(Leader/Randomization/Signature)-free Byzantine Consensus for Consortium Blockchains

Tyler Crain[†] Vincent Gramoli[†,‡] Mikel Larrea[†,§] Michel Raynal[⋆,◦]

[†] University of Sydney, Australia
{tyler.crain,vincent.gramoli}@sydney.edu.au
[‡] Data61-CSIRO, Australia
[§] University of the Basque Country UPV/EHU, Spain
mikel.larrea@ehu.eus
[⋆] Institut Universitaire de France
[◦] IRISA, Université de Rennes, France
raynal@irisa.fr

May 5, 2017

Abstract

This paper presents a new resilience optimal Byzantine consensus algorithm targeting consortium blockchains. To this end, it first revisits the consensus validity property by requiring that the decided value satisfies a predefined predicate, which does not systematically exclude a value proposed only by Byzantine processes, thereby generalizing the validity properties found in the literature. Then the paper presents a simple and modular Byzantine consensus algorithm that relies neither on a leader, nor on signatures, nor on randomization. It consists of a reduction of multivalued Byzantine consensus to binary Byzantine consensus satisfying this validity property. This reduction terminates after a constant-sized sequence of binary Byzantine consensus instances. The idea is to spawn concurrent instances of binary consensus but to decide only after a sequence of two of these instances. The binary consensus instances result in a bitmask that the reduction applies to a vector of multivalued proposals to filter out a valid proposed value that is decided. The paper then presents an underlying binary Byzantine consensus algorithm that assumes eventual synchrony to terminate.

Keywords: Asynchronous message-passing system, Binary consensus, Byzantine process, Consensus, Consortium blockchain, Eventual synchrony, Leader-freedom, Modularity, Optimal resilience, Reduction, Signature-freedom.

1 Introduction

Blockchain: a state machine replication paradigm *Blockchain*, as originally coined in the seminal Bitcoin paper [56], is a promising technology to track ownerships of digital assets within a distributed ledger. This technology aims at allowing processes to agree on a series of consecutive blocks of transactions that may invoke contract functions to exchange these assets. While the first instances of these distributed ledgers were accessed by Internet users with no specific permissions, companies have since then successfully deployed other instances in a *consortium* context, restricting the task of deciding blocks to a set of carefully selected institutions with appropriate permissions [11].

In 2016, multiple scientific events devoted to blockchain-related topics outlined the growing interest of the distributed computing community in this technology. These events included DCCL[1], a satellite workshop of ACM PODC 2016 on the topic, and keynote talks presented by C. Cachin [12] and M. Herlihy [31] at major distributed computing conferences. For the distributed computing community, a blockchain may seem like the application of classical state machine replication [39, 62], where processes can be Byzantine [42], to the cryptocurrency context. In the classical state machine replication paradigm, each command (or operation) can be invoked at any time by any process, and be applied to the state machine, regardless of the previously applied commands. The goal in blockchain is for processes to agree on the next block of transactions to be appended to the chain while the goal of a state machine replication is to agree on the next batch of commands to apply to the state machine: both require consecutive consensus instances.

A major distinction between blockchain and state machine replication is, however, the relation between consecutive consensus instances. A blockchain requires each of its consensus instances to be explicitly related to the previous one. More precisely, for a block to be appended it must explicitly contain information pointing to the last block previously appended to the blockchain. This is typically implemented using a collision-resilient hash function that, when applied to the content of a block, outputs a hash identifying this block. To be decided, a block proposed in consensus instance number x must embed the hash of the block decided at instance number $(x-1)$. This is the reason why a blockchain typically starts with processes knowing about a special sentinel block that does not embed any hash, namely the *genesis* block. By contrast, the classical state machine replication simply concatenates consensus instances one after the other without relating the input of a consensus instance to the previous consensus instance: the result of a command may depend on the previous commands, but not the fact that it can be applied. In the terminology used in [32], each command is *total*. While the total order is implementation-defined in classical state machine replication, it is determined (by an application-defined hashing function) in the blockchain.

Which kind of consensus for blockchains? This relation between instances is interesting as it entails a natural mechanism during a consensus instance for discarding fake proposals or, instead, considering that a proposal is *valid* and could potentially be decided. Provided that processes have a copy of the blockchain and the hashing function, they can locally evaluate whether each new block they receive is a valid candidate for a consensus instance: they simply have to re-hash a block and compare the result to the hash embedded in the new proposed block. If the two hashes differ, the new block is considered an invalid proposal and is simply ignored. If the hashes are identical, then the block could potentially be decided. (Whether this block is eventually decided depends on additional well-formedness properties of the block and the execution of the consensus instance.) This validity generalizes common definitions of Byzantine consensus, that either assume that no value proposed only by Byzantine processes can be decided [19, 49, 53], or, in the case where not all non-faulty processes propose the same value, that any value can be decided (i.e., possibly a value proposed by a Byzantine process) [21, 34, 45, 46, 61].

As it is impossible to solve consensus in asynchronous message-passing systems where even a single process may crash (unexpected premature stop) [24], it follows that it is also impossible to solve consensus in a more general model like the one mentioned above. We list below the classic approaches used in the past to circumvent this impossibility, namely failure detectors, conditions, randomization, and eventual synchrony, and describe why we believe that the additional synchrony assumption is the most suited one for blockchains.

- A classical approach in asynchronous crash-prone systems consists in providing processes with information on failures. This is the *failure detector*-based approach [18]. It is shown in [17] that the eventual leader failure detector Ω is the weakest failure detector that allows consensus to be solved in the presence of asynchrony and process crashes. Failure detectors suited to Byzantine failures have been proposed

[1] `https://www.zurich.ibm.com/dccl/`.

(e.g., [26, 34]), but, as they all are defined from the actual failure pattern, they are helpless when Byzantine processes propose valid values. Moreover, when considering eventual leadership, a process may behave correctly from a leader election point of view and behave in a Byzantine way once it has been elected leader. Hence, leader-based algorithms do not seem appropriate to solve agreement issues on valid values in the presence of processes with a Byzantine behavior.

- A second approach, called *condition-based*, consists in restricting the set of possible input vectors [51]. An input vector is a vector where each entry contains the value proposed by the corresponding process. This approach established a strong connection relating Byzantine agreement and error-correcting codes [25]. As the previous eventual leader-based approach and due to the restriction it imposes on input vectors, this approach does not seem suited to blockchain-based applications where processes are not a priori restricted in their proposals.
- A third approach consists in looking for *randomized* consensus algorithms (e.g., [2, 4, 35, 49, 60]). In the context of Byzantine failures, this approach has mainly been investigated for binary consensus algorithms (where the set of values that can be proposed is $\{0, 1\}$). Algorithms that reduce multivalued Byzantine consensus to binary Byzantine consensus have been proposed for both synchronous systems (e.g., [65]) and asynchronous systems (e.g., [53]). Binary randomized algorithms rely on local coins (one coin per process [4]) or a common coin (a coin shared by all processes [60]). When local coins are used, the convergence time to obtain the same value is potentially exponential. When a common coin is used, the expected number of rounds can be a small constant (e.g., 4 rounds in [49]). However, the implementation of a distributed common coin introduces an inherent complexity [15]. Hence, it does not seem appropriate for blockchain repeated consensus.
- The fourth (but first in chronological order) approach to circumvent the asynchronous consensus impossibility in the presence of faulty processes is to enrich the system with an appropriate *synchrony assumption* [20, 21]. The weakest synchrony assumption that allows consensus to be solved is presented in [8]. This last approach (*additional synchrony* assumption) is the one we consider in this paper. As the synchrony assumption is assumed to hold eventually, the consensus algorithm is indulgent to initial arbitrary delays [28]: it always preserves safety, and guarantees liveness once the synchrony assumption holds.

Content of the paper This paper presents a time and resilience optimal Byzantine consensus algorithm suited to *consortium blockchains*. As far as we know the term *consortium blockchain* was initially used in a blog post[2] of the founder of Ethereum [66], Vitalik Buterin, to refer to an intermediate blockchain model between public and fully-private blockchains where only a pre-selected set of processes can participate in the consensus and where the blockchain could potentially be accessed by anyone. The consortium blockchain is generally in contrast with public blockchains, where any Internet users could participate in the consensus algorithm, and fully-private blockchains, where only users of an institution can update the state of the blockchain. Public blockchains like Bitcoin are generally pseudonymous while fully-private blockchains are typically centralized, which makes consortium blockchain an appealing alternative blockchain model for companies. A typical example of consortium blockchains is the testbed ran by R3, a consortium of more than 70 financial institutions around the world.[3] In 2016, R3 led some experiments on an Ethereum consortium blockchain where any institution of the consortium could participate actively in the consensus instance. It is important to note that even within a consortium, one cannot reasonably assume synchronous communications or failures limited to crashes. Typically, the members of the consortium often have conflicting interests—in the R3 consortium example, the banks of the consortium are competitors—and processes cannot control the delay of messages as they typically use Internet to communicate where congestions cannot be avoided.

The Byzantine consensus algorithm proposed in the paper is designed to comply with an extended definition of the consensus validity property. From a structural point of view, it is made of two components.

- The first component is a reduction of multivalued consensus to binary consensus. The reduction, which is fully asynchronous, uses neither randomization, nor an eventual leader, nor signatures. As far as we know, this is the first asynchronous reduction that always decides a non-predetermined value in $O(1)$ sequence of binary consensus. The reduction only waits for the earliest terminating of the concurrent reliable broadcast instances before spawning binary consensus instances. As it assumes $t < n/3$, where n is the number of processes and t is an upper bound on the number of faulty processes, this reduction is

[2] `https://blog.ethereum.org/2015/08/07/on-public-and-private-blockchains/`.
[3] `http://www.r3cev.com/`.

resilience optimal.

- The second component is a binary Byzantine consensus (BBC) algorithm that requires neither randomization, nor an eventual leader, nor signatures. It is based on an appropriate binary-value broadcast (BV-broadcast) abstraction, introduced for randomized consensus [49]. From a computability point of view, the BBC algorithm requires $t < n/3$ (as the previous reduction), and an additional synchrony assumption, namely, there is a time after which the transfer delay of the messages sent by the non-faulty processes is upper bounded by some constant (but neither the time at which this occurs, nor the constant are known by the processes [20, 21]). Practically, this means that the BBC algorithm always terminates except if transfer delays are always increasing (in this case, a different synchrony assumption such as the one described in [50] can be used). The binary Byzantine consensus always terminates in time $O(1)$ if all non-faulty processes propose the same value, otherwise it may still terminate in constant time but is guaranteed to terminate in $O(t)$ time, which is optimal [23].

The resulting multivalued Byzantine consensus algorithm is resilience optimal ($t < n/3$) but also time optimal as it terminates in $O(t)$. In addition to its optimalities and conceptual simplicity, the resulting multivalued Byzantine consensus algorithm is well suited for consortium blockchains for the three following reasons:

1. The algorithm does not use an elected leader (that favors the value proposed by a particular process) or proof-of-work, meaning that every consensus participant plays an equal role in proposing a value. In particular, because it does not rely on the proof-of-work alternative as in Bitcoin or Ethereum, a node of the consortium cannot outweigh other votes during consensus. We already noted that one machine among the 50 machines of the R3 Ethereum consortium in June 2016 owned 12% of the total mining power of the R3 Ethereum consortium blockchain, which gives a significant advantage to this machine to attack the blockchain [57].
2. The algorithm is indulgent [28] in that it is always safe despite arbitrary delays. We believe this is an important property for blockchain applications that trade millions of US$ volume every day[4] as financial institutions may prefer their blockchain service to be unavailable rather than compromised after congestions affect the Internet communication delays. This is typically in contrast with the Ethereum algorithm used as a testbed for the R3 consortium, where an attacker can exploit network delays to double spend by deciding two conflicting blocks [58].
3. Finally, because we focus on the consortium blockchain model where consensus participants are restricted to the members of the consortium, we can assume that the identities of the n consortium members are known by all the participants. Typically only a subset of all blockchain participants participate in the consensus, e.g., only $n = 15$ out of 50 processes of R3 were participating in the consensus [57]. These identities provide a natural protection to our algorithm against Sybil attacks without the need for any costly proof-of-work mechanisms.

Roadmap The paper is structured in 7 sections. Section 2 presents the computation model. Section 3 introduces the Blockchain Byzantine consensus. Section 4 presents a reduction of multivalued Byzantine consensus to binary Byzantine consensus, and Section 5 presents a binary Byzantine consensus that relies on an eventual synchrony assumption. The composition of these two algorithms provides a leader-free, randomization-free and signature-free multivalued Byzantine consensus. Section 6 presents related works. Finally, Section 7 concludes the paper.

2 Basic Byzantine Computation Model and Reliable Broadcast

2.1 Base computation model

Asynchronous processes The system is made up of a set Π of n asynchronous sequential processes, namely $\Pi = \{p_1, \ldots, p_n\}$; i is called the "index" of p_i. "Asynchronous" means that each process proceeds at its own speed, which can vary with time and remains unknown to the other processes. "Sequential" means that a process executes one step at a time. This does not prevent it from executing several threads with an appropriate multiplexing.

As local processing time are negligible with respect to message transfer delays, they are considered as being equal to zero. (We show how to relax this assumption in Appendices B and C.) Both notations $i \in Y$ and $p_i \in Y$ are used to say that p_i belongs to the set Y.

[4] `https://coinmarketcap.com/`.

Communication network The processes communicate by exchanging messages through an asynchronous reliable point-to-point network. "Asynchronous" means that there is no bound on message transfer delays, but these delays are finite. "Reliable" means that the network does not lose, duplicate, modify, or create messages. "Point-to-point" means that any pair of processes is connected by a bidirectional channel. Hence, when a process receives a message, it can identify its sender.

A process p_i sends a message to a process p_j by invoking the primitive "send TAG(m) to p_j", where TAG is the type of the message and m its content. To simplify the presentation, it is assumed that a process can send messages to itself. A process p_i receives a message by executing the primitive "receive()". The macro-operation broadcast TAG(m) is used as a shortcut for "**for each** $p_i \in \Pi$ **do** send TAG(m) to p_j **end for**".

Failure model Up to t processes can exhibit a *Byzantine* behavior [59]. A Byzantine process is a process that behaves arbitrarily: it can crash, fail to send or receive messages, send arbitrary messages, start in an arbitrary state, perform arbitrary state transitions, etc. Moreover, Byzantine processes can collude to "pollute" the computation (e.g., by sending messages with the same content, while they should send messages with distinct content if they were non-faulty). A process that exhibits a Byzantine behavior is called *faulty*. Otherwise, it is *non-faulty*.

Let us notice that, as each pair of processes is connected by a channel, no Byzantine process can impersonate another process. Byzantine processes can control the network by modifying the order in which messages are received, but they cannot postpone forever message receptions.

Notation The acronym $\mathcal{BAMP}_{n,t}[\emptyset]$ is used to denote the previous basic Byzantine Asynchronous Message-Passing computation model; $\emptyset$ means that there is no additional assumption.

2.2 Reliable broadcast in Byzantine systems

Definition This broadcast abstraction (in short, RB-broadcast) was proposed by G. Bracha [9]. It is a one-shot one-to-all communication abstraction, which provides processes with two operations denoted RB_broadcast() and RB_deliver(). When p_i invokes the operation RB_broadcast() (resp., RB_deliver()), we say that it "RB-broadcasts" a message (resp., "RB-delivers" a message). An RB-broadcast instance, where process p_x is the sender, is defined by the following properties.

- RB-Validity. If a non-faulty process RB-delivers a message m from a non-faulty process p_x, then p_x RB-broadcast m.
- RB-Unicity. A non-faulty process RB-delivers at most one message from p_x.
- RB-Termination-1. If p_x is non-faulty and RB-broadcasts a message m, all the non-faulty processes eventually RB-deliver m from p_x.
- RB-Termination-2. If a non-faulty process RB-delivers a message m from p_x (possibly faulty) then all the non-faulty processes eventually RB-deliver the same message m from p_x.

The RB-Validity property relates the output to the input, while RB-Unicity states that there is no message duplication. The termination properties state the cases where processes have to RB-deliver messages. The second of them is what makes the broadcast reliable. It is shown in [10] that $t < n/3$ is an upper bound on t when one has to implement such an abstraction.

Let us remark that it is possible that a value may be RB-delivered by the non-faulty process while its sender is actually Byzantine and has not invoked RB_broadcast(). This may occur for example when the Byzantine sender played at the network level, at which it sent several messages to different subsets of processes, and the RB-delivery predicate of the algorithm implementing the RB-broadcast abstraction is eventually satisfied for one of these messages. When this occurs, by abuse of language, we say that the sender invoked RB-broadcast. This is motivated by the fact that, in this case, a non-faulty process cannot distinguish if the sender is faulty or not.

Notation The basic computing model strengthened with the additional constraint $t < n/3$ is denoted $\mathcal{BAMP}_{n,t}[t < n/3]$.

Algorithms The algorithm described in [9] implements RB-broadcast in $\mathcal{BAMP}_{n,t}[t < n/3]$. Hence, it is t-resilience optimal. This algorithm requires three communication steps to broadcast an application message. An algorithm requiring only two communication steps in the system model $\mathcal{BAMP}_{n,t}[t < n/5]$ is presented in [33].

3 Blockchain Byzantine Consensus

As in all message-passing consensus algorithms, it is assumed (in both the multivalued and binary consensus algorithms presented below) that all non-faulty processes propose a value.

3.1 Multivalued Byzantine consensus with predicate-based validity

In this paper we consider a generalization of the classical Byzantine consensus problem, informally stated in the Introduction. As its validity requirement is motivated by blockchain and relies on an application-specific $\mathsf{valid}()$ predicate to indicate whether a value is *valid*, we call this problem the Validity Predicate-based Byzantine Consensus (denoted VPBC) and define it as follows. Assuming that each non-faulty process proposes a valid value, each of them has to decide on a value in such a way that the following properties are satisfied.[5]

- VPBC-Termination. Every non-faulty process eventually decides on a value.
- VPBC-Agreement. No two non-faulty processes decide on different values.
- VPBC-Validity. A decided value is valid, it satisfies the predefined predicate denoted $\mathsf{valid}()$.

This definition generalizes the classical definition of Byzantine consensus, which does not include the predicate $\mathsf{valid}()$. As an example, in the crash failure model, any proposed value is valid. In the basic Byzantine consensus, any proposed value is valid, except when all non-faulty processes propose the same value v, in which case only v is valid. This predicate is introduced to take into account the distinctive characteristics of consortium blockchains, and possibly other specific Byzantine consensus problems. In the context of consortium blockchains, a proposal is not valid if it does not contain an appropriate hash of the last block added to the Blockchain.

3.2 Binary Byzantine consensus

The implementation of multivalued VPBC relies on an underlying binary Byzantine consensus (denoted BBC). A leader-free, randomization-free and signature-free implementation of it will be described in Section 5.

The validity property of this binary Byzantine consensus is the following: if all non-faulty processes propose the same value, no other value can be decided. To prevent confusion, the validity, agreement and termination properties of BBC are denoted BBC-Validity, BBC-Agreement and BBC-Termination.

4 From Multivalued to Binary Consensus in a Byzantine System

This section describes a reduction of multivalued Byzantine consensus to the previous binary Byzantine consensus. Our reduction is guaranteed to terminate after a sequence of 2 binary consensus instances. This is, as far as we know, the first reduction that decides a non-predetermined value in a sequence of $O(1)$ binary consensus instances. Other reductions either return a predefined $\perp$ value as if the consensus aborted [19, 54, 65], or do not tolerate Byzantine failures and require the execution of $\lceil \log n \rceil$ sequential binary consensus instances [67]. Our reduction is based on the RB-broadcast communication abstraction, and underlying instances of binary Byzantine consensus. Let BBC denote the computational power needed to solve binary Byzantine consensus. Hence, the "multivalued to binary" reduction works in the model $\mathcal{BAMP}_{n,t}[t < n/3, \mathrm{BBC}]$, which is resilience optimal.

4.1 The reduction

Binary consensus objects As just said, in addition to the RB-broadcast abstraction, the processes cooperate with an array of binary Byzantine consensus objects denoted $BIN\_CONS[1..n]$. The instance $BIN\_CONS[k]$ allows the non-faulty processes to find an agreement on the value proposed by p_k. This object is implemented with the binary Byzantine consensus algorithm presented in Section 5.

To simplify the presentation, we consider that a process p_i launches its participation in $BIN\_CONS[k]$ by invoking $BIN\_CONS[k].\mathsf{bin\_propose}(v)$, where $v \in \{0, 1\}$. Then, it executes the corresponding code in a specific thread, which eventually returns the value decided by $BIN\_CONS[k]$.

[5] Note that our consensus definition decouples the validity of a value from the nature (faulty or non-faulty) of the process proposing it. We assume that every non-faulty process proposes a valid value for the sake of simplicity. However, if we assume that the Byzantine behavior of a process is related only to its code and not to its input value (which is application-dependent), our algorithm remains correct as long as at least one non-faulty process proposes a valid value.

Local variables Each process p_i manages the following local variables; $\perp$ denotes a default value that cannot be proposed by a (faulty or non-faulty) process.

- An array $proposals_i[1..n]$ initialized to $[\perp, \cdots, \perp]$. The aim of $proposals_i[j]$ is to contain the value proposed by p_j.
- An array $bin\_decisions_i[1..n]$ initialized to $[\perp, \cdots, \perp]$. The aim of $bin\_decisions_i[k]$ is to contain the value (0 or 1) decided by the binary consensus object $BIN\_CONS[k]$.

```
operation mv_propose(v_i) is
(01) RB_broadcast VAL(v_i);
(02) repeat if (∃ k : (proposals_i[k] ≠ ⊥) ∧ (BIN_CONS[k].bin_propose() not invoked))
(03)          then invoke BIN_CONS[k].bin_propose(1) end if;
(04) until (∃ℓ : bin_decisions_i[ℓ] = 1) end repeat;
(05) for each k such that BIN_CONS[k].bin_propose() not yet invoked
(06)       do invoke BIN_CONS[k].bin_propose(0) end for;
(07) wait_until (⋀_{1≤x≤n} bin_decisions_i[x] ≠ ⊥);
(08) j ← min{x such that bin_decisions_i[x] = 1};
(09) wait_until (proposals_i[j] ≠ ⊥);
(10) return(proposals_i[j]).

(11) when VAL(v) is RB-delivered from p_j do if valid(v) then proposals_i[j] ← v end if.

(12) when BIN_CONS[k].bin_propose() returns a value b do bin_decisions_i[k] ← b.
```

Figure 1: From multivalued to binary Byzantine consensus in $\mathcal{BAMP}_{n,t}[t < n/3, \text{BBC}]$

The algorithm The algorithm reducing multivalued Byzantine consensus to binary Byzantine consensus is described in Figure 1. In this algorithm, a process invokes the operation mv_propose(v), where v is the value it proposes to the multivalued consensus. The behavior of a process p_i can be decomposed into four phases.

- Phase 1: p_i disseminates its value (lines 01 and 11). A process p_i first sends its value to all the processes by invoking the RB-broadcast operation (line 01). When a process RB-delivers the value v RB-broadcast by a process p_j, it stores it in $proposals_i[j]$ if v is valid (line 11).
- Phase 2: p_i starts participating in a first set of binary consensus instances (lines 02-04).
  Then, p_i enters a loop in which it starts participating in the binary consensus instances $BIN\_CONS[k]$, to which it proposes the value 1, associated with each process p_k from which it has RB-delivered the proposed value (lines 02-03). This loop stops as soon as p_i discovers a binary consensus instance $BIN\_CONS[\ell]$ in which 1 was decided (line 04). (The binary consensus we propose later allows to reach the end of phase (2) after only $O(1)$ message delays.)
- Phase 3: p_i starts participating in all other binary consensus instances (lines 05-06).
  After it knows a binary consensus instance decided 1, p_i invokes bin_propose(0) on all the binary consensus instances $BIN\_CONS[k]$ in which it has not yet participated. Let us notice that it is possible that, for some of these instances $BIN\_CONS[k]$, no process has RB-delivered a value from the associated process p_k. The aim of these consensus participations is to ensure that all binary consensus instances eventually terminate.
- Phase 4: p_i decides a value (lines 07-10 and 12).
  Finally p_i considers the first (according to the process index order) among the successful binary consensus objects, i.e., the ones that returned 1 (line 08).[6] Let $BIN\_CONS[j]$ be this binary consensus object. As the associated decided value is 1, at least one non-faulty process proposed 1, which means that it RB-delivered a value from the process p_j (lines 02-03). Let us observe that, due to the RB-Termination-2 property, this value is eventually RB-delivered by every non-faulty process. Consequently, p_i decides it (lines 09-10).

4.2 Correctness proof

Lemma 1. *There is at least one binary consensus instance that decides value* 1*, and all non-faulty processes exit the repeat loop.*

[6] One could replace min by a deterministic function suited for blockchain that returns the value that represents the block with the maximum number of transactions to prevent a Byzantine process from exploiting a lack of fairness to its own benefit.

From an operational point of view, this lemma can be re-stated as follows: there is at least one $\ell \in [1..n]$ such that at each non-faulty process p_i, we eventually have $bin\_decisions_i[\ell] = 1$.

Proof The proof is by contradiction. Let us assume that, at any non-faulty process p_i, no $bin\_decisions_i[\ell]$, $1 \leq \ell \leq n$, is ever set to 1 (line 12). It follows that no non-faulty process exits the "repeat" loop (lines 02-04). As a non-faulty process p_j RB-broadcasts a valid value, it follows from the RB-Termination-1 property, that each non-faulty process p_i RB-delivers the valid proposal of p_j, and consequently we eventually have $proposals_i[j] \neq \perp$ at each non-faulty process p_i (line 11).

It follows from the first sub-predicate of line 02 that all non-faulty processes p_i invokes bin_propose(1). on the BBC object $BIN\_CONS[j]$. Hence, from its BBC-Termination, BBC-Agreement, BBC-Validity, and Intrusion-tolerance properties, this BBC instance returns the value 1 to all non-faulty processes, which exit the "repeat" loop. $\square_{Lemma\ 1}$

Lemma 2. *A decided value is a valid value (i.e., it satisfies the predicate* valid()*).*

Proof Let us first observe that, for a value $proposals_i[j]$ to be decided by a process p_i, we need to have $bin\_decisions_i[j] = 1$ (lines 08-10).

If the value 1 is decided by $BIN\_CONS[j]$, $bin\_decisions_i[j] = 1$ is eventually true at each non-faulty process p_i (line 12). If follows from (i) the fact that the value 1 can be proposed to a BBC instance only at line 03, and (ii) the Intrusion-tolerance property of $BIN\_CONS[j]$, that at least one non-faulty process p_i invoked $BIN\_CONS[j]$.bin_propose(1). Due to the predicate of line 02, this non-faulty process p_i was such that $proposals_i[j] \neq \perp$ when it invoked $BIN\_CONS[j]$.bin_propose(1). Due to line 11, it follows that $proposals_i[j]$ contains a valid value. $\square_{Lemma\ 2}$

Lemma 3. *No two non-faulty processes decide different values.*

Proof Let us consider any two non-faulty processes p_i and p_j, such that p_i decides $proposals_i[k1]$ and p_j decides $proposals_j[k2]$. It follows from line 08 that $k1 = \min\{x$ such that $bin\_decisions_i[x] = 1\}$ and $k2 = \min\{x$ such that $bin\_decisions_j[x] = 1\}$.

On the one hand, it follows from line 07 that $(\bigwedge_{1 \leq x \leq n} bin\_decisions_i[x] \neq \perp)$ and $(\bigwedge_{1 \leq x \leq n} bin\_decisions_j[x] \neq \perp)$, from which we conclude that both p_i and p_j know the binary value decided by each binary consensus instance (line 12). Due to the BBC-Agreement property of each binary consensus instance, we also have $\forall x : bin\_decisions_i[x] = bin\_decisions_j[x]$. Let $dec[x] = bin\_decisions_i[x] = bin\_decisions_j[x]$. It follows then from line 08 that $k1 = k2 = \min\{x$ such that $dec[x] = 1\} = k$. Hence, $dec[k] = 1$.

On the other hand, it follows from the Intrusion-tolerance property of $BIN\_CONS[k]$ that a non-faulty process p_ℓ invoked $BIN\_CONS[k]$.bin_propose(1). As this invocation can be issued only at line 03, we conclude (from the predicate of line 02) that $proposals_\ell[k] = v \neq \perp$. As p_ℓ is non-faulty, it follows from the RB-Unicity and RB-Termination-2 properties that all non-faulty processes RB-delivers v from p_k. Hence, we eventually have $proposals_i[k] = proposals_j[k]$, which concludes the proof of the lemma. $\square_{Lemma\ 3}$

Lemma 4. *Every non-faulty process decides a value.*

Proof It follows from Lemma 1 that there is some p_j such that we eventually have $bin\_decisions_i[j] = 1$ at all non-faulty processes, and no non-faulty process blocks forever at line 04. Hence, all non-faulty processes invoke each binary consensus instance (at line 03 or line 06). Moreover, due to their BBC-Termination property, each of the n binary consensus instances returns a result at each non-faulty process (line 12). It follows that no non-faulty process p_i blocks forever at line 07. Finally, as seen in the proof of Lemma 3, the predicate of line 09 is eventually satisfied at each non-faulty process, which concludes the proof of the lemma. $\square_{Lemma\ 4}$

Theorem 1. *The algorithm described in Figure* 1 *implements multivalued Byzantine consensus (*VPBC*) in the system model* $\mathcal{BAMP}_{n,t}[t < n/3, \text{BBC}]$.

Proof Follows from Lemma 2 (VPBC-Validity), Lemma 3 (VPBC-Agreement), and Lemma 4 (VPBC-Termination). $\square_{Theorem\ 1}$

5 Binary Consensus in Eventually Synchronous Byzantine Systems

This section describes the underlying binary Byzantine consensus algorithm BBC, which provides the processes with the operation bin_propose(). An advantage of this algorithm is that it is guaranteed to terminate if all non-faulty processes propose the same value, even without synchrony, and always in a constant number of message delays. This algorithm may terminate in constant time, this happens for example if all non-faulty processes propose the same value. This algorithm relies on an all-to-all binary communication abstraction (BV-broadcast) and an eventual synchrony assumption, which are described in the next subsections. The algorithm is built incrementally. We first present a simple algorithm that satisfies only the consensus safety properties (BBC-Validity and BBC-Agreement). This algorithm is then extended with the eventual synchrony assumption to satisfy the consensus liveness property (BBC-Termination). The aim of this incremental approach is to facilitate the understanding and the proofs.

5.1 The BV-broadcast all-to-all communication abstraction

The binary value broadcast (BV-broadcast) communication abstraction has been introduced in [49] (its implementation is recalled in Appendix A).

Definition BV-broadcast is an all-to-all communication abstraction that provides the processes with a single operation denoted BV_broadcast(). When a process invokes BV_broadcast TAG(m), we say that it "BV-broadcasts the message TAG(m)". The content of a message m is 0 or 1 (hence the term "binary-value" in the name of this communication abstraction).

In a BV-broadcast instance, each non-faulty process p_i BV-broadcasts a binary value and obtains a set of binary values, stored in a local read-only set variable denoted bin_values_i. This set, initialized to $\emptyset$, increases when new values are received. BV-broadcast is defined by the four following properties.

- BV-Obligation. If at least $(t + 1)$ non-faulty processes BV-broadcast the same value v, v is eventually added to the set bin_values_i of each non-faulty process p_i.
- BV-Justification. If p_i is non-faulty and $v \in bin\_values_i$, v has been BV-broadcast by a non-faulty process.
- BV-Uniformity. If a value v is added to the set bin_values_i of a non-faulty process p_i, eventually $v \in bin\_values_j$ at every non-faulty process p_j.
- BV-Termination. Eventually the set bin_values_i of each non-faulty process p_i is not empty.

A BV-broadcast property The following property is an immediate consequence of the previous properties. Eventually the sets bin_values_i of the non-faulty processes p_i (i) become non-empty, (ii) become equal, (iii) contain all the values broadcast by non-faulty processes, and (iv) never contain a value broadcast only by Byzantine processes. However, no non-faulty process knows when (ii) and (iii) occur.

5.2 A safe binary Byzantine consensus algorithm in $\mathcal{BAMP}_{n,t}[t < n/3]$

Figure 2 describes a simple binary Byzantine consensus algorithm, which satisfies the BBC-Validity and BBC-Agreement properties in the system model $\mathcal{BAMP}_{n,t}[t < n/3]$. This algorithm, which is round-based and relies on the previous BV-broadcast abstraction, has the same structure as the randomized consensus algorithm introduced in [49].

Local variables Each process p_i manages the following local variables.

- est_i: local current estimate of the decided value. It is initialized to the value proposed by p_i.
- r_i: local round number, initialized to 0.
- $bin\_values_i[1..]$: array of binary values; $bin\_values_i[r]$ (initialized to $\emptyset$) stores the local output set filled by BV-broadcast associated with round r. (This unbounded array can be replaced by a single local variable bin_values_i, reset to $\emptyset$ at the beginning of every round. We consider here an array to simplify the presentation.)
- b_i: auxiliary binary value.
- $values_i$: auxiliary set of values.

Message types The algorithm uses two message types, denoted EST and AUX. Both are used in each round, hence they always appear with a round number.

- EST$[r]()$ is used at round r by p_i to BV-broadcast its current decision estimate est_i.
- AUX$[r]()$ is used by p_i to disseminate its current value of $bin\_values_i[r]$ (with the help of the broadcast() macro-operation).

The algorithm Let us consider Figure 2. After it has deposited its binary proposal in est_i (line 01), each non-faulty process p_i enters a sequence of asynchronous rounds. Each round r uses a BV-broadcast instance whose associated local variable at process p_i is $bin\_values_i[r]$.

```
operation bin_propose(v_i) is
(01) est_i ← v_i; r_i ← 0;
(02) while (true) do
(03)    r_i ← r_i + 1;
(04)    BV_broadcast EST[r_i](est_i);
(05)    wait_until (bin_values_i[r_i] ≠ ∅);
(06)    broadcast AUX[r_i](bin_values_i[r_i]);
(07)    wait_until (messages AUX[r_i](b_val_p(1)), ..., AUX[r_i](b_val_p(n−t)) have been received
                    from (n − t) different processes p(x), 1 ≤ x ≤ n − t, and their contents are
                    such that ∃ a non-empty set values_i such that (i) values_i ⊆ bin_values_i[r_i]
                    and (ii) values_i = ∪_{1≤x≤n−t} b_val_x);
(08)    b_i ← r_i mod 2;
(09)    if (values_i = {v}) // values_i is a singleton whose element is v
(10)      then est_i ← v; if (v = b_i) then decide(v) if not yet done end if;
(11)      else est_i ← b_i
(12)    end if;
(13) end while.
```

Figure 2: A safe algorithm for binary Byzantine consensus in $\mathcal{BAMP}_{n,t}[t < n/3]$

The behavior of a non-faulty process p_i during a round r can be decomposed in three phases.

- Phase 1: Coordinated exchange of current estimates (lines 03-05).
  Process p_i first progresses to the next round, and BV-broadcasts its current estimate (line 04). Then p_i waits until its set $bin\_values_i[r]$ is not empty (let us recall that, when $bin\_values_i[r]$ becomes non-empty, it has not necessarily its final value).
- Phase 2: Second exchange of estimates to favor convergence (lines 06-07).
  In this second phase, p_i broadcasts (hence, this is neither a BV-broadcast nor a RB-broadcast) a message AUX$[r]()$ whose content is $bin\_values_i[r]$ (line 06). Then, p_i waits until it has received a set of values $values_i$ satisfying the two following properties.
  - $values_i \subseteq bin\_values_i[r]$. Thanks to the BV-Justification property, this ensures that (even if Byzantine processes send fake messages AUX$[r]()$ containing values proposed only by Byzantine processes) $values_i$ will contain only values broadcast by non-faulty processes.
  - The values in $values_i$ come from the messages AUX$[r]()$ of at least $(n - t)$ different processes.

  Hence, at any round r, after line 07, $values_i \subseteq \{0, 1\}$ and contains only values BV-broadcast at line 04 by non-faulty processes.
- Phase 3: Try to decide (lines 08-12).
  This phase is a purely local computation phase, during which (if not yet done) p_i tries to decide the value $b = r \bmod 2$ (lines 08 and 10), depending on the content of $values_i$.
  - If $values_i$ contains a single element v (line 09), then v becomes p_i's new estimate. Moreover, v is candidate to be decided. To ensure BBC-Agreement, v can be decided only if $v = b$. The decision is realized by the statement decide(v) (line 10).
  - If $values_i = \{0, 1\}$, then p_i cannot decide. As both values have been proposed by non-faulty processes, to entail convergence to agreement, p_i selects one of them (b, which is the same at all non-faulty processes) as its new estimate (line 11).

  Let us observe that the invocation of decide(v) by p_i does not terminate the participation of p_i in the algorithm, namely p_i continues looping forever. The algorithm can be made terminating, using the randomized technique presented in [49]. Instead we preserve the simplicity of this algorithm and postpone a deterministic terminating solution in Section 5.5.

5.3 Safety proof

Process p_i being a non-faulty process, let $values_i^r$ denote the value of the set $values_i$ which satisfies the predicate of line 07. Moreover, let us recall that, given a run, C denotes the set of non-faulty processes in this run.

Lemma 5. *Let $t < n/3$. If at the beginning of a round r, all non-faulty processes have the same estimate v, they never change their estimate value thereafter.*

Proof Let us assume that all non-faulty processes (which are at least $n - t > t + 1$) have the same estimate v when they start round r. Hence, they all BV-broadcast the same message $\text{EST}[r](v)$ at line 04. It follows from the BV-Justification and BV-Obligation properties that each non-faulty process p_i is such that $bin\_values_i[r] = \{v\}$ at line 05, and consequently can broadcast only $\text{AUX}[r](\{v\})$ at line 06. Considering any non-faulty process p_i, it then follows from the predicate of line 07 ($values_i$ contains only v), the predicate of line 09 ($values_i$ is a singleton), and the assignment of line 10, that est_i keeps the value v. $\square_{Lemma\ 5}$

Lemma 6. *Let $t < n/3$.* $\big((p_i, p_j \in C) \wedge (values_i^r = \{v\}) \wedge (values_j^r = \{w\})\big) \Rightarrow (v = w)$.

Proof Let p_i be a non-faulty process such that $values_i^r = \{v\}$. It follows from line 07 that p_i received the same message $\text{AUX}[r](\{v\})$ from $(n - t)$ different processes, i.e., from at least $(n - 2t)$ different non-faulty processes. As $n - 2t \geq t + 1$, this means that p_i received the message $\text{AUX}[r](\{v\})$ from a set Q_i including at least $(t + 1)$ different non-faulty processes.

Let p_j be a non-faulty process such that $values_j^r = \{w\}$. Hence, p_j received $\text{AUX}[r](\{w\})$ from a set Q_j of at least $(n - t)$ different processes. As $(n - t) + (t + 1) > n$, it follows that $Q_i \cap Q_j \neq \emptyset$. Let $p_k \in Q_i \cap Q_j$. As $p_k \in Q_i$, it is a non-faulty process. Hence, at line 06, p_k sent the same message $\text{AUX}[r](\{\})$ to p_i and p_j, and we consequently have $v = w$. $\square_{Lemma\ 6}$

Lemma 7. *Let $t < n/3$. The value decided by a non-faulty process was proposed by a non-faulty process.*

Proof Let us consider the round $r = 1$. Due to the BV-Justification property of the BV-broadcast of line 04, it follows that the sets $bin\_values_i[1]$ contains only values proposed by non-faulty processes. Consequently, the non-faulty processes broadcast at line 06 messages $\text{AUX}[1]()$ containing sets with values proposed only by non-faulty processes. It then follows from the predicate (i) of line 07 ($values_i^1 \subseteq bin\_values_i[1]$), and the BV-Justification property of the BV-broadcast abstraction, that the set $values_i^1$ of each non-faulty process contains only values proposed by non-faulty processes. Hence, the assignment of est_i (be it at line 10 or 11) provides it with a value proposed by a non-faulty process. The same reasoning applies to rounds $r = 2$, $r = 3$, etc., which concludes the proof of the lemma. $\square_{Lemma\ 7}$

Lemma 8. *Let $t < n/3$. No two non-faulty processes decide different values.*

Proof Let r be the first round during which a non-faulty process decides, let p_i be a non-faulty process that decides in round r (line 10), and let v be the value it decides. Hence, we have $values_i^r = \{v\}$ where $v = (r \bmod 2)$.

If another non-faulty process p_j decides during round r, we have $values_j^r = \{w\}$, and, due to Lemma 6, we have $w = v$. Hence, all non-faulty processes that decide in round r, decide v. Moreover, each non-faulty process that decides in round r has previously assigned $v = (r \bmod 2)$ to its local estimate est_i.

Let p_j be a non-faulty that does not decide in round r. As $values_i^r = \{v\}$, and p_j does not decide in round r, it follows from Lemma 6 that we cannot have $values_j^r = \{1 - v\}$, and consequently $values_j^r = \{0, 1\}$. Hence, in round r, p_j executes line 11, where it assigns the value $(r \bmod 2) = v$ to its local estimate est_j.

It follows that all non-faulty processes start round $(r+1)$ with the same local estimate $v = r \bmod 2$. Due to Lemma 5, they keep this estimate value forever. Hence, no different value can be decided in a future round by a non-faulty process that has not decided during round r, which concludes the proof of the lemma. $\square_{Lemma\ 8}$

Lemma 9. *Let the system model be $\mathcal{BAMP}_{n,t}[t < n/3]$. No non-faulty process remains blocked forever in a round.*

Proof Let us assume by contradiction that there is a first round in which some non-faulty process p_i remains blocked forever. As all non-faulty processes terminate round $(r-1)$, they all start round r and all invoke the round r instance of BV-broadcast. Due to the BV-Termination property, the wait_until() statement of line 05 terminates at each non-faulty process. Then, as all non-faulty processes broadcast a message AUX$[r]()$ (line 06), it follows that the wait_until() statement of line 07 terminates at each non-faulty process. It follows that there is no first round at which a non-faulty process remains blocked forever during round r. $\square_{Lemma\ 9}$

Lemma 10. *Let the system model be $\mathcal{BAMP}_{n,t}[t < n/3]$. If all non-faulty processes p_i terminate a round r with $values_i^r = \{v\}$, they all decide by round $(r+1)$.*

Proof If all non-faulty processes are such that $values_i^r = \{v\}$, and the round r is such that $v = (r \bmod 2)$, it follows from lines 08-10 that (if not yet done) each non-faulty process decides during round r.

If r is such that $v \neq (r \bmod 2)$, each non-faulty process sets its current estimate to v (line 10). As during the next round we have $v = ((r+1) \bmod 2)$, and $values_i^{r+1} = bin\_values_i[r+1] = \{v\}$ at each non-faulty process p_i, each non-faulty process decides during round $(r+1)$. $\square_{Lemma\ 10}$

Lemma 11. *Let the system model be $\mathcal{BAMP}_{n,t}[t < n/3]$. If every non-faulty process p_i terminates a round r with $values_i^r = \{0,1\}$, they it decides by round $(r+2)$.*

Proof If every non-faulty processes p_i is such that $values_i^r = \{0,1\}$, it executes line 11 during round r, and we have $est_i = (r \mod 2) = v$ when it starts round $(r+1)$. Due to Lemma 5, it keeps this estimate forever. As all non-faulty processes execute rounds $(r+1)$ and $(r+2)$ (Lemma 9) and $v = ((r+2) \mod 2)$, we have $values_i^{r+2} = \{v\}$, at each non-faulty process p_i. It follows that each non-faulty process decides at line 10. $\square_{Lemma\ 11}$

Theorem 2. *The algorithm described in Figure 2 satisfies the safety consensus properties.*

Proof The proof follows from Lemma 7 (BBC-Validity) and Lemma 8 (BBC-Agreement). $\square_{Theorem\ 2}$

Decision The algorithm described in Figure 2 does not guarantee decision. This may occur for example when some non-faulty processes propose 0, the other non-faulty processes propose 1, and the Byzantine processes play double game, each proposing 0 or 1 to each non-faulty process, so that it never happens that at the end of a round all non-faulty processes have either $values_i = \{0,1\}$, or they all have $values_i = \{v\}$ with v either 0 or 1. In other words, if not all non-faulty processes propose the same initial value, Byzantine processes can make, round after round, some non-faulty processes have $values_i = \{0,1\}$, while the rest of non-faulty processes have $values_i = \{v\}$, with $v \neq (r \mod 2)$, avoiding them to decide.[7]

5.4 Eventual synchrony assumption

Consensus impossibility It it well-known that there is no consensus algorithm ensuring both safety and liveness properties in fully asynchronous message-passing systems in which even a single process may crash [24]. As the crash failure model is less severe than the Byzantine failure model, the consensus impossibility remains true if processes may commit Byzantine failures.

To circumvent such an impossibility, and ensure the consensus termination property, the model must be enriched with additional computational power. Examples of such a power can be provided with failure detectors [18, 26, 34], constraints on the set of input vectors [25, 51], randomization [4, 49, 60], or synchrony assumptions [20, 21] (see [61] for more developments). As announced, we consider here the approach based on additional synchrony assumptions.

Additional synchrony assumption In the following, it is assumed that after some finite time τ, there is an upper bound δ on message transfer delays. This assumption is denoted $\diamond Synch$ (Eventual Synchrony assumption). To exploit it through the use of timers, we also assume that processes can measure accurately intervals of time, although they do not need to have synchronized clocks.

Notation The model $\mathcal{BAMP}_{n,t}[t < n/3]$ enriched with $\diamond Synch$ is denoted $\mathcal{BAMP}_{n,t}[t < n/3, \diamond Synch]$.

[7] In the case of the randomized binary consensus algorithm of [49], the common coin guarantees termination with probability 1, because eventually the singleton value in $values_i$ will match the coin.

5.5 A binary Byzantine consensus algorithm in $\mathcal{BAMP}_{n,t}[t < n/3, \diamond Synch]$

In this section, we describe our binary Byzantine consensus algorithm that is guaranteed to terminate in $O(t)$ rounds, which is known to be optimal [23]. The algorithm described in Figure 3 is an extension of the safe algorithm of Figure 2, whose aim is to add the consensus termination property. The lines with the same numbers are the same in both algorithms. The new lines in Figure 3 are numbered "New$x$", where x is an integer, and the lines that are modified are prefixed by "M-". In addition to the use of local timers, to eventually benefit from the $\diamond Synch$ assumption, this extended round-based algorithm uses the *round coordinator* notion: in each round a predetermined process plays a special role, namely, the round coordinator strives to impose a value that the other processes would decide. To this end, each process in turn plays the round coordinator role [18, 21]. More precisely, the processes being p_1, .., p_n, the coordinator of round r is the process p_i such that $i = ((r-1) \mod n) + 1$.[8]

Additional local variables and message type In addition to est_i, r_i, $bin\_values_i[r]$, and $values_i$, each process p_i manages the following local variables.

- $timer_i$ is a local timer, and $timeout_i$ a timeout value, both used to exploit the assumption $\diamond Synch$.
- $coord_i$ is the index of the current round coordinator.
- aux_i is an auxiliary set of values, used to store the value (if any) that the current coordinator strives to impose as decision value.

The coordinator of round r, uses the message type COORD_VALUE$[r]()$ to broadcast the value it tries to favor to become the decided value.

```
operation bin_propose(v_i) is
(01)  est_i ← v_i; r_i ← 0;
      timeout_i ← 0;
(02)  while (true) do
(03)     r_i ← r_i + 1;
(New1)  coord_i ← ((r_i − 1) mod n) + 1;
        timeout_i ← timeout_i + 1; set timer_i to timeout_i;
(04)     BV_broadcast EST[r_i](est_i);
(New2)  if (i = coord_i) then
          wait_until (bin_values_i[r_i] = {w}); // w is the first value to enter bin_values_i[r_i]
          broadcast COORD_VALUE[r_i](w)
        end if;
(M-05)  wait_until ((bin_values_i[r_i] ≠ ∅) ∧ (timer_i expired));
(New3)  set timer_i to timeout_i;
(New4)  if ((COORD_VALUE[r_i](w) received from p_coord_i) ∧ (w ∈ bin_values_i[r_i]))
          then aux_i ← {w}
          else aux_i ← bin_values_i[r_i]
        end if;
(M-06)  broadcast AUX[r_i](aux_i);
(M-07)  wait_until ((messages AUX[r_i](b_val_p(1)), ..., AUX[r_i](b_val_p(n−t)) have been received
                    from (n − t) different processes p(x), 1 ≤ x ≤ n − t, and their contents are
                    such that ∃ a non-empty set values_i such that (i) values_i ⊆ bin_values_i[r_i]
                    and (ii) values_i = ∪_{1≤x≤n−t} b_val_x) ∧ (timer_i expired));
(New5)  if (when considering the whole set of the messages AUX[r_i]() received, several sets
             values1_i, values2_i, ... satisfy the previous wait predicate) ∧ (one of them is aux_i)
                    then values_i ← aux_i end if;
(08)     b_i ← r_i mod 2;
(09)     if (values_i = {v}) // values_i is a singleton whose element is v
(10)       then est_i ← v; if (v = b_i) then decide(v) if not yet done end if;
(11)       else est_i ← b_i
(12)     end if;
(13)  end while.
```

Figure 3: A safe and live algorithm for binary Byzantine consensus in $\mathcal{BAMP}_{n,t}[t < n/3, \diamond Synch]$

Description of the extended algorithm The following items explain the new and modified statements that appear in Figure 3.

[8] Let us notice that the notion of round coordinator is different from the notion of an eventual leader. A round coordinator is a simple algorithmic mechanism which can be implemented in $\mathcal{BAMP}_{n,t}[t < n/3]$, while an eventual leader is an oracle which cannot be implemented in $\mathcal{BAMP}_{n,t}[t < n/3]$.

- At line New1, p_i computes the current round coordinator, and sets its local timer whose expiry is used in the predicate of line M-05. The timeout value is initialized before entering the loop, and then increased at every round.
- Line New3 is a simple reset of the timer, whose expiry is used in the predicate of the modified line M-07.
- Lines New2, New4, M-06, and New5 realize a mechanism that allows the current round coordinator to try to impose the first value that enters into its bin_values set as the decided value[9]. Combined with the fact that there is a time after which the messages exchanged by the non-faulty processes are timely, this ensures that there will be a round during which the non-faulty processes will have a single value in their sets $values_i$ (which –by Lemma 10– entails their decision).
- Modified lines M-05 and M-07: addition of the timer expiration in the predicate considered at the corresponding line.

As just seen, the idea made operational by these new or modified statements is the following: benefit from a non-faulty round coordinator to entail decision, by requiring this process to broadcast a proposed value so that all non-faulty processes adopt it. To this end:

- The round coordinator p_k broadcasts the message COORD_VALUE$[r_i](w)$, where w is the first value that enters its bin_values set (line New2). If p_k is non-faulty, the timeout values of the non-faulty processes are big enough, and there is a bound on message transfer delays, all non-faulty processes will receive it before their timer expiration at line M-06.
- Then, assuming the previous item, all non-faulty processes set aux_i to $\{w\}$ (line New4), and broadcast it (line M-06). The predicate $w \in bin\_values_i[r_i]$ is used to prevent a Byzantine coordinator to send fake values that would foil non-faulty processes.
- Finally, all the non-faulty processes will receive the message AUX$[r_i](\{w\})$ from $(n-t)$ different processes, and by line New5 will set $values_i = \{w\}$. This will entail their decision during the round $(r+1)$ or $(r+2)$.

From asynchrony to synchrony In order to guarantee decision, after the eventual synchrony assumption holds and the timeout value at each non-faulty process is big enough (i.e., bigger than the upper bound on message transmission delay), we need that eventually all non-faulty processes execute rounds synchronously. Observe that, due to initial asynchrony, non-faulty processes can start the consensus algorithm at different instants. Moreover, due to the potential participation of Byzantine processes, some non-faulty processes can advance rounds, without deciding, while other non-faulty processes are still executing previous rounds. By using a timeout that grows by 1 each round all processes eventually reach a round from which they behave synchronously.

Lemma 12. *Let us consider the algorithm of Figure* 3*. Eventually the non-faulty processes attain a round from which they behave synchronously.*

Proof By $\diamond Synch$ there is eventually an unknown bound δ on message transfer delays. As indicated in Section 2, it is assumed that local processing time is equal to zero. (Alternatively, two additional proofs are provided in Appendices B and C that do not rely on this assumption.) In the following, time units will be given in integers. The notation t with a subscript (for example t_{first_0}) wil be used to represent a time measurement that is given by the number of time units that have passed since the algorithm started, as measured by an omniscient global observer G. G observes time passing at the same rate as the non-faulty processes and events can occur on integer time units.

We will use the following definitions:

- t_{first_r} is the time as measured by G at which the first non-faulty process p_{first} reaches round r (t_{first_0} is the time when the first non-faulty process starts the consensus).
- t_{last_r} is the time as measured by G at which the last non-faulty process p_{last} reaches round r (t_{last_0} is the time when the last non-faulty process starts the consensus).

For a round to be synchronous, all non-faulty processes must arrive at that round with enough time to broadcast their messages to all non-faulty processes before the timeout of that round expires at any non-faulty process. In the case that the last non-faulty process to arrive at the round is the coordinator, it may take up to

[9] A similar mechanism based on a round coordinator and the same message exchange pattern is used in the Byzantine synchronous algorithm presented in [3] and in [52] to solve asynchronous k-set agreement with restricted failure detectors.

3 message delays before its COORD_VALUE$[r]()$ message is received by all non-faulty processes (this includes up to 2 message delays until a value enters its $bin\_values[r]$ and an additional message delay to broadcast COORD_VALUE$[r]()$). Therefore, we must have a round r where

$$t_{last_r} + \delta \leq t_{first_r} + timeout_r. \tag{1}$$

Note that given the timeout starts at 0 on round 0 and grows by one each round, we can replace $timeout_r$ for r for any round r.

Consider the first round r' where $timeout_{r'} \geq \delta$ is satisfied. For any round r'' where $r'' \geq r'$ the maximum amount of time for $p_{last_{r''}}$ to complete the round will be $2 \times timeout_{r''}$. This is due to the fact that the last non-faulty process to arrive at a round will not have to wait longer than δ to receive the messages needed to satisfy the conditions on lines M-05 and M-07, thus the time taken to execute the round will be no more than the length of the two timeouts. All other non-faulty processes will take at least $2 \times timeout_{r''}$ to complete round r''.

From this the time where the last non-faulty process reaches some round r'' can be written as:

$$t_{last_{r''}} = t_{last_{r'}} + 2\left(\sum_{x=r'}^{r''-1} x\right).$$

And the time when the first non-faulty process reaches round r'' as:

$$t_{first_{r''}} \geq t_{first_{r'}} + 2\left(\sum_{x=r'}^{r''-1} x\right).$$

Plugging this into inequality 1 results in:

$$t_{last_{r'}} + 2\left(\sum_{x=r'}^{r''-1} x\right) + 3 \times \delta \leq t_{first_{r'}} + 2\left(\sum_{x=r'}^{r''-1} x\right) + r''.$$

Removing equal components we have:

$$t_{last_{r'}} + 3 \times \delta \leq t_{first_{r'}} + r''.$$

Thus, by round $r'' = t_{last_{r'}} + 3 \times \delta - t_{first_{r'}}$ synchrony is ensured.

It will now be shown that once Inequality (1) is satisfied for one round r'' (where $timeout_{r''} \geq \delta$), it will remain satisfied in the all following rounds. Consider round $r'' + 1$, given that Inequality (1) is satisfied for round r'', we have:

$$t_{last_{r''}} + 3 \times \delta \leq t_{first_{r''}} + timeout_{r''}. \tag{2}$$

And it needs to be shown that the following inequality is true:

$$t_{last_{r''+1}} + 3 \times \delta \leq t_{first_{r''+1}} + timeout_{r''+1} \tag{3}$$

Using the same argument as above, the times at which the last and first processes arrive at round $r'' + 1$ are: $t_{last_{r''+1}} = t_{last_{r''}} + 2 \times timeout_{r''}$ and $t_{first_{r''+1}} \geq t_{first_{r''}} + 2 \times timeout_{r''}$. Plugging this into inequality (3) results in:

$$t_{last_{r''}} + 2 \times timeout_{r''} + 3 \times \delta \leq t_{first_{r''}} + 2 \times timeout_{r''} + timeout_{r''+1}.$$

Removing equal parts leads to:

$$t_{last_{r''}} + 3 \times \delta \leq t_{first_{r''}} + timeout_{r''+1}.$$

This inequality, which is equivalent to Inequality (3) has the same components as Inequality (2), except having $timeout_{r''+1}$ instead of $timeout_{r''}$. Therefore, Inequality (3) must be satisfied, given that Inequality (2) is satisfied. By induction this holds true for any round after r''. $\square_{Lemma\ 12}$

5.6 Proof of the $\diamond Synch$-based algorithm

The proof consists of two parts: (i) show that the added statements preserve the consensus safety properties proved for the time-free algorithm of Figure 2, and (ii) show that all non-faulty processes eventually decide.

Lemma 13. *The algorithm described in Figure* 3 *satisfies the BBC-Validity and BBC-Agreement properties.*

Proof The proof consists in showing that the Lemmas 5, 6, 7 and 8 remain correct when considering the algorithm of Figure 3. Basically, these proofs remain correct because, as the new and modified statements do not assign values to the sets $bin\_values_i[r]$ at the non-faulty processes, and no property of bin_values_i is related to a timing assumption, the set $bin\_values_i[r]$ of a non-faulty process p_i can never contain values proposed by Byzantine processes only. It follows from this observation that the local variables est_i and $values_i$ of any non-faulty process (defined or updated at lines M-07, New5, 10, or 11) can contain only values from non-faulty processes. More specifically we have the following.

- Lemma 5. Let r be the considered round, and v be the current estimate of the non-faulty processes. We then have $bin\_values_i[r] = \{v\}$ at line M-05 of every non-faulty process p_i.
  - If the round coordinator p_k is non-faulty, we have at every non-faulty process $aux_i = bin\_values_i[r] = \{v\}$. It then follows that $values_i^r = \{v\}$ and the lemma remains true due to lines 09 and 10.
  - If the round coordinator p_k is Byzantine and sends possibly different values to the non-faulty processes, let us consider a non-faulty process that receives the message COORD_VALUE$[r](\{1-v\})$. As $(1-v) \notin bin\_values_i[r]$, at line New4, p_i executes the "else" part where it sets aux_i to $\{v\}$ (the only value in $bin\_values_i[r]$), and the lemma follows.
- Lemma 6. As it does not depend on the timers, and is related only to the fact that each of the sets $values_i^r$ and $values_j^r$ of two non-faulty processes are singletons, the proof remains valid.
- Lemma 7. The proof follows from the fact that the sets bin_values_i of any non-faulty process can contain only values proposed by non-faulty processes.
- Lemma 8. As it relies only on the sets $values_i^r$ of the non-faulty processes, this proof remains correct.

$\square_{Lemma\ 13}$

Lemma 14. *The algorithm described in Figure* 3 *ensures that every non-faulty process decides.*

Proof Let us first observe that, as timers always expire, the "wait" statements (modified lines M-05 and M-07) always terminate, and consequently Lemma 9 remains true. The reader can also check that the proof of Lemma 10 remains valid.

It remains to show that there is eventually a round r at the end of which all non-faulty processes p_i have the same value w in their set variables ($values_i^r = \{w\}$) (from which decision follows due to Lemma 10) The proof shows that, due to (a) the eventual synchrony assumption, (b) the round coordinator mechanism, and (c) the messages COORD_VALUE$[\,]()$ sent by the round coordinators, there is a round r such that $values_i^r = \{w\}$ at each non-faulty process.

Let us consider a time τ from which (due to Lemma 12) the system behaves synchronously (the timeout values of all non-faulty processes are such that all the messages exchanged by the non-faulty processes arrive timely). Let r be the smallest round number coordinated by a non-faulty process p_k after τ. At line New2 of round r, p_k broadcasts COORD_VALUE$[r](w)$, being w the first value that enters its set $bin\_values_k[r]$. The message COORD_VALUE$[r](w)$ is received timely by all non-faulty processes, that set aux_i to $\{w\}$ in line New4. Consequently, in line M-06 all non-faulty processes broadcast AUX$[r](\{w\})$, and receive in line M-07 $(n-t)$ AUX$[r](\{w\})$ messages from different processes, setting in line New5 $values_i$ to $\{w\}$. By Lemma 10, all non-faulty processes decide w by round $r+1$, which concludes the proof of the lemma. $\square_{Lemma\ 14}$

Theorem 3. *The algorithm described in Figure* 3 *solves binary Byzantine consensus in the system model* $\mathcal{BAMP}_{n,t}[t < n/3, \diamond Synch]$.

Proof The proof follows directly from Lemma 13 (BBC-Validity and BBC-Agreement) and Lemma 14 (BBC-Termination). $\square_{Theorem\ 3}$

6 Related Work

Byzantine consensus Our validated predicate-based consensus differs from previous definitions in the way validity is defined. The seminal paper on the agreement between Byzantine generals considers a single source; its validity requires that if the source proposes only one value, then only this value can be decided [42]. In the case where multiple, potentially Byzantine, processes propose values, validity often requires that if all non-faulty processes propose the same value then this value should be decided [21]. The classic validity used in the crash model is sometimes used in the Byzantine model requiring that a decided value is proposed by some process [18] but the notion of valid value proposed by a Byzantine process can be unclear. A predicate was previously used to assess whether a value is valid, however, the resulting predicate-based validity requires that if all non-faulty processes propose the same valid value then this value should be decided [37]. A variant of this definition does not require the value proposed by all correct to be decided but can be violated with a non-null probability [14]. It has been informally suggested for randomized consensus that a binary value could be decided if it was provided along with some validating data [15]. When values are not necessarily binary, the decided value must sometimes be within the range of the values proposed by the non-faulty processes [6] or sufficiently close to the median of values proposed by non-faulty processes [64]. Finally, validity sometimes requires that the decided value is either a special value $\perp$ or is a value proposed by a non-faulty process [19]. As $\perp$ is a predefined value, deciding this value is similar to aborting [29]. Our validity property allows for a valid value proposed only by Byzantine processes to be decided rather than aborting.

Multivalued to binary consensus reduction Despite its simplicity of presentation, there are surprisingly few reductions of multivalued consensus to binary consensus. The first reduction was designed for the synchronous model [65]. The key idea is to use a single binary consensus instance such that a non-faulty process would only propose value 1 in a round if it knows that all non-faulty processes received the same value before. In the crash model, some reductions executed multiple binary consensus instances, but always sequentially [55, 67]. The first crash-resilient reduction completes after n binary consensus instances [55] whereas the second one improved upon it to complete after $\lceil \log n \rceil$ sequential binary consensus instances [67]. By contrast, our reduction algorithm, besides applying to the Byzantine model, executes all binary consensus instances in parallel. Interestingly, a very similar algorithm to our reduction was used to solve agreement on a common subset in order to acheive secure computation [5]. The same single binary instance reduction to [65] was later used in the asynchronous model for randomized consensus [19, 54]. Besides being randomized, the drawback of these reductions is that they may have to return $\perp$ in case no decision can be taken regarding the proposed value. By contrast, our reduction decides a valid value that was proposed.

As far as we know, eventually-synchronous signature-free Byzantine consensus algorithms do not use reduction. The classic implementation can only terminate in a round coordinated by a non-faulty process if the faulty coordinator sends inappropriate values [21], a drawback our algorithm does not have.

Leader-based consensus It is well known that leader-based consensus algorithms have some drawbacks. In crash-prone systems, the weakest class of failure detectors that allows to solve the consensus problem is $\diamond S$ [18] (which is equivalent to the eventual leader failure detector Ω introduced in [17]). Its eventual accuracy property guarantees that there is a time after which there is a non-faulty process that is never suspected by the non-faulty processes. Building upon this guarantee, several consensus algorithms were proposed, namely, the processes proceed in asynchronous rounds managed by a pre-determined leader or coordinator that tries to impose a value as the decision. The approach is similar to our binary Byzantine consensus algorithm, with the difference that we do not rely on any failure detector or eventual leader. Moreover, $\diamond S$ and Ω cannot be easily implemented in Byzantine systems.

Leader-based algorithms, like PBFT [16], FAB [46] or Zyzzyva [36], do not adopt the rotating coordinator approach. The drawback of classic rotating coordinator approaches is that they may have to run through the $t+1$ rounds even in periods of synchrony [21]. Instead, leader-based algorithms may terminate faster in period of synchrony if the leader is non-faulty because they do not have to run through the $t+1$ rounds, as explained in [13]. These leader-based approaches require a period of synchrony long enough not only for consensus to terminate, provided that the leader is non-faulty or the value is stable [46], but also to elect a non-faulty leader or obtain a stable value. Another interesting point is that PBFT executes $O(t)$ rounds in which all non-faulty processes broadcast $2t+1$ checkpoint messages, which results in $O(n^4)$ bits exchanged when $t = \Omega(n)$, whereas in our multivalued consensus algorithm, all the non-faulty processes broadcast in each of $O(t)$ rounds of n binary consensus instances, leading to the same $O(n^4)$ bits complexity.

To bypass the difficulty of electing a leader in the presence of Byzantine failures, a synchronous Byzantine agreement that implements a virtual leader was proposed [41], it avoids having to detect and remove a malicious leader. Similar to our algorithm, processes exchange proposals that they record in a vector where proposals from non-faulty processes are identical. The main difference is that it assumes that processes synchronize their clocks as it requires the vector to be full and contain proposals from all non-faulty processes within some fixed time Δ [40]. Our approach is different as we do not need the vector to be full or to contain proposals from all non-faulty processes, we simply apply the results of the binary consensus instances as a bitmask to the vector.

To conclude, our algorithm does not need a leader, making it harder to delay termination. In addition, while our algorithm is based on a rotating coordinator it does not require $t + 1$ rounds to terminate if good conditions are met, in which case non-faulty processes can decide even in rounds coordinated by faulty processes.

Unrestricted blockchains Original blockchain systems like Bitcoin [56] and Ethereum [66] target a peer-to-peer model where the number of processes is frequently changing and is not known by processes. Their consensus algorithms rely on processes to generate a computationally-intensive proof-of-work to limit the power of malicious processes when trying to reach consensus. The drawback is that they usually solve consensus probabilistically [27]. To improve the scalability of Bitcoin, Bitcoin-NG [22] solves a variant of consensus with probabilistic termination and probabilistic agreement by relying on a leader. Elastico [44] reaches a variant of consensus whose proposed values satisfy a specific predicate function among subcommittees of c processes each, but requires the communication to be synchronous.

Private and consortium blockchains With the advent of private and consortium blockchains, where the participation is restricted to n processes, new blockchain systems suggested the use of classic Byzantine fault tolerant algorithms (e.g., [16]) when $n > 3t$. After the seminal PBFT approach, several implementations were proposed to reduce latency when $n > 5t$ (e.g., [46]), simplify the development (e.g., [1]) or trade fault-tolerance for performance (e.g., [43]). Tendermint consensus uses a variant of PBFT with a rotating leader election [38]. Ripple's consensus algorithm relies on unique node lists that define a quorum system where sufficient nodes are controlled by the Ripple company [63]. Stellar uses adaptive quorum systems to implement consensus [47]. R3 has just released Corda that may use a Byzantine fault tolerant algorithm or a crash-tolerant consensus algorithm with stronger assumption [30]. Hyperledger Fabric [12] uses PBFT but should provide support to a variant of Apache Kafka in the near future. A recent implementation [48] suggested to reach consensus with a probabilistic termination.

7 Conclusion

This paper has presented a new multivalued Byzantine consensus algorithm tailored for consortium blockchains. It is asymptotically time optimal and resilience optimal and does not rely on a leader, randomization or signatures. It combines a reduction from multivalued to binary consensus that applies a bitmask to an array of proposals and a binary consensus to build this bitmask. By spawning binary consensus instances in parallel, our reduction is as far as we know the first to decide a non-predefined value in a sequence of $O(1)$ binary consensus instances.

Despite the large interest in blockchains (and cryptocurrency applications), there is relatively few results on the formalization of this problem. Our paper tried to address this limitation by formalizing the blockchain consensus as a general variant of Byzantine consensus. The resulting algorithm decouples the problem of validating a block from the problem of deciding on a block; it builds upon the number n of consortium members to avoid Sybil attacks; and it is indulgent in that safety is not affected when communication is delayed.

Acknowledgements

Tyler Crain and Vincent Gramoli were supported by the Australian Research Council's Discovery Projects funding scheme (project number 160104801). Vincent Gramoli is the recipient of the Australian Research Council Discovery International Award. Mikel Larrea was supported by the Spanish Research Council, grant TIN2016-79897-P, and the Basque Country Research Council, grants IT980-16 and MV_2016_1_0031. Michel Raynal was supported by the French ANR project DESCARTES (grant 16-CE40-0023-03) devoted to distributed software engineering.

References

[1] Aublin P.-L., Guerraoui R., Knezevic N., Quema V., and Vukolić M., The next 700 BFT protocols. *ACM Transactions on Computer Systems*, 32(4), Article 12, 45 pages (2015)

[2] Aspnes J., Randomized protocols for asynchronous consensus. *Distributed Computing*, 16(2-3):165-175 (2003)

[3] Berman P. and Garay J.A., Cloture voting: $n/4$-resilient distributed consensus in $t + 1$ rounds. *Mathematical System Theory*, 26(1):3-19 (1993)

[4] Ben-Or M., Another advantage of free choice: completely asynchronous agreement protocols. *Proc. 2nd Annual ACM Symposium on Principles of Distributed Computing (PODC'83)*, ACM Press, pp. 27-30 (1983)

[5] Ben-Or M., Kelmer B., and Rabin T., Asynchronous Secure Computations with Optimal Resilience. *Proc. Annual ACM Symposium on Principles* pp. 183-192 (1994)

[6] Bonomi S., Del Pozzo A., Potop-Butucaru M., and Tixeuil S., Approximate Agreement under Mobile Byzantine Faults. *Proc. 36th IEEE International Conference on Distributed Computing Systems (ICDCS'16)*, pp. 727-728 (2016)

[7] Borran F., Hutle M. and Schiper A., Timing analysis of leader-based and decentralized Byzantine consensus algorithms. *Proc. 5th Latin-American Symposium on Dependable Computing (LADC'11)*, IEEE Press, pp.166-175 (2011)

[8] Bouzid Z., Mostéfaoui A., and Raynal M., Minimal synchrony for byzantine consensus. *Proc. 34th Annual ACM Symposium on Principles of Distributed Computing (PODC'15)*, ACM press, pp. 461-470 (2015)

[9] Bracha G., Asynchronous Byzantine agreement protocols. *Information & Computation*, 75(2):130-143 (1987)

[10] Bracha G. and Toueg S., Asynchronous consensus and broadcast protocols. *Journal of the ACM*, 32(4):824-840 (1985)

[11] Buterin V., Ethereum: platform review, opportunites and challenges for private and consortium blockchains (2016)

[12] Cachin C., Blockchain - From the anarchy of cryptocurrencies to the enterprise. Keynote presentation at *20th Int'l Conference on Principles of Distributed Systems (OPODIS'16)* (2016)

[13] Cachin C., Guerraoui R., and Rodrigues L., *Reliable and secure distributed programming*, Springer, 367 pages (2011) ISBN 978-3-642-15259-7

[14] Cachin C., Kursawe K., Petzold F., and Shoup V., Secure and Efficient Asynchronous Broadcast Protocols *Proc. 21st Annual International Cryptology Conference (CRYPTO)*, pp.524-541, 2001

[15] Cachin C., Kursawe K., and Shoup V., Random oracles in Constantinople: practical asynchronous Byzantine agreement using cryptography. *Journal of Cryptology*,18(3):219-246 (2005, first version: PODC 2000)

[16] Castro M. and Liskov B., Practical Byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems*, 20(4):398-461 (2002)

[17] Chandra T., Hadzilacos V., and Toueg S., The weakest failure detector for solving consensus. *Journal of the ACM*, 43(4):685-722 (1996)

[18] Chandra T. and Toueg S., Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225-267 (1996)

[19] Correia M., Ferreira Neves N., and Verissimo P., From consensus to atomic broadcast: time-free Byzantine-resistant protocols without signatures. *The Computer Journal*, 49(1):82-96 (2006)

[20] Dolev D., Dwork C. and Stockmeyer L., On the minimal synchronism needed for distributed consensus. *Journal of the ACM*, 34(1):77-97 (1987)

[21] Dwork C., Lynch N., and Stockmeyer L., Consensus in the presence of partial synchrony. *Journal of the ACM*, 35(2):288-323 (1988)

[22] Eyal I., Gencer A.E., Sirer E.G., and van Renesse R., Bitcoin-NG: a scalable blockchain protocol. *Proc. 13th Usenix Conference on Networked Systems Design and Implementation (NSDI'16)*, pp.45-59 (2016)

[23] Fischer M.J. and Lynch N.A., A lower bound for the time to assure interactive consistency. *Information Processing Letters*, 14(4):183-186 (1982)

[24] Fischer M.J., Lynch N.A., and Paterson M.S., Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374-382 (1985)

[25] Friedman R., Mostéfaoui A., Rajsbaum S., and Raynal M., Distributed agreement problems and their connection with error-correcting codes. *IEEE Transactions on Computers*, 56(7):865-875 (2007)

[26] Friedman R., Mostéfaoui A., and Raynal M., Simple and efficient oracle-based consensus protocols for asynchronous Byzantine systems. *IEEE Transactions on Dependable and Secure Computing*, 2(1):46-56 (2005)

[27] Garay J., Kiayias A., and Leonardos, N. The Bitcoin Backbone Protocol: Analysis and Applications. Advances in Cryptology - EuroCrypt, pp. 281-310. (2015)

[28] Guerraoui R., Indulgent algorithms. *Proc. 19th Annual ACM Symposium on Principles of Distributed Computing (PODC'00)*, ACM Press. pp. 289-297. (2000)

[29] Hadzilacos V. and Toueg S., On deterministic abortable objects. *Proc. 32th Annual ACM Symposium on Principles of Distributed Computing (PODC'13)*, ACM Press, pp.4-12 (2013)

[30] Hearn M., Corda: a distributed ledger. Version 0.5 (2016)

[31] Herlihy M., Distributed computing and blockchains. Keynote presentation at *18th Int'l Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS'16)* (2016)

[32] Herlihy M.P. and Wing J.M, Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463-492 (1990)

[33] Imbs D. and Raynal M., Trading t-resilience for efficiency in asynchronous Byzantine reliable broadcast. *Parallel Processing Letters*, 26(4), 8 pages (2017)

[34] Kihlstrom K.P., Moser L.E., and Melliar-Smith P.M., Byzantine fault detectors for solving consensus. *The Computer Journal*, 46(1):16-35 (2003)

[35] King V. and Saia J., Byzantine agreement in expected polynomial time. *Journal of the ACM*, 63(2), Article 13, 21 pages (2016)

[36] Kotla R., Alvisi L., Dahlin M., Clement A., and Wong E.L., Zyzzyva: speculative Byzantine fault tolerance. *ACM Transactions on Computer Systems,* 27(4):7:1-7:39 (2009)

[37] Kursawe K., Optimistic asynchronous Byzantine agreement. Manuscript (2000)

[38] Kwong J., Tendermint: Consensus without mining. v.0.7 (2016)

[39] Lamport L., Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558-565 (1978)

[40] Lamport L., Leaderless Byzantine consensus. *United States Patent, Microsoft Corporation, Redmond, WA (USA)* (2010)

[41] Lamport L., Leaderless Byzantine Paxos. *Proc. 25th International Simposium on Distributed Computing. (DISC'11)*, pp.141-142 (2011)

[42] Lamport L., Shostack R., and Pease M., The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3)-382-401 (1982)

[43] Liu S., Viotti P., Cachin C., Quéma V., and Vukolić M., XFT: practical fault tolerance beyond crashes. *Proc. 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI'16)*, ACM Press, pp. 485-500 (2016)

[44] Luu L., Narayanan V., Zheng C., Baweja K., Gilbert S. and Saxena P., A secure sharding protocol for open blockchains. *ACM Conference on Computer and Communications Security (CCS'16)*, ACM Press, pp. 17-30 (2016)

[45] Lynch N.A., *Distributed algorithms*. Morgan Kaufmann Pub., San Francisco (CA), 872 pages (1996) ISBN 1-55860-384-4

[46] Martin J.-Ph. and Alvisi L., Fast Byzantine consensus. *IEEE Transactions on Dependable and Secure Computing*, 3(3):202-215 (2006)

[47] Mazieres D., The stellar consensus protocol: A federated model for internet-level consensus. (2015)

[48] , Miller A., Xia Y., Croman K., Shi E., and Song D., The Honey Badger of BFT Protocols *Proc. of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, p.31-42 (2016)

[49] Mostéfaoui A., Moumen H., and Raynal M., Signature-free asynchronous binary Byzantine consensus with $t < n/3$, $O(n^2)$ messages, and $O(1)$ expected time. *Journal of ACM*, 62(4), Article 31, 21 pages (2015)

[50] Mostéfaoui A., Mourgaya E., and Raynal M., Asynchronous implementation of failure detectors. *Int'l IEEE Conference on Dependable Systems and Networks (DSN'03)*, IEEE Computer Press, pp. 351-360 (2003)

[51] Mostéfaoui A., Rajsbaum S., and Raynal M., Conditions on input vectors for consensus solvability in asynchronous distributed systems. *Journal of the ACM*, 50(6):922-954 (2003)

[52] Mostéfaoui A. and Raynal M., k-Set agreement and limited accuracy failure detectors. *Proc. 19th ACM SIGACT-SIGOPS Int'l Symposium on Principles of Distributed Computing (PODC'00)*, ACM Press, pp. 143-152 (2000)

[53] Mostéfaoui A. and Raynal M., Intrusion-tolerant broadcast and agreement abstractions in the presence of Byzantine processes. *IEEE Transactions on Parallel and Distributed Systems*, 27(4):1085-1098 (2016)

[54] Mostéfaoui A. and Raynal M., Signature-free asynchronous Byzantine systems: from multivalued to binary consensus with $t < n/3$, $O(n^2)$ messages, and constant time. *Acta Informatica*, DOI 10.1007/s00236-016-0269-y, 20 pages (2016)

[55] Mostéfaoui A., Tronel F., and Raynal M., From binary consensus to multivalued consensus in asynchronous message-passing systems. *Information Processing Letters*, 73:207-213 (2000)

[56] Nakamoto S., Bitcoin: a peer-to-peer electronic cash system. http://www.bitcoin.org (2008)

[57] Natoli C. and Gramoli V., The balance attack against proof-of-work blockchains: The R3 testbed as an example. *arXiv, technical report* 1612.09426 (2016)

[58] Natoli C. and Gramoli V., The blockchain anomaly. *Proc. 5th IEEE Int'l Symposium on Network Computing and Applications (NCA'16)*, IEEE ComputerPress, pp. 310-317 (2016)

[59] Pease M., R. Shostak R., and Lamport L., Reaching agreement in the presence of faults. *Journal of the ACM*, 27:228-234 (1980)

[60] Rabin M., Randomized Byzantine generals. *Proc. 24th IEEE Symposium on Foundations of Computer Science (FOCS'83)*, IEEE Computer Society Press, pp. 116-124(1983)

[61] Raynal M., *Communication and agreement abstractions for fault-tolerant asynchronous distributed systems*. Morgan & Claypool, 251 pages (2010) ISBN 978-1-60845-293-4

[62] Schneider F.B., Implementing fault-tolerant services using the state machine approach. *ACM Computing Surveys*, 22(4):299-319 (1990)

[63] Schwartz A.B.D. and Youngs N., The ripple protocol consensus algorithm. *Ripple Labs Inc.* (2014)

[64] Stolz D. and Wattenhofer R., Byzantine Agreement with Median Validity. *Proc. 19th International Conference on Principles of Distributed Systems (OPODIS'15)*, pp. 22:1-22:14 (2015)

[65] Turpin R. and Coan B.A., Extending binary Byzantine agreement to multivalued Byzantine agreement. *Information Processing Letters*, 18:73-76 (1984)

[66] Wood G., Ethereum: A secure decentralized generalized transaction ledger. *White paper* (2015)

[67] Zhang J. and Chen W., Bounded cost algorithms for multivalued consensus using binary consensus instances. *Information Processing Letters*, 109:10056-1009 (2009)

A A BV-broadcast algorithm

A simple self-explanatory algorithm implementing the BV-broadcast abstraction in the system model $\mathcal{BAMP}_{n,t}[t < n/3]$ is described in Figure 4. A proof of it can be found in [49] where BV-broadcast was used in the context of randomized consensus.

B Correctness proof of the $\diamond Synch$ algorithm with Byzantine tolerant distributed clocks

It is possible to replace the timeouts used in the algorithm in Figure 3 and to relax the assumption that processing time is null with the Byzantine fault tolerant distributed clocks of [21] to decide when processes can move forward in rounds. We have then the following lemma, which is then used instead of Lemma 12 in the proof of Theorem 3.

Lemma 15. *Let us consider the algorithm of Figure* 3 *with the addition of Byzantine fault tolerant distributed clocks from* [21]. *Eventually the non-faulty processes attain a round from which they behave synchronously.*

```
operation BV_broadcast MSG(v_i) is
(01)  broadcast B_VAL(v_i).

when B_VAL(v) is received
(02)  if (B_VAL(v) received from (t + 1) different processes and B_VAL(v) not yet broadcast)
(03)    then broadcast B_VAL(v) // a process echoes a value only once
(04)  end if;
(05)  if (B_VAL(v) received from (2t + 1) different processes)
(06)    then bin_values_i ← bin_values_i ∪ {v}    // local delivery of a value
(07)  end if.
```

Figure 4: An algorithm implementing BV-broadcast in $\mathcal{BAMP}_{n,t}[t < n/3]$ (from [49])

C Correctness proof of the $\diamond Synch$ algorithm with a catch-up mechanism

From asynchrony to synchrony Here a catch-up mechanism is provided as a modification to the algorithm in Figure 3 that ensures processes eventually execute synchronous rounds where non-faulty processes may observe time at different rates and processing time is non-negligible.

Lemma 12 ensures that all non-faulty processes eventually participate in synchronous rounds in a system where eventually the non-faulty processors run at the same speed and computation time is negligible. As this timing assumption cannot be ensured in all systems, this section extends the algorithm with a *catch-up* mechanism, which guarantees progress in systems where the difference in process speed is bounded by some unknown constant.

As we have seen, in order to guarantee a decision, after the assumed $\diamond Synch$ assumption holds, it is needed that eventually all non-faulty processes execute rounds synchronously. Observe that, due to initial asynchrony, non-faulty processes can start the consensus algorithm at different instants. Moreover, due to the potential participation of Byzantine processes, some non-faulty processes can advance rounds –without deciding– while other non-faulty processes are still executing previous rounds.

Before describing the catch-up mechanism the *mini-round* notation will be defined. Each round r is split into two mini-rounds, with the first mini-round representing lines 03 to (New4) and the second representing lines M-06 to 12. Like rounds, mini-rounds can be identified by a number with the first mini-round starting at 0 and increasing by one. For example round 3 is made up of mini-rounds 6 and 7.

Catch-up mechanism The mechanism that allows non-faulty processes to catch up relies on the following two changes to the algorithm[10]:

- First, the timers are started only when the conditions of lines M-05 and M-07 are satisfied, i.e., on line M-05 only once $bin\_values_i[r_i] \neq \emptyset$ is true, is the timer started. Similarly on line M-07 the timer is only started when $n - t$ AUX$[r_i]()$ messages satisfying the conditions are received. Let the timeout start at 0 on mini-round 0 and grow by 1 each mini-round.
- Second, when a process is in a mini-round ρ and receives messages corresponding to another mini-round $\rho' > \rho$ from $(t + 1)$ different processes (i.e., from at least one non-faulty process), the process no longer waits for timers in mini-rounds ρ, .., $(\rho' - 1)$. It still completes these mini-rounds, but does it without waiting for timers expiration.

The idea of these mechanisms is to allow late or slow non-faulty processes to catch up to the most advanced non-faulty processes (as measured by their mini-round number).

We assume that each process has a local clock that allows it to measure time units. Time units will be given in integers. A process uses its local clock to measure the amount of time it waits for a timeout (where a timeout of 1 is 1 time unit). The notation t with a subscript (for example t_{first_0}) will be used to represent a time measurement that is given by the number of time units that have passed since the algorithm started, as measured by an omniscient global observer G. By $\diamond Synch$, processes are able to observe time at different rates, but within an unknown fixed bound. For simplicity assume that the fastest process observes time at a rate no faster than observed by the global observer G and all other processes observe time at this rate or slower.

Notations Similarly to previous sections, the following notations and definitions are used in the following.

- δ is a fixed, but unknown bound on message transfer delays as ensured by $\diamond Synch$ and measured in time units as observed by G.

[10] Similar mechanisms are used by PBFT [16].

- t_{first_ρ} is the time, as measured by G, at which the first non-faulty process p_{first_ρ} reaches mini-round ρ (t_{first_0} is the time at which the first non-faulty process starts the consensus).
- t_{last_ρ} is the time, as measured by G, at which the last non-faulty process p_{last_ρ} reaches mini-round ρ (t_{last_0} is the time at which the last non-faulty process starts the consensus).
- θ_{fast} (resp. θ_{slow}) is the minimum (resp. maximum) amount of time, as observed by G, for any process to perform the computation of any mini-round (a unknown bounded difference between θ_{fast} and θ_{slow} is ensured by $\diamond Synch$).
- γ_{fast_ρ} is the minimum amount of time, as observed by G, in a mini-round ρ that any process waits on line M-05 or M-07 before starting its timer for that mini-round.
- Mini-round ρ_δ is the first mini-round where $timeout > \delta$.

Lemma 16. *Consider the algorithm of Figure* 3 *enriched with the previous catch-up mechanism. There is a mini-round ρ_t such that in ρ_t and all following mini-rounds all non-faulty processes must wait for at least part of the $timeout$, i.e., they do not receive $t+1$ messages from a round larger than ρ_t until after they start waiting at the timeout of round ρ_t.*

Proof Let us only consider mini-rounds where $\rho_t > \rho_\delta$. For all non-faulty processes to wait at a timeout in a mini-round ρ_t the last non-faulty process to arrive at that mini-round must arrive before it could receive a message from some other non-faulty process that is executing a later mini-round (note that given $\rho_t > \rho_\delta$, this can only be ensured when the processes are no more than 1 mini-round apart), i.e., the following must hold:

$$t_{last_{\rho_t}} < t_{first_{\rho_{t+1}}}. \tag{4}$$

Let us now find when this can be satisfied. By definition, a process can spend no less time than $(\gamma_{fast_{\rho'}} + \theta_{fast} + timeout)$ in a mini-round ρ'. Given that the timeout starts at 0 on mini-round 0 and grows by 1 in each mini-round, $timeout$ can be replaced with ρ for any mini-round ρ. We then have:

$$t_{first_{\rho'}} \geq t_{first_{\rho_\delta}} + \left(\sum_{x=\rho_\delta}^{\rho'-1} \gamma_{fast_x} + \theta_{fast} + x \right).$$

Notice that from the component $\sum_{x=\rho_\delta}^{\rho'-1} x$ (i.e., the timeout), the value of $t_{first_{\rho'}}$ is quadratic in the number of mini-rounds.

Now consider the slowest non-faulty process starting from mini-round ρ_δ that does not wait at a timeout. This process can spend no more time in a mini-round ρ' than 2 message delays plus the processing time for the mini-round, i.e., $2 \times \delta + \theta_{slow}$. The reason for this is as follows: To complete the mini-round the process must satisfy the condition on line M-05 or M-07. Keep in mind that, given that the process does not wait at a timeout, it must have received $t+1$ messages from a later mini-round, meaning that some non-faulty process has already completed ρ'. Line M-05 requires $(bin\_values_i[r_i] \neq \perp)$. It is known that a non-faulty process has completed this round and that process has therefore already satisfied this condition, and that all non-faulty processes have executed line 04 (given that this is the slowest process). With this, and the BV-Uniformity property of the BV_broadcast() operation, all non-faulty processes will have a value in $bin\_values_i[r_i]$ after at most 2 message delays after the slowest process executes the BV_broadcast(). Similarly, on line M-07 all non-faulty processes will receive the $n-t$ AUX messages needed to satisfy the condition in 1 message delay after the slowest non-faulty process broadcasts its AUX message, but may need to wait another message delay in case a non-faulty process had a value enter its $bin\_values_i[r_i]$ immediately before broadcasting its AUX message. We then have:

$$t_{last_{\rho'}} \leq t_{last_{\rho_\delta}} + \left(\sum_{x=\rho_\delta}^{\rho'-1} 2 \times \delta + \theta_{slow} \right).$$

Notice that the value of $t_{last_{\rho'}}$ is linear in the number of mini-rounds.

Now given $t_{first_{\rho'}}$ is quadratic while $t_{last_{\rho'}}$ is linear, inequality (4) must eventually be satisfied, thus there will be a mini-round where all non-faulty processes wait for at least part of their timeout.

It will now be shown by induction that after $timeout > (2 \times \delta + \theta_{slow})$ inequality (4) holds for all following mini-rounds. Consider $t_{last_{\rho_t}} < t_{first_{\rho_{t+1}}}$ is satisfied, let us now show that $t_{last_{\rho_t+1}} < t_{first_{\rho_{t+2}}}$ is satisfied as well. For this to not hold, the slowest non-faulty process must spend more time on round ρ_t than the fastest

non-faulty process spends on round $(\rho_t + 1)$. Given that $\rho_t > \rho_\delta$, process $p_{last_{\rho_t}}$ must receive $(t+1)$ messages from round $(\rho_t + 1)$ before process $p_{first_{\rho_t+1}}$ completes round $(\rho_t + 1)$. Once these messages are received we have already shown that this process will take no more than $2 \times \delta + \theta_{slow}$ time to complete the round. Thus, as long as $timeout > (2 \times \delta + \theta_{slow})$, which will eventually be true given $\diamond Synch$ and the growing timeout, process $p_{last_{\rho_t}}$ will reach round $(\rho_t + 1)$ before $p_{first_{\rho_t+1}}$ reaches round $(\rho_t + 2)$. $\square_{Lemma\ 16}$

Lemma 17. *Consider the algorithm of Figure 3 enriched with the previous catch-up mechanism. Eventually the non-faulty processes attain a mini-round from which they behave synchronously.*

Proof By Lemma 16 it is known that there exists a mini-round ρ_t where all non-faulty processes wait for at least part of their timeout. This is ensured after rounds where $timeout > (2 \times \delta + \theta_{slow})$. Consider we are in such rounds. Now for a mini-round to be synchronous, all non-faulty processes need to arrive at that mini-round with enough time to broadcast its messages to all non-faulty processes before any non-faulty process moves onto the next mini-round. In the case that the last non-faulty process to arrive at the round is the coordinator, it may take up to 3 message delays before its COORD_VALUE$[r]()$ message is received by all non-faulty processes (this includes up to 2 message delays until a value enters its $bin\_values[r]$ and an additional message delay to broadcast COORD_VALUE$[r]()$). Notice that before a non-faulty process starts its timer for a mini-round it must wait until the condition on line M-05 or M-07 is satisfied. Thus, for a mini-round ρ'_t to by synchronous where $\rho'_t \geq \rho_t$, the following needs to be ensured:

$$t_{last_{\rho'_t}} + (3 \times \delta) + \theta_{slow} \leq t_{first_{\rho'_t}} + \gamma_{fast_{\rho'_t}} + timeout. \tag{5}$$

By time $(t_{first_{\rho'_t}} + \gamma_{fast_{\rho'_t}} + \theta_{fast})$ at least one process has satisfied the condition on line M-05 or M-07 (this is given by the definition of γ). As a result all processes will receive $(t+1)$ messages from mini-round ρ'_t by time $(t_{first_{\rho'_t}} + \gamma_{fast_{\rho'_t}} + \theta_{fast} + \delta)$. Now given Lemma 16 and that $(\rho'_t - 1) > \delta$, it is known that that the slowest process is no further behind than waiting at the timeout of mini-round $(\rho'_t - 1)$. After getting these $(t+1)$ messages from mini-round ρ'_t the slow process will then skip the timeout of mini-round $(\rho'_t - 1)$ and reach the following mini-round in at most 2 additional message delays (2 message delays are needed for the same reasons given in Lemma 16 to satisfy the condition line M-05 or M-07) plus any processing time, i.e.:

$$t_{last_{\rho'_t}} \leq t_{first_{\rho'_t}} + \gamma_{fast_{\rho'_t}} + \theta_{fast} + \theta_{slow} + (3 \times \delta).$$

Now plugging this into inequality (5) leads to $timeout \geq (7 \times \delta) + (2 \times \theta_{slow}) + \theta_{fast}$ (note that $2 \times \theta_{slow}$ is included to account for possible processing times in both rounds $(\rho'_t - 1)$ and ρ'_t). But given that the timeout grows in each mini-round and that δ, θ_{fast}, and θ_{slow} are bound by $\diamond Synch$ there will eventually be a mini-round where this holds true.

Finally, notice that as long as the timeout is this large (i.e. $timeout \geq (7 \times \delta) + (2 \times \theta_{slow}) + \theta_{fast}$) and Lemma 16 holds then the above argument is valid for any round. Now given that $timeout \geq (7 \times \delta) + (2 \times \theta_{slow}) + \theta_{fast}$ is larger than the timeout needed for Lemma 16 to hold for every following round, once inequality (5), i.e. synchrony, it true for one round, it will also hold for every following round. $\square_{Lemma\ 17}$