the value of the trustedCounter c_p of process p. We call that sequence of operations TMC-Object-Send u.

When receiving a message $(u, \mathcal{C}_{(p,u)}, c_p)$, a process must check if the certificate $\mathcal{C}_{(p,u)}$ for message u corresponds to the value of the counter c_p. If not, the message is considered invalid and is ignored. If they correspond, the message is said to be valid according to the TMC-Object.

4. Bracha's Byzantine Reliable Broadcast with Trusted Components

In this section, we prove that modifying Bracha's reliable broadcast algorithm [1] by (only) equipping each process with a trusted component (here, TMC-Object) does not change the tolerance threshold of Byzantine processes, which remains 1/3, we do so by using the modular formalism introduced by [28].

To send a message u certified with TMC-Object, a process p first invokes TMC-Object, which creates a certificate $\mathcal{C}_p$ corresponding to the value of the trustedCounter c_p, then the process sends the tuple $(u, \mathcal{C}_p, c_p)$, which can be verified by any other process receiving the message. Each invocation to TMC-Object increments the value of the trustedCounter c_p of process p, and the initial value of the counter is 0. In the following, that sequence of operation is simply called **TMC-Object-Send** u.

In the following, we describe Algorithm 1 which is Bracha's reliable broadcast where each process uses the TMC-Object-Send operation instead of a

[2]When the value of a counter can be verified against a given certificate, we say that the certificate corresponds to that value of the counter, similarly to digital signatures.

Send operation. In more detail, the protocol works in sequential steps. In the broadcast primitive described in Algorithm 1, there are three types of messages used in the protocol: *initial*, *echo*, and *ready*. All these messages are sent using the TMC-Object-Send operation. In the initial step (Step 0) of the protocol, when a process p wants to broadcast a value u, it TMC-Object-Sends an initial message for u ($< initial, u >$) to all other processes, therefore incrementing its trusted counter. Recall that the process initiating the broadcast is called the *initiator*.

In Step 1, upon receiving a valid[3] initial message with value v from the initiator, a process A2M-Sends an echo message for v ($< echo, v >$). An echo message is also sent if instead of receiving the initial message, the process receives enough (here, α) echo messages for the same value from different processes, implying that many processes saw the initiator message. After, and only after the A2M-Send operation, the process moves to Step 2.

In Step 2, each process waits to receive echo messages for the same value, say v, from at least α different processes sent from Step 1. When that is the case, the process TMC-Object-Sends a ready message for the value v ($< ready, v >$). After the send operation, the process moves to Step 3.

Step 3 is similar to Step 2. The process waits for β ready messages, when the β ready messages are received for the same value, say v, the process delivers value v and finishes the instance of broadcast.

The broadcast is successful if all correct processes rb-Deliver the same value. Thus, rb-Broadcast and rb-Deliver provide us with a pair of communication primitives.

Lemma 3. *Consider Algorithm 1 with parameter $n \geq \alpha > t$ and $\alpha \geq n/2+1$ where t is the number of Byzantine processes. If two correct processes TMC-Object-Send $< echo, v >$ and $< echo, u >$ messages, respectively, then $u = v$.*

Proof. The proof will be conducted by contradiction. Assume there exist two correct processes that TMC-Object-Send $< echo, v >$ and $< echo, u >$ messages, respectively, with $u \neq v$. Let q be the first correct process that TMC-Object-Sends an $< echo, v >$ message, and let r be the first correct process that TMC-Object-Sends an $< echo, u >$ message.

- Case 1: Process q receives an initial message $< initial, v >$ and process

[3]Here, valid TMC-Object-message. If the value of the counter is strictly greater than 2, the initiator may have equivocated (having already A2M-sent another message).

Algorithm 1 Reliable Broadcast with a Trusted Environment

```
1: Step 0
2:   if p is the initiator then
3:     TMC-Object-Send < initial, u > to all
4: Step 1
5:   Wait until receipt of
6:     1 valid TMC-Object-message < initial, v > message, or
7:     α < echo, v > messages
8:     for some v
9:   TMC-Object-Send < echo, v > to all
10: Step 2
11:   Wait until receipt of
12:     α < echo, v > messages
13:       (including messages received in Step 1)
14:     for some v
15:   TMC-Object-Send < ready, v > to all
16: Step 3
17:   Wait until receipt of
18:     β < ready, v > messages
19:       (including messages received in Steps 1 and 2)
20:     for some v
21:   rb-Deliver v
```

r receives an initial message $< \mathit{initial}, u >$. If the initiator is correct, then this situation is impossible since a correct initiator sends only one initial value. If the initiator is Byzantine, then either q or r rejects the initial value since the TMC-Object nominal sequence is invalid (the counter associated with one of these values is strictly greater than 1, hence the message is not valid).

- Case 2: Process q must have received $\alpha < \mathit{echo}, v >$ messages, and process r must have received $\alpha < \mathit{echo}, u >$ messages. Notice that a correct process can send only one echo. Since $\alpha > t$, where t is the number of Byzantine processes in the system, among the α messages some come from correct processes. Since $\alpha \geq n/2 + 1$ then there is at least one correct process that TMC-Object-Sent $< \mathit{echo}, v >$ and $< \mathit{echo}, u >$ messages which is impossible since the process is correct.

□

Lemma 4. *Consider Algorithm 1 with parameter $n \geq \alpha > t$ and $\alpha \geq n/2+1$ where t is the number of Byzantine processes. If two correct processes TMC-Object-Send $< \mathit{ready}, v >$ and $< \mathit{ready}, u >$ messages, respectively, then $u = v$.*

Proof. Proof by contradiction. Assume there exist two correct processes which TMC-Object-Send $< \mathit{ready}, v >$ and $< \mathit{ready}, u >$ messages, with $u \neq v$.

Let q be the first process that TMC-Object-Sends a $< \mathit{ready}, v >$ message, and let r be the first process that TMC-Object-Sends a $< \mathit{ready}, u >$ message. Process q must have received more than $\alpha < \mathit{echo}, v >$ messages and process r must have received more than $\alpha < \mathit{echo}, u >$ messages. Since $\alpha > t$ and $\alpha \geq n/2+1$ it follows that at least one correct process must have TMC-Object-Sent $< \mathit{echo}, v >$ and at least one correct process must have TMC-Object-Sent $< \mathit{echo}, u >$ messages. Following Lemma 3, we then have $u = v$. □

Lemma 5. *Consider Algorithm 1 with parameter $n \geq \alpha > t$ and $\alpha \geq n/2+1$ where t is the number of Byzantine processes. If two correct processes, q and r, deliver the values v and u, respectively, then $u = v$.*

Proof. If q delivers the value v then it must have received $\alpha < \mathit{ready}, v >$ messages, and therefore a $< \mathit{ready}, v >$ message is from at least 1 correct process. Similarly, r must have received a $< \mathit{ready}, u >$ message from at least 1 correct process. By Lemma 4, $u = v$. □

We now show that Algorithm 1 satisfies the property brb-CorrectInit of the Byzantine reliable broadcast.

Theorem 6. *Consider Algorithm 1 with parameters α and β. If the initiator is a correct process Algorithm 1 satisfies the brb-CorrectInit property if $\alpha = \beta$ and $n/2 + 1 \leq \alpha \leq n$ and $n \geq 2t + 1$ where t is the number of Byzantine processes and n is the total number of processes.*

Proof. The proof follows directly from Lemmas 3, 4 and 5. Let p be the initiator. Since the initiator is correct, the value broadcast by p, u, will eventually be received by all other correct processes (at least $n - t = t + 1$ correct processes). These processes will echo that value u (TMC-Object-Send an echo message for u). Since all correct processes echo the same value (Lemma 3), and their number is sufficient to make the protocol advance (there are at least $n/2+1$ correct processes), each correct process will receive enough echoes to send a ready message, and the same one (Lemma 4). By the same argument and applying Lemma 5, all correct processes will receive enough ready messages for the initiator value u, and then will rb-Deliver the initiator message. □

Unfortunately, the following result shows that in the presence of a Byzantine initiator, Algorithm 1 could produce undesirable behaviour, and hence does not implement the Byzantine reliable broadcast.

Lemma 7. *Let n be the number of processes, and t be an upper bound on the Byzantine processes with $n \geq 2t + 1$. Consider Algorithm 1 with parameters α and β with $\alpha = t + 1$ and $t + 1 \leq \beta < 2t + 1$ Algorithm 1 does not satisfy the* brb-ByzantineInit *property when the initiator is Byzantine.*

Proof. If the initiator is a Byzantine process, Byzantine processes could force a subset of correct processes to deliver a value, and another subset of correct processes to never deliver any value. Note that even though all processes use a TMC-Object abstraction such that Byzantine processes cannot equivocate, Byzantine processes still can send a message to some processes but not to others.

Let p be the Byzantine initiator. p TMC-Object-sends a value u to $1 \leq x \leq t$ correct processes $q_1, q_2 \ldots q_x$ but not to the other $n-t-x$ processes. Denote by Q this set of x correct processes receiving the initiator's initial message. Since processes in Q receive the message from the Byzantine initiator, they TMC-Object-Send an echo message for u. Assume now that all the Byzantine processes TMC-Object-Send an echo message for value u only to processes in Q but not to the others. It follows that all correct processes but those in Q have no message from the initiator and only the $< echo, u >$ from processes in Q. Those processes cannot advance past Line 5 of Algorithm 1 since they need at least $\alpha = t + 1 > x$ echo messages.

All correct processes but those in Q have only x echo messages that come from the processes in Q. Processes in Q on the other hand would have the echo messages from all Byzantine processes in addition to their own echo messages, which sums to $t + x$ echo messages for the value u. Therefore, processes in Q will advance and TMC-Object-Send a ready message for u.

In the same spirit, Byzantine processes can TMC-Object-Send a ready message for value u to processes in Q only. The other correct processes are still blocked at 5 of Algorithm 1. In addition to the ready messages from the Byzantine processes, processes in Q also get their own ready messages for value u, so each process $q \in Q$ has a total of $t + x$ ready messages and hence delivers the value u (rb-Delivery). The other correct processes only receive the values TMC-Object-Sent by processes in Q, meaning x echo message and x ready message, both for u, hence, they can never reach an acceptance decision in Algorithm 1. It follows that Algorithm 1 does not satisfy the brb-ByzantineInit property when the initiator is Byzantine. □

When $\alpha = \beta = t + 1$, Lemma 7 violates the brb-ByzantineInit property of the Byzantine reliable broadcast (Definition 1), and, hence, does not satisfy

the Byzantine reliable broadcast (Definition 1) as stated by the following Corollary.

Corollary 8. *Let n be the number of processes, and t an upper bound of the Byzantine processes. If $n \geq 2t + 1$, Algorithm 1 does not implement the Byzantine reliable broadcast.*

However, for Algorithm 1 to implement the Byzantine reliable broadcast, we show in Theorem 9 that we must have $\beta = 2t + 1$.

Theorem 9. *Necessary conditions for 1 with parameters α and β to implement the Byzantine reliable broadcast are $\alpha = t+1$, $\beta = 2t+1$, and $n-t \geq \beta$, where t is the upper bound of the number of Byzantine processes and n is the total number of processes.*

Proof. If the initiator is correct, all correct processes decide to deliver the initiator message, by Theorem 6.

It remains to show that when the initiator is Byzantine, either no correct process delivers any value or all correct processes deliver the same value.

- By Lemma 5, if two correct processes deliver a value, they must deliver the same one.

- Now, let us turn to the case where only one correct process reaches a decision.

  Assume that process q reaches a decision and delivers a value u. It means that q received at least β ready messages, from which at least $\beta - t$ are from correct processes.

  At least $\beta - t$ correct processes sent a ready message. However, to TMC-Object-Send a ready message, a correct process must have reached Step 2, and must have completed Step 1 of Algorithm 1. In fact, if a correct process does not TMC-Object-Send an echo message, it cannot enter Step 2. Therefore, we know that at least $\beta - t$ correct processes have TMC-Object-Sent an echo message. By Lemma 3, all correct processes that TMC-Object-Sent an echo message have TMC-Object-Sent it for the same value, hence it means that they all TMC-Object-Sent an echo message for the value u.

  We would like to have that, with at least $\beta - t$ correct processes TMC-Object-Sending an echo message for the same value, say a value u, all

correct processes must have received at least α echo messages for value u. Hence, we have that $\beta - t \geq \alpha \implies \beta \geq \alpha + t \geq 2t + 1$. For lower bounds, now assume that $\alpha = t + 1$ and $\beta = 2t + 1$. The rest of the proof shows that it is sufficient for Algorithm 1 to implement the Byzantine reliable broadcast.

Hence they will all TMC-Object-Send an echo message for u. In that case, all correct processes (at least $2t + 1$ processes) TMC-Object-Sent an echo for u, all correct processes (at least $2t + 1$) will then TMC-Object-Send a ready message for u, which leads to all correct processes eventually delivering value u. Hence, if one correct process delivers a value u, all other correct processes eventually deliver the same value u.

□

5. Byzantine Reliable Broadcast with optimal TMC-Object

Algorithm 2 Byzantine Reliable Broadcast with a unique Trusted Environment for the initiator — brb-Broadcast(u)

```
1: Step 0
2:   if p is the initiator then
3:     TMC-Object-Send < initial, u, id_initiator > to all
4: Step 1
5:   Wait until receipt of
6:     1 valid TMC-Objectmessage < initial, v, id_initiator > message
7:     for some v
8:   Send < echo_in, v, (< initial, v, id_initiator >, C_initiator, c_initiator) > to all   //the process broadcasts
     back the initiator's message with the associated certificate and trusted counter.
9: Step 2
10:   Wait until receipt of
11:     t + 1 < echo_in, v > messages     //Projection of the echoes received keeping only the value of the
      message.
12:       (including messages received in Step 1)
13:     for some v
14:   Send < ready, v > to all
15: Step 3
16:   Wait until receipt of
17:     t + 1 < ready, v > messages
18:       (including messages received in Steps 1 and 2)
19:     for some v
20:   brb-Deliver v
```

In this section, we present Algorithm 2, which contains a small modification of Bracha's algorithm [1]. Algorithm 2 solves the reliable broadcast problem tolerating $t < n/2$ Byzantine processes. Algorithm 2 uses the

trusted component setups to increase the security threshold of Byzantine reliable broadcast from 1/3 of Byzantine processes to 1/2.

Moreover, to reduce the use of the trusted component, which can be resource-intensive, only the initiator is required to send certified messages. The other processes simply check the validity of the message and its certification but do not require the equipment to send certified messages. In this section, we consider the use of TMC-Object defined in Section 3.

In Algorithm 2, only the initiator sends a message using the light TMC-Object abstraction for its send operation. We say that the initiator TMC-Object-Sends a message. The other processes send their message classically. The other difference with Algorithm 1 is that during Step 1, when a process receives the initiator's message, say for value u, it sends an echo message for u coupled with the initiator message (meaning the message, and the certificate and counter sent by the initiator). In such a way, it ensures that all other processes will eventually receive the initiator's certified message. The rest of the algorithm proceeds as in Algorithm 1 where the TMC-Object-Sends are replaced by classical Send operations).

The broadcast is successful if all the correct processes brb-Deliver the same value, say u. Thus, brb-Broadcast and brb-Deliver provide a pair of communication primitives resilient to $t < n/2$ Byzantine processes.

We can now prove the correctness of Algorithm 2 against the Byzantine reliable broadcast abstraction.

Recall that we say that a process *accept*s a message if it receives and adds the "valid" messages in terms of the validity of the TMC-Object, meaning that there was no message before in that same category, hence the value of the trusted counter is the lowest. Notice that if the message received is not expected to be a TMC-Object message, and is indeed not part of a TMC-Object operation (e.g., a send which is not TMC-Object-Send), such a message is valid by default and, therefore, is accepted.

Finally, for any value v, the message $< echo\_in, v, (< initial, v, id\_initiator > , \mathcal{C}_{\text{initiator}}, c_{\text{initiator}}) >$ should be understood as two messages bundled together, i.e., the echo message $< echo, v >$ sent after the reception of the initiator message and sending back the initiator's message $< initial, v, id\_initiator >$, along with the certificates and trusted counter, i.e., $\mathcal{C}_{\text{initiator}}, c_{\text{initiator}}$. The message is exactly the initiator's message, the TMC-Object message will be correctly validated.

Lemma 10. *In any execution of Algorithm 2, with $n \geq 2t + 1$ where t is the*

number of Byzantine processes, a correct process sends each type of message (initial, echo_in, ready) at most once.

Proof. In Steps 1 and 2 of Algorithm 2, the protocol requires sending exactly 1 message, and then moving to the subsequent phase. A correct process cannot send more messages.

In Step 0, a correct process TMC-Object-Sends a message if and only if it is the initiator, and after that moves to Step 1. If the process is not the initiator, it does not (TMC-Object-)Send anything, but moves directly to Step 1. Hence, in Step 0, at most 1 message is sent. □

Lemma 11. *Consider Algorithm 2 with $n \geq 2t + 1$ where t is the number of Byzantine processes. If two correct processes p and q receive and accept respectively $< initial, u, id\_initiator >$ and $< initial, v, id\_initiator >$, then $u = v$.*

Proof. This holds thanks to the properties of the TMC-Object. Since equivocation is not possible at the initiator level, thanks to the use of the counter in TMC-Object if two correct processes p and q accept an initiator message, it means they received the same message, and they then echo the accepted initial message (Line 8 of Algorithm 2). Therefore, they receive and accept the same message. □

By Lemma 11, we know that if two correct processes accept an initiator message, then they accept the same message. The only thing that could happen is for one correct process to receive the initiator message, while another process does not receive such a message.

Lemma 12. *Consider Algorithm 2 with $n \geq 2t + 1$ where t is the number of Byzantine processes. If one correct process sends $< echo\_in, v >$ for some v, then all correct processes will eventually send $< echo\_in, v >$.*

Proof. Let p and q be processes. Without loss of generality, assume that p is the first correct process to do an echo. If a correct process echoes a message, it means it accepts the initiator message, and will send the echo along with the TMC-Object-Send of the initiator (Line 8 of Algorithm 2). Two cases can arise. Either the initiator sent the initial message to both p and q or the initiator did not send a message to q. Notice that it is not possible for the initiator not to have TMC-Object-Sent a message to p, since p echoed the initiator message.

First, consider the case where p received the initiator message, but not q. The process p sent an echo message containing the initiator's initial message respecting the TMC-Object format. By assumption, a message sent by a correct process will eventually be received by all the other correct processes. Therefore, eventually, q will receive p's echo message, containing the initial message. q will be able to assess the validity of the initial message (according to the initiator signatures), will accept it, and will send an echo for the message too. Because before receiving p's echo message, q is still waiting in Step 1.

Finally, if the initiator sends a message to both p and q, therefore, either it sends the same message to p and q, and so q will echo that same message (by Lemmas 10 and 11, it is not possible for q to echo something else), or the message sent to q is invalid and not accepted. That last case is equivalent to the above situation, since an invalid message is not registered nor considered, and is equivalent to not having received a message. □

Thanks to Lemmas 10 and 12, we know that whenever a correct process sends an *echo_in* message, all other correct processes will also echo a message (and more accurately, the same message).

Lemma 13. *If two correct processes p and q send respectively $< ready, v >$ and $< ready, u >$, then $u = v$.*

Proof. By contradiction. Assume $u \neq v$. Without loss of generality, let p be the process that sends a $< ready, v >$ message, and let q be a process that sends a $< ready, u >$ message. To send a ready message for value x, a correct process must have received from at least $t+1$ different processes the message $< echo\_in, x >$ (Line 14 of Algorithm 2). Therefore, p must have received the message $< echo\_in, v >$ from at least $t + 1$ different processes, and process q must have received the message $< echo\_in, u >$ from at least $t + 1$ different processes. Since there are at most t Byzantine processes, at least one correct process must have sent an echo message for both u and v, which is impossible by Lemma 10. Therefore, it is impossible to have $u \neq v$. □

Lemma 14. *Consider Algorithm 2 with $n \geq 2t + 1$ where t is the number of Byzantine processes. If two correct processes, p and q, brb-deliver the values v and u, respectively, then $u = v$.*

Proof. This proof is similar to the proof for 13. We proceed by contradiction. Assume two messages containing respectively values u and value v

such that $u \neq v$. Without loss of generality, let p be the process that brb-delivers the message containing v, and let q be a process that brb-delivers u. To brb-deliver a value x, a correct process must have received from at least $t+1$ different processes the message $< \mathit{ready}, x >$ (Line 17 of Algorithm 2). Therefore, p must have received the message $< \mathit{ready}, v >$ from at least $t+1$ different processes, and process q must have received the message $< \mathit{ready}, u >$ from at least $t+1$ different processes. Since there are at most t Byzantine processes, it means that at least one correct process sent ready messages for both values u and v, which is impossible by Lemma 10. Therefore, it is impossible to have $u \neq v$. □

Lemma 15. *Consider Algorithm 2 with $n \geq 2t+1$ where t is the number of Byzantine processes. If a correct process p delivers the value v then every other correct process will eventually deliver v.*

Proof. If p brb-Deliver v then p received the message $< \mathit{ready}, v >$ from at least $t+1$ different processes. Since there are at most t Byzantine processes, it means that at least one correct process sent a message $< \mathit{ready}, v >$. Since one correct process sent a ready message for v, it means that it received an $< \mathit{echo}, v >$ message from at least $t+1$ different processes; hence, (since there are at most t Byzantine processes) it means that at least one correct process sent a message $< \mathit{echo}, v >$. Therefore, by Lemma 12, all other correct processes will (eventually) send an echo message for value v. Those will be received by all the correct processes. This will lead to having at least $t+1$ different processes sending it. All correct processes will, therefore, eventually send a ready message for value v (by Lemma 13, since we already know that one correct sent a ready for v). Hence, at least $t+1$ ready messages will be received by all correct processes, which will lead them to brb-Deliver v. □

Lemma 16. *Consider Algorithm 2 with $n \geq 2t+1$ where t is the number of Byzantine processes. If a correct process p broadcasts v then all correct processes brb-deliver v.*

Proof. The proof of the lemma is straightforward. If a correct process broadcasts an initial message, it does so to all processes. All processes in Step 1 will echo_in the initiator message containing value v, thanks to Lemma 12. Since correct processes are the majority, and the network is eventually synchronous, they will all eventually receive at least $t+1$ echo_in message for value v and send each a ready message for value v. Thanks to Lemma 13,

since one correct process sends a ready for v, v is the only value correct process will send a ready for. That value will, therefore, be present in at least $t + 1$ ready messages, hence in Step 3, a correct process will brb-deliver v. By Lemma 15, all correct processes will eventually brb-deliver v. □

We can now prove that the algorithm implements the Byzantine reliable broadcast.

Theorem 17. *Let n be the number of processes, and t an upper bound of the Byzantine processes. If $n \geq 2t + 1$, Algorithm 2 implements Byzantine reliable broadcast.*

Proof. By Lemma 16, when a correct initiator broadcasts a value, all correct processes brb-deliver that value.

By Lemma 12, if a correct process brb-delivers an initiator message (even if the initiator is Byzantine), all correct processes will eventually brb-deliver the same initiator message. In that case, the rest of the proof follows thanks to Lemma 16.

Otherwise, no correct process brb-delivers any value. In more detail, if a Byzantine initiator does not send an initial message to any correct process, no correct process will deliver anything. That is because no correct process will send an echo message (then none will send ready messages). Since all advances require $t + 1$ messages from different processes, and Byzantine processes are at most t, the correct processes will be stuck in Step 1 and will make no decision. □

Theorem 18. *Let n be the number of processes, and t be an upper bound of the Byzantine processes. If $n \geq 2t + 1$, in Algorithm 2, the number of TMC-Object used is optimal, in the sense that if we remove the only TMC-Object (initiator), Algorithm 2 does not implement the Byzantine reliable broadcast.*

Proof. In Algorithm 2, only 1 TMC-Object is used, the one at the initiator. If the TMC-Object is removed (instead of doing a TMC-Object-Send, the initiator does only a Send operation), then the algorithm resembles the Bracha's Byzantine reliable broadcast protocol [1] where instead of a $2t + 1$ bound to advance, we have only a $t + 1$ bound. Since Bracha's Byzantine reliable broadcast protocol is optimal in the number of faults [1, 28], Algorithm 2 with no TMC-Object cannot implement Byzantine reliable broadcast. □

6. TenderTee consensus algorithm

In this section we present an updated version of TenderTee (initially introduced in [15]), a variant of the Tendermint consensus [23, 29] in an eventually synchronous model in presence of Byzantine faulty nodes. We integrate our TMC-Object to the Tendermint consensus algorithm to increase its resilience. Tendermint has been created in the context of the newly emerged blockchain technology. A blockchain is a distributed ledger that mimics the functioning of a classical traditional ledger (i.e. transparency and falsification-proof of documentation) in an untrusted environment where the computation is distributed. In blockchain systems, nodes (a.k.a miners) maintain a replica of a continuously-growing list of ordered blocks that include one or more transactions that have been verified by the members of the system. Blocks are linked using cryptography and the order and the content of newly-added blocks is the outcome of a distributed agreement algorithm among the nodes.

First examples of blockchains (Bitcoin [30] and Ethereum [31]) use the Proof-of-Work paradigm. That is, nodes have to solve a cryptographical puzzle in order to be allowed to produce a new block. The difficulty of this puzzle is high enough such as, with high probability, only one block is generated at a specific time. Once produced, the new block is diffused in the network and each correct node adds the newly produced block to its local ledger. Proof-of-Work blockchains have two main drawbacks. Firstly, they present a huge electrical consumption. Secondly, they potentially allow the creation of forks which can be a major issue for using blockchain in industrial applications requiring strong consistency.

These problems motivated the emergence of new blockchains (e.g. Solidus [32], Byzcoin [33], PeerCensus [34], Hyperledger [35], RedBelly [27], Tendermint [29, 22, 23], Hotstuff [36], Tenderbake [37], etc) using the consensus paradigm, a necessary building block in order to ensure blocks linearizability. The repeated production of blocks in committee-based blockchains can be viewed as a repeated consensus problem. At each height of the blockchain exactly one block is decided and added via a one-shot consensus specified below. The traditional specification has been modified using the validity borrowed from [27] in order to meet the requirements for blockchain systems. That is, a new block is added to the blockchain only if it does not contain transactions that conflict with existing transactions in the blockchain. In the following we propose and prove correct one-shot TenderTee protocol that improve the resilience of Tendermint protocol proposed in [22] by using the

TMC-Object abstraction.

6.1. Detailed description

Algorithms 3 and 4 proceed in 3 rounds for a given epoch e at height h. Each protocol message (pre-propose, propose or vote) is sending through TMC-Broadcast (Algorithm 5) which broadcasts the message value to the other processes in a certified manner and increases the corresponding counter. We use the notation TMC-Broadcast($TYPE$, content) to mean that the certified broadcast is done on the counter for the messages of type $TYPE$.

The epoch e_i is initially set to 0.

Counters: Each process has three certified counters, one for each round type. Counters are initially set to 0.

- Round PRE-PROPOSE: Firstly, nodes verify if a decision has been taken in a previous round, if it is the case they only broadcast the Votes that helped them reach a decision. If the validator p_i is the proposer of the epoch, it pre-proposes its proposal value, otherwise, it waits for the proposal from the proposer. If a validator p_j delivers the pre-proposal from the proposer of the epoch, p_j checks the validity of the pre-proposal and that certified attestation is valid and if both conditions are verified, it accepts it with respect to the values in $validValue_i$, $lockedValue_i$, $validEpoch_j$ and $lockedEpoch_i$. If the pre-proposal is accepted and valid, p_j sets its proposal $proposal_j$ to the pre-proposal, otherwise it sets it to nil.

- Round PROPOSE: During the PROPOSE round, each validator broadcasts (through TMC-Broadcast primitive) its proposal, and collects the proposals sent by other validators (only the one that have a valid attestation). After the Delivery phase, validator p_i has a set of proposals, and checks if v, pre-proposed by the proposer, was proposed by at least $t + 1$ different validators. If it is the case, and the value is valid, then p_i sets $vote_i$, $validValue_i$ and $lockedValue_i$ to v and updates $lockedEpoch_i$ to the current epoch e_i, otherwise it sets $vote_i$ to nil.

- Round VOTE: In the round VOTE, a correct validator p_i votes $vote_i$ and broadcasts (using TMC-Broadcast primitive) all the proposals it delivered during the current epoch. Then p_i collects all the messages that were broadcast. First p_i checks if it has delivered at least $t + 1$ proposals for a value v' pre-proposed by the proposer of the epoch, in

that case, it sets $validValue_i$ to that value then it checks if a value v' pre-proposed by the proposer of the current epoch is valid and has at least $t+1$ votes, if it is the case, then p_i decides v' and goes to the next height; otherwise it increases the epoch number and updates the value of $proposal_i$, with respect to $validValue_i$.

Algorithm 3 : Partial Synchronous TenderTee for height h executed by i: Round Pre-Propose

```
1:        /* There are 3 counters for process i, one for each round: Prepropose, Propose, and Vote */
2:  Initialisation:
3:    e_i := 0                                                          /* Current epoch number */
4:    decision_i := nil                                                 /* Store the decision of the process i */
5:    lockedValue_i := nil; validValue_i := nil
6:    lockedEpoch_i := −1; validEpoch_i := −1
7:    proposal_i := getValue()                                          /* Store the value the process will (pre-)propose */
8:    v_i := nil                                                        /* Local variable stocking the pre-proposal if delivered */
9:    validEpoch_j := nil                                               /* Local variable stocking the proposer's validEpoch */
10:   vote_i := nil                                                     /* Store the value the process will vote for */
11:   timeoutPrePropose := Δ_Pre-propose; timeoutPropose := Δ_Propose; timeoutVote := Δ_Vote

12: Round PRE-PROPOSE:
13:   Send phase:
14:     if decision_i ≠ nil then
15:       ∀v, j : (⟨VOTE, e_i, v⟩, j) ∈ messagesDelivered_i, Broadcast(⟨VOTE, e_i, v, C_j, c_j⟩, j)  /* Send back
            all the VOTE messages (as well as the corresponding certification (attestation) C_j and
            counter c_j) the process received that made it decides. */
16:       return
17:     if proposer(e_i) = i then
18:       TMC-Broadcast(PRE − PROPOSE, e_i, proposal_i, validEpoch_i, voteMessagesDelivered_i(e_i))
          /* This broadcast gives an certified attestation, and increment the counter of the pre-propose,
          such that other processes can know if there were other pre-proposal.  */
19:   Delivery phase:
20:     set timerPrePropose to timeoutPrePropose
21:     while (timerPrePropose not expired) ∧ ¬(∃v_j, e_j : sentByProposer(e_i, v_j, e_j)) do
22:       if ∃v_j, e_j : sentByProposer(e_i, v_j, e_j)) then
23:         v_i ← v_j                                                   /* v_j is the value sent by the proposer */
24:         validEpoch_j ← e_j                                          /* e_j is the validEpoch sent by the proposer */
25:     if ¬(∃v, epochProp : sentByProposer(e_i, v, epochProp)) then
26:       timeoutPrePropose ← timeoutPrePropose + 1
27:   Compute phase:
28:     if
          t+1(⟨PROPOSE, validEpoch_j, v_i⟩) ∧ validEpoch_j ≥ lockedEpoch_i ∧ validEpoch_j < e_i ∧ isValid(v_i)
        then
29:       proposal_i ← v_i
30:     else
31:       if ¬isValid(v_i) ∨ (lockedEpoch_i > validEpoch_j ∧ lockedValue_i ≠ v_i) then
32:         proposal_i ← nil                                            /* Note that isValid(nil) is set to false */
33:       if isValid(v_i) ∧ (lockedEpoch_i = −1 ∨ lockedValue_i = v_i) then
34:         proposal_i ← v_i
```

Algorithm 4 : Partial Synchronous TenderTee for height h executed by i: Rounds Propose and Vote

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

21: Round VOTE:
22:   Send phase:
23:     if vote_i ≠ nil then
24:       TMC-Broadcast(VOTE, e_i, logVote_i, vote_i, validEpoch_i, proposeMessagesDelivered_i(e_i))
25:     else
26:       TMC-Broadcast(VOTE, e_i, logVote_i, HeartBeat, validEpoch_i, proposeMessagesDelivered_i(e_i))
27:   Delivery phase:
28:     set timerVote to timeoutVote
29:     while (timerVote not expires) ∧ ¬t+1(⟨(HeartBeat, VOTE)|VOTE, e_i⟩) do{}
30:     if ¬t+1(⟨(HeartBeat, VOTE)|VOTE, e_i⟩) then
31:       timeoutVote ← timeoutVote + 1
32:   Compute phase:
33:     if ∃v'' : t+1(⟨PROPOSE, e_i, v''⟩) ∧ isValid(v'') ∧ sentByProposer(e_i, v'') then
34:       validValue_i ← v''
35:       validEpoch_i ← e_i
36:     if ∃v_d, e_d : t+1(⟨VOTE, e_d, v_d⟩) ∧ isValid(v_d) ∧ decision_i = nil then
37:       decision_i ← v_d
38:     else
39:       e_i ← e_i + 1
40:       v_i ← nil
41:       if validValue_i ≠ nil then
42:         proposal_i ← validValue_i
43:       else
44:         proposal_i ← getValue()
```

Broadcast: When a process TMC-Broadcasts a value, it broadcasts in the same message all the messages received at the previous step as well as its vote message from the previous epochs (and *nil* when in epoch 0), proving that his message is causally related to the previous messages, notice that theses messages are also accompanied by the corresponding certificates and counters. Therefore, when a process delivers a TMC-Broadcast message, in addition to checking the certificates and counters, the process also checks that the message is causally possible according to the accompanying messages. When a message does not satisfy the causality property, it is kept, but not yet used until it delivers messages allowing to satisfy such property.

We now prove the correctness of TenderTee (Algorithms 3 & 4) in a partially synchronous system. We suppose that $n = 2t + 1$ and that each protocol's message m sent by a node i is sent with an attestation $< att_{m,i}, m >$ produced by TMC-Object.

Lemma 19 (Validity)**.** *In a partially synchronous system, TenderTee satisfies the following property: A decided value satisfies the predefined predicate denoted as* `isValid`().

Proof. The proof directly follows by construction. When a correct process decides (Line 37 of Algorithm 4), it checks before if that value is valid (Line 36 of Algorithm 4). Therefore, a correct process only decides valid values. □

Lemma 20 (Integrity)**.** *In a partially synchronous system, TenderTee satisfies the following property: No correct process decides twice.*

Proof. The proof follows by construction. Before deciding (Lines 36 - 37), a correct process i checks if there is not already a value decided ($decision_i = nil$) for the current height (*i.e.* line 36). If there is already a value decided ($decision_i \neq nil$), there is no decision (Lines 38 - 44). Moreover, a correct process exits the algorithm after it has decided (Line 16 of Algorithm 3). No correct process decides twice. □

Lemma 21. *In a partially synchronous system, TenderTee satisfies the following property: A correct process proposes and votes only once per epoch.*

Proof. We prove this lemma by construction. In Algorithm 4, a correct process proposes (Line 4) and votes only once during the corresponding round (Line 24 or Line 26). At the end of the VOTE round, a process changes epoch

(Line 39). Therefore, it cannot propose nor vote for that epoch any more. □

Lemma 22. *If a correct process accepts a propose (or vote) message from another process, only that message can be accepted by the other correct processes. Somehow, it means that processes can* validly[4] *propose and vote at most once.*

Proof. By Lemma 21, we have that correct processes propose and vote at most once per epoch. Now, we can focus on the Byzantine processes. Assume that a Byzantine process proposes (resp. votes) for two different value. Broadcast should be done with the TMC-Broadcast. If the process does not use the TMC-Object for the broadcast, there will be no certificate, hence the corresponding message will not be considered by correct processes, since it is not valid. If the process uses the TMC-Broadcast for both propose messages (resp. vote messages) v and v', the two messages cannot have the same counter number, however, only one could be valid and accepted by correct processes, all other values cannot be accepted thanks to the TMC-Object guarantees. The Byzantine process can, however, also not propose (resp. vote) any message. □

Lemma 23. *In a partially synchronous system, TenderTee satisfies the following property: At most one value can be proposed by at least $t+1$ processes per epoch, and at most one value can be voted at least $t+1$ times per epoch.*

Proof. We prove this lemma by contradiction. Let v, v' such that $v \neq v'$. Since there are $2t+1$ processes in the system, if v or v' gets at least $t+1$ proposals (resp. votes), it means that at least 1 process proposes (resp. votes) for both v and v' which contradicts Lemma 22. □

Lemma 24. *Let v be a value, and let e an epoch. In a partially synchronous system, TenderTee satisfies the following property: If at least $t+1$ different processes (validly) vote for v during epoch e, then no correct process i will have $lockedValue_i \neq v \wedge lockedEpoch_i \geq e$, at the end of each epoch $e' > e$; moreover, all correct processes who voted v during epoch e can only proposes v or nil for each epoch $e' > e$.*

[4]Validly here means that the vote message can be accepted by correct processes, meaning it satisfies the TMC-Object guarantees.

Proof. Let v be a value, and let e be an epoch, and $L^{v,e} = \{i : i \text{ valid votes for } v \text{ during epoch } e\}$, we assume that $|L^{v,e}| \geq t + 1$. We prove the theorem by induction on epoch number.

- *Initialisation:* At the end of epoch e, by assumption $|L^{v,e}| \geq t + 1$. There is a correct process in that set, say i ($i \in L^{v,e}$). It means that i updates $lockedValue_i$ to v during epoch e, therefore i delivered $t + 1$ proposals for the value v (Lines 13 - 15 of Algorithm 4). By Lemma 23, at most one value can have at least $t + 1$ proposals during epoch e, and since v has at least $t+1$ proposals, no correct process j can update $lockedValue_j$ to a value $v' \neq v$ during epoch e. At the end of e, for any correct process j, $lockedValue_j = v \vee lockedEpoch_j < e$.

- Induction: Let $a \geq 1$, we assume that $\forall i \in L^{v,e}$, $lockedValue_i = v$ at the end of each epoch between e and $e + a$, we also assume that if a value was proposed at least $t + 1$ times during these epochs it was either v or *nil*. We prove that at the end of epoch $e + a + 1$, no correct process j will have $lockedValue_j = v' \wedge lockedEpoch_j = e + a + 1$ with $v' \neq v$.

  Let $i \in L^{v,e}$ such that i delivers a pre-proposal for v, then i will set $proposal_i$ to v; it will propose v since $lockedValue_i = v$ (Lines 28 - 34 of Algorithm 3 & Line 4 of Algorithm 4), in any other case, if i does not deliver a pre-proposal or deliver a pre-proposal for a value $v' \neq v$, it will set $proposal_i$ to *nil* and will propose *nil* (Lines 28 - 34 of Algorithm 3 & Line 4 of Algorithm 4), since $\texttt{isValid}(nil) = \texttt{false}$ and by assumption, there is no $e' \in \{e, \ldots, e + a\}$ where there were at least $t + 1$ proposals for a value $v' \neq v$, and $lockedEpoch_i \geq e$. All processes in $L^{v,e}$ will then propose v, *nil* or not propose at all during epoch $e + a + 1$. By Lemma 21, correct processes only propose once per epoch, at least $t + 1$ processes propose v or *nil*; since messages cannot be forged, the only values that can get at least $t + 1$ proposals for the epoch $e + a + 1$ are v and *nil*. If a correct process j delivers at least $t + 1$ proposals for v, it sets $lockedValue_j$ to v and $lockedEpoch_j$ to $e + a + 1$ (Lines 13 - 15 of Algorithm 4); otherwise, it does not change $lockedValue_j$ nor $lockedEpoch_j$ (Line 20 of Algorithm 4). At the end of epoch $e + a + 1$, there is no correct process j such that $lockedValue_j \neq v \wedge lockedEpoch_j = e + a + 1$. Moreover, processes in $L^{v,e}$, only propose v or *nil* during epoch $e + a + 1$.

We proved that if $|L^{v,e}| \geq t+1$, no correct process i will have $lockedValue_i \neq$

$v \wedge lockedEpoch_i \geq e$; moreover a process in $L^{v,e}$ only validly proposes v or *nil* for each epoch $e' > e$. □

Lemma 25 (Agreement)**.** *In a partially synchronous system, TenderTee satisfies the following property: If there is a correct process that decides a value v, then eventually all the correct processes decide v.*

Proof. Let i be a correct process. Without loss of generality, assume that i is the first correct process that decides and assume that it decides value v during epoch e. At time t where i decided, no other node has decided, even those having a bigger epoch number. To decide, i delivered at least $t + 1$ votes for v for epoch e. Since there are less than t Byzantine processes, and since by Lemma 21 correct processes can only vote once per epoch, so at least 1 correct process voted for v during epoch e, so we have $L^{v,e} = \{i : i$ validly votes for v during epoch $e\}$, we assume that $|L^{v,e}| \geq t+1$. By Lemma 24 a process in $L^{v,e}$ only proposes v or *nil* during each epoch after e, and no correct process j will have $lockedValue_i \neq v \wedge lockedEpoch_i \geq e$. Thanks to the broadcast guarantees (Definition 1), all correct processes will eventually deliver the $t + 1$ votes for v from epoch e that made i decides; especially so because when a correct process decides, it sends back all votes it delivered that make it decide (Line 14 of Algorithm 3).

If a correct process j does not decide before delivering these votes, when eventually it delivers them, it will decide v (Lines 36 - 37 of Algorithm 4). Otherwise, it means that j decides before delivering the votes from epoch e.

By contradiction, we assume that j decides a value $v' \neq v$ during an epoch $e' > e$, so j delivered at least $t + 1$ votes for v' during epoch e' (Lines 36 - 37 of Algorithm 4). Since a correct process only votes once by Lemma 21, there are less than t Byzantine processes and the messages are unforgeable, at least 1 correct process voted for v' during epoch e'.

A correct process votes a non-*nil* value if that value was proposed at least $t + 1$ times during the current epoch (Lines 13 - 26 of Algorithm 4). By Lemma 22 processes validly proposes at most once, there are less than t Byzantine processes and the messages are unforgeable, so at least 1 correct process proposed v' during e'. Since $e' > e$ and $|L^{v,e}| \geq t + 1$, by Lemma 24 there are at least 1 correct process that proposed v or *nil* during epoch e'. Even if all the t processes remaining proposes v', there cannot be $t + 1$ proposals for v', which is a contradiction. So, j cannot decide $v' \neq v$ after epoch e since by assumption, e is the first epoch where a correct process decides. □

Lemma 26. *In a partially synchronous system, if there is an epoch after which, when a correct process broadcasts a message during a round, it is delivered by all correct processes during the same round, TenderTee satisfies the following property: If a correct process i updates $lockedValue_i$ to a value v during epoch e, then at the end of the epoch e, all correct processes have $validValue = v$ and $validEpoch = e$.*

Proof. We prove this lemma by construction.

Let e be the epoch after which when a correct process broadcasts a message during a round r, it is delivered by all correct processes during the same round r. Let i be a correct process, we assume that at the end of epoch $e' \geq e$, i has $lockedValue_i = v$ and $lockedEpoch_i = e'$, it means that i delivered at least $t+1$ proposals for v during epoch e' (Lines 13 - 15 of Algorithm 4). Thanks to the reliable broadcast guarantees, and since all messages are propagated, all correct processes will deliver these proposals for v in the worst-case in the VOTE round. Let j be a correct process since j will deliver at least $t + 1$ proposals for v and epoch e' during the VOTE round, it will set $validValue_j = v$ and $validEpoch_j = e'$ (Lines 33 - 35 of Algorithm 4). □

Lemma 27 (Termination)**.** *In a partially synchronous system, TenderTee satisfies the following property: Every correct process eventually decides a value.*

Proof. By construction, if a correct process does not deliver more than $t + 1$ messages (or 1 from the proposer in the PRE-PROPOSE round) from different processes during the corresponding round, it increases the duration of its round. So eventually, during the synchronous period of the system, all the correct processes will deliver the pre-proposals, proposals and votes from correct processes respectively during the PRE-PROPOSE, PROPOSE and the VOTE round; and messages delivered by a correct process will be delivered by the others at most in the following round. Let e be the first epoch after that time.

If a correct process decides before e, by Lemma 25 all correct processes will eventually decide, which ends the proof.

Otherwise, at the beginning of epoch e, no correct process decides yet. Let i be the proposer of epoch e. First, we assume that i is correct and pre-proposes v; v is valid since getValue() always gives a valid value (Line 7 of Algorithm 3 & Line 44 of Algorithm 4), and $validValue_i$ is always valid (Lines 13 & 33 of Algorithm 4). We have two cases:

- Case 1: At the beginning of epoch e, $|\{j : j \text{ correct} \wedge (lockedEpoch_j \leq validEpoch_i \vee lockedValue_j = v)\}| \geq t + 1$.

  Let j be a correct process where the condition $lockedEpoch_j \leq validEpoch_i \vee lockedValue_j = v$ holds. After the delivery of the pre-proposal v from i, j will update $proposal_j$ to v (Lines 28 - 34 of Algorithm 3). During the PROPOSE round, j proposes v (Line 4 of Algorithm 4), since there are at least $t+1$ similar correct processes (included j), they will all propose v, and all correct processes will deliver at least $t + 1$ proposals for v (Line 8 of Algorithm 4).

  Correct processes will set their variable *vote* to v (Lines 13 - 4 of Algorithm 4), then will vote v, and they will deliver all the votes (at least $t + 1$) from this epoch (Lines 24, 26 & 27 of Algorithm 4). Since no correct process has decided yet, and since each of them delivers at least $t + 1$ votes for v, they will decide v (Lines 36 - 37 of Algorithm 4).

- Case 2: At the beginning of epoch e, $|\{j : j \text{ correct} \wedge (lockedEpoch_j \leq validEpoch_i \vee lockedValue_j = v)\}| < t + 1$.

  Let j be a correct process where the condition $lockedEpoch_j > validEpoch_i \wedge lockedValue_j \neq v$ holds. When i will make the pre-proposal, j will set $proposal_j$ to *nil* (Line 32 of Algorithm 3) and will propose *nil* (Line 4 of Algorithm 4).

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

  - No correct process delivers at least $t + 1$ proposals for v during the PROPOSE round, so they will all set their variable *vote* to *nil*, then they will vote *nil* and go to the next epoch without changing their state (Lines 20 & 24 - 27 & 38 - 44 of Algorithm 4).
  - If some correct processes deliver at least $t + 1$ proposals for v during the PROPOSE round, *i.e.*, some Byzantine processes send proposals for v to those processes.

    As in the previous case, they will vote for v, and since there are $t+1$ of them, all correct processes will decide v. Otherwise, there are less than $t+1$ correct processes that deliver at least $t+1$ proposals for v. Only them will vote for v (Line 24 or 26 of Algorithm 4). Without Byzantine processes, there will be less than $t+1$ votes for v, no correct process will decide (Lines 36 - 37 of Algorithm 4) and

they will go to the next epoch; if Byzantine processes send votes for v to a correct process such that it delivers at least $t+1$ votes for v during VOTE round, then the correct process will decide (Lines 36 - 37 of Algorithm 4), and by Lemma 25 all correct processes will eventually decide.

Let k be one of the correct processes that delivers at least $t+1$ proposals for v during the PROPOSE round, it means that $lockedValue_k = v$ and $lockedEpoch_k = e$. It follows by Lemma 26 that at the end of epoch e, all correct processes will have $validValue = v$ and $validEpoch = e$.

If there is no decision, either no correct process changes its state, or all correct processes change their state and have the same *validValue* and *validEpoch*; therefore, eventually, a proposer of an epoch will satisfy Case 1, and that ends the proof.

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

Since proposers are selected in a round-robin fashion, a correct process will eventually be the proposer, and correct processes will decide. □

Theorem 28. *In a partially synchronous system, TenderTee implements the consensus specification.*

Proof. The proof follows directly from Lemmas 19, 20, 25 and 27. By Lemma 19, we show that TenderTee satisfies Validity; by Lemma 20, we show that TenderTee satisfies Integrity; by Lemma 25, we show that TenderTee satisfies Agreement; and by Lemma 27, we show that TenderTee satisfies Termination. □

7. Conclusion

We focus on Byzantine Reliable Broadcast in trusted components. First, we show that adding a TMC-Object to all processes to prevent equivocation does not improve the security threshold or security guarantees of Bracha's

Byzantine Reliable Broadcast, thanks to the formalism introduced in [28]. Second, we propose an optimal trusted component-based algorithm that implements Byzantine Reliable Broadcast in asynchronous settings with resilience $n \geq 2t + 1$. Our algorithm employs a very simple trusted component that provides a trusted monotonic counter.

Furthermore, we present TenderTee, an enhanced version of the Tendermint blockchain. TenderTee uses a lightweight TMC-Object trusted abstraction in order to increase the Byzantine resilience of the original and repeated consensus Tendermint protocols from one-third to one half. By reducing the number of needed nodes this protocol is appealing for industrialisation since the number of nodes to be maintained in order to guarantee agreement is drastically reduced.

References

[1] G. Bracha, Asynchronous byzantine agreement protocols, Inf. Comput. 75 (1987) 130–143. doi:10.1016/0890-5401(87)90054-X.

[2] L. Lamport, R. E. Shostak, M. C. Pease, The byzantine generals problem, ACM Trans. Program. Lang. Syst. 4 (1982) 382–401. URL: http://doi.acm.org/10.1145/357172.357176. doi:10.1145/357172.357176.

[3] M. Borcherding, Levels of authentication in distributed agreement, in: Distributed Algorithms,10th International Workshop, WDAG '96, Bologna, Italy, October 9-11, 1996, 1996, pp. 40–55.

[4] S. Bonomi, J. Decouchant, G. Farina, V. Rahli, S. Tixeuil, Practical byzantine reliable broadcast on partially connected networks, in: 2021 IEEE 41st International Conference on Distributed Computing Systems (ICDCS), IEEE, 2021, pp. 506–516.

[5] S. Bonomi, G. Farina, S. Tixeuil, Reliable broadcast despite mobile byzantine faults, in: A. Bessani, X. Défago, J. Nakamura, K. Wada, Y. Yamauchi (Eds.), 27th International Conference on Principles of Distributed Systems, OPODIS 2023, December 6-8, 2023, Tokyo, Japan, volume 286 of *LIPIcs*, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2023, pp. 18:1–18:23. URL: https://doi.org/10.4230/LIPIcs.OPODIS.2023.18. doi:10.4230/LIPICS.OPODIS.2023.18.

[6] R. Guerraoui, J. Komatovic, P. Kuznetsov, Y.-A. Pignolet, D.-A. Seredinschi, A. Tonkikh, Dynamic byzantine reliable broadcast [technical report], arXiv preprint arXiv:2001.06271 (2020).

[7] A. Maurer, S. Tixeuil, Self-stabilizing byzantine broadcast, in: 2014 IEEE 33rd International Symposium on Reliable Distributed Systems, IEEE, 2014, pp. 152–160.

[8] M. Raynal, Fault-tolerant message-passing distributed systems: an algorithmic approach, springer, 2018.

[9] M. Correia, N. F. Neves, P. Veríssimo, How to tolerate half less one byzantine nodes in practical distributed systems, in: 23rd International Symposium on Reliable Distributed Systems (SRDS 2004), 18-20 October 2004, Florianpolis, Brazil, IEEE Computer Society, 2004, pp. 174–183. URL: https://doi.org/10.1109/RELDIS.2004.1353018. doi:10.1109/RELDIS.2004.1353018.

[10] M. Correia, N. F. Neves, P. Veríssimo, BFT-TO: intrusion tolerance with less replicas, Comput. J. 56 (2013) 693–715. URL: https://doi.org/10.1093/comjnl/bxs148. doi:10.1093/comjnl/bxs148.

[11] M. J. Fischer, N. A. Lynch, M. Paterson, Impossibility of distributed consensus with one faulty process, in: R. Fagin, P. A. Bernstein (Eds.), Proceedings of the Second ACM SIGACT-SIGMOD Symposium on Principles of Database Systems, March 21-23, 1983, Colony Square Hotel, Atlanta, Georgia, USA, ACM, 1983, pp. 1–7. URL: https://doi.org/10.1145/588058.588060. doi:10.1145/588058.588060.

[12] B. Chun, P. Maniatis, S. Shenker, J. Kubiatowicz, Attested append-only memory: making adversaries stick to their word, in: T. C. Bressoud, M. F. Kaashoek (Eds.), Proceedings of the 21st ACM Symposium on Operating Systems Principles 2007, SOSP 2007, Stevenson, Washington, USA, October 14-17, 2007, ACM, 2007, pp. 189–204. URL: https://doi.org/10.1145/1294261.1294280. doi:10.1145/1294261.1294280.

[13] S. Yandamuri, I. Abraham, K. Nayak, M. K. Reiter, Communication-efficient BFT protocols using small trusted hardware to tolerate minority corruption, IACR Cryptol. ePrint Arch. (2021) 184. URL: https://eprint.iacr.org/2021/184.

[14] J. Decouchant, D. Kozhaya, V. Rahli, J. Yu, DAMYSUS: streamlined BFT consensus leveraging trusted components, in: Y. Bromberg, A. Kermarrec, C. Kozyrakis (Eds.), EuroSys '22: Seventeenth European Conference on Computer Systems, Rennes, France, April 5 - 8, 2022, ACM, 2022, pp. 1–16. URL: https://doi.org/10.1145/3492321.3519568. doi:10.1145/3492321.3519568.

[15] L. Beltrando, M. Potop-Butucaru, J. Alfaro, Tendertee: Increasing the resilience of tendermint by using trusted environments, in: 24th International Conference on Distributed Computing and Networking, ICDCN 2023, Kharagpur, India, January 4-7, 2023, ACM, 2023, pp. 90–99. URL: https://doi.org/10.1145/3571306.3571394. doi:10.1145/3571306.3571394.

[16] D. Levin, J. R. Douceur, J. R. Lorch, T. Moscibroda, Trinc: Small trusted hardware for large distributed systems, in: J. Rexford, E. G. Sirer (Eds.), Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2009, April 22-24, 2009, Boston, MA, USA, USENIX Association, 2009, pp. 1–14. URL: http://www.usenix.org/events/nsdi09/tech/full_papers/levin/levin.pdf.

[17] G. S. Veronese, M. Correia, A. N. Bessani, L. C. Lung, P. Veríssimo, Efficient byzantine fault-tolerance, IEEE Trans. Computers 62 (2013) 16–30. URL: https://doi.org/10.1109/TC.2011.221. doi:10.1109/TC.2011.221.

[18] M. Correia, G. S. Veronese, L. C. Lung, Asynchronous byzantine consensus with 2f+1 processes, in: S. Y. Shin, S. Ossowski, M. Schumacher, M. J. Palakal, C. Hung (Eds.), Proceedings of the 2010 ACM Symposium on Applied Computing (SAC), Sierre, Switzerland, March 22-26, 2010, ACM, 2010, pp. 475–480. URL: https://doi.org/10.1145/1774088.1774187. doi:10.1145/1774088.1774187.

[19] R. Wattenhofer, Distributed systems, Lecture notes, https://disco.ethz.ch/courses/hs21/distsys/lnotes/DistSys_Script.pdf, 2022.

[20] Y. Amoussou-Guenou, L. Beltrando, M. Herlihy, M. Potop-Butucaru, Byzantine reliable broadcast with one trusted monotonic counter, in:

T. Masuzawa, Y. Katayama, H. Kakugawa, J. Nakamura, Y. Kim (Eds.), Stabilization, Safety, and Security of Distributed Systems - 26th International Symposium, SSS 2024, Nagoya, Japan, October 20-22, 2024, Proceedings, volume 14931 of *Lecture Notes in Computer Science*, Springer, 2024, pp. 360–374.

[21] J. Kwon, Tendermint: Consensus without mining, `https://tendermint.com/static/docs/tendermint.pdf`, 2014. [Online; accessed 2020 July 10].

[22] Y. Amoussou-Guenou, A. D. Pozzo, M. Potop-Butucaru, S. Tucci Piergiovanni, Correctness of tendermint-core blockchains, in: J. Cao, F. Ellen, L. Rodrigues, B. Ferreira (Eds.), 22nd International Conference on Principles of Distributed Systems, OPODIS 2018, December 17-19, 2018, Hong Kong, China, volume 125 of *LIPIcs*, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018, pp. 16:1–16:16.

[23] Y. Amoussou-Guenou, A. D. Pozzo, M. Potop-Butucaru, S. Tucci Piergiovanni, Dissecting tendermint, in: M. F. Atig, A. A. Schwarzmann (Eds.), Networked Systems - 7th International Conference, NETYS 2019, Marrakech, Morocco, June 19-21, 2019, Revised Selected Papers, volume 11704 of *Lecture Notes in Computer Science*, Springer, 2019, pp. 166–182. URL: `https://doi.org/10.1007/978-3-030-31277-0_11`. doi:`10.1007/978-3-030-31277-0\_11`.

[24] A. Clement, F. Junqueira, A. Kate, R. Rodrigues, On the (limited) power of non-equivocation, in: D. Kowalski, A. Panconesi (Eds.), ACM Symposium on Principles of Distributed Computing, PODC '12, Funchal, Madeira, Portugal, July 16-18, 2012, ACM, 2012, pp. 301–308. URL: `https://doi.org/10.1145/2332432.2332490`. doi:`10.1145/2332432.2332490`.

[25] N. Ben-David, B. Y. Chan, E. Shi, Revisiting the power of non-equivocation in distributed protocols, in: A. Milani, P. Woelfel (Eds.), PODC '22: ACM Symposium on Principles of Distributed Computing, Salerno, Italy, July 25 - 29, 2022, ACM, 2022, pp. 450–459.

[26] C. Dwork, N. A. Lynch, L. J. Stockmeyer, Consensus in the presence of partial synchrony, J. ACM 35 (1988) 288–323. URL: `http://doi.acm.org/10.1145/42282.42283`. doi:`10.1145/42282.42283`.

[27] T. Crain, C. Natoli, V. Gramoli, Red belly: A secure, fair and scalable open blockchain, in: 42nd IEEE Symposium on Security and Privacy, SP 2021, San Francisco, CA, USA, 24-27 May 2021, IEEE, 2021, pp. 466–483.

[28] M. Raynal, On the versatility of bracha's byzantine reliable broadcast algorithm, Parallel Process. Lett. 31 (2021) 2150006:1–2150006:9. doi:10.1142/S0129626421500067.

[29] E. Buchman, J. Kwon, Z. Milosevic, The latest gossip on bft consensus, arXiv preprint arXiv:1807.04938 (2018).

[30] S. Nakamoto, Bitcoin: A Peer-to-Peer Electronic Cash System (2008) 9. URL: https://bitcoin.org/bitcoin.pdf.

[31] V. Buterin, Ethereum whitepaper, ???? URL: https://ethereum.org.

[32] I. Abraham, D. Malkhi, K. Nayak, L. Ren, A. Spiegelman, Solidus: An incentive-compatible cryptocurrency based on permissionless byzantine consensus, CoRR, abs/1612.02916 (2016).

[33] E. Kokoris-Kogias, P. Jovanovic, N. Gailly, I. Khoffi, L. Gasser, B. Ford, Enhancing Bitcoin Security and Performance with Strong Consistency via Collective Signing, in: Proceedings of the 25th USENIX Security Symposium, 2016.

[34] C. Decker, J. Seidel, R. Wattenhofer, Bitcoin Meets Strong Consistency, in: Proceedings of the 17th International Conference on Distributed Computing and Networking Conference (ICDCN), 2016.

[35] E. Androulaki, A. Barger, V. Bortnikov, C. Cachin, K. Christidis, A. D. Caro, D. Enyeart, C. Ferris, G. Laventman, Y. Manevich, S. Muralidharan, C. Murthy, B. Nguyen, M. Sethi, G. Singh, K. Smith, A. Sorniotti, C. Stathakopoulou, M. Vukolic, S. W. Cocco, J. Yellick., Hyperledger fabric: a distributed operating system for permissioned blockchains, in: Proceedings of the Thirteenth EuroSys Conference, EuroSys 2018, Porto, Portugal, April 23-26, 2018, 2018, pp. 30:1–30:15.

[36] I. Abraham, G. Gueta, D. Malkhi, Hot-stuff the linear, optimal-resilience, one-message BFT devil, CoRR abs/1803.05069 (2018). URL: http://arxiv.org/abs/1803.05069. arXiv:1803.05069.

[37] L. Astefanoaei, P. Chambart, A. D. Pozzo, T. Rieutord, S. Tucci Piergiovanni, E. Zalinescu, Tenderbake - A solution to dynamic repeated consensus for blockchains, in: V. Gramoli, M. Sadoghi (Eds.), 4th International Symposium on Foundations and Applications of Blockchain 2021, FAB 2021, May 7, 2021, University of California, Davis, California, USA (Virtual Conference), volume 92 of *OASIcs*, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021, pp. 1:1–1:23.