This paper introduces a deterministic Byzantine consensus algorithm that relies on a new weak coordinator. As opposed to previous algorithms that cannot terminate in the presence of a faulty or slow coordinator, our algorithm can terminate even when its coordinator is faulty, hence the name weak coordinator. The key idea is to allow processes to complete asynchronous rounds as soon as they receive a threshold of messages, instead of having to wait for a message from a coordinator that may be slow.

The resulting algorithm assumes partial synchrony, is resilience optimal, time optimal and does not need signatures. Our presentation
is didactic: we first present a simple safe binary Byzantine consensus algorithm, modify it to ensure termination, and finally present an optimized reduction from multivalue consensus to binary consensus that may terminate in 4 message delays.

To evaluate our algorithm, we deployed it on 100 machines distributed in 5 datacenters across different continents and compared its performance against the randomized solution from Mostefaoui, Moumen and Raynal [PODC’14] that terminates in $O(1)$ rounds in expectation. Our algorithm always outperforms the latter even in the presence of Byzantine behaviors. Our algorithm has a subsecond average latency in most of our geo-distributed experiments, even when attacked by a well-engineered coalition of Byzantine processes.

1 Introduction and Related Work

To circumvent the impossibility of solving consensus in asynchronous message-passing systems [24] where processes can be faulty or Byzantine [31], researchers typically use randomization [5, 6, 15] or additional synchrony assumptions.

Randomized algorithms can use per-process “local” coins or a shared “common” coin to solve consensus probabilistically among $n$ processes despite $t < \frac{n}{3}$ Byzantine processes. When based on local coins, the existing algorithms converge in $O(n^{2.5})$ expected time [27]. A recent randomized algorithm without signature [37] solves consensus in $O(1)$ expected time under a fair scheduler. The fair scheduler assumption was later relaxed in an extended version [38] that we refer to as Coin in the remainder of the paper. Unfortunately, implementing a common coin increases the message complexity of the consensus algorithm.

To avoid the need of a common coin and solve the consensus problem deterministically, researchers have assumed partial or eventual synchrony [22]. Interestingly, these solutions typically require a unique coordinator process, sometimes called a leader, to be non-faulty [3, 8, 16, 21, 22, 28, 32, 34]. The advantage is that if the coordinator is non-faulty and if the messages are delivered in a timely manner in an asynchronous round, then the coordinator broadcasts its proposal to all processes and this value is decided after a constant number of message delays. The drawback is that a faulty coordinator can dramatically impact the algorithm performance by leveraging the power it has in a round and imposing its value to all. Non-faulty processes thus have no other choices but to decide nothing in this round.

In this paper, we present a weak coordinator alternative that does not suffer from this drawback. It allows us to introduce a new deterministic
Byzantine consensus algorithm that is time optimal, resilience optimal and
does not need signatures. As opposed to a classic (strong) coordinator, the
weak coordinator does not impose its value. On the one hand, this allows
non-faulty processes to decide a value quickly without the help of the coordi-
nator. On the other hand, the coordinator helps the algorithm terminating
if non-faulty processes know that they proposed distinct values that might
all be decided. Furthermore, having a weak coordinator allows rounds to be
executed optimistically without waiting for a specific message. This differs
from classic BFT algorithms \cite{16} that have to wait for a particular message
from their coordinator and sometimes have to recover from a slow network
or faulty coordinator.

To mitigate the problem of a slow or Byzantine coordinator, other ap-
proaches were previously explored. Some protocols progressively reduce the
time allocated to a coordinator to solve consecutive consensus instances
in order to force the change of a slow coordinator \cite{4, 20}. While this still
requires a (strong) coordinator in each round, it favors the fastest coordi-
nator in successive rounds. An exponential information gathering tree was
used to terminate in $t + 3$ rounds without a coordinator \cite{9}. Other solu-
tions \cite{22,45} require at least $O(t)$ rounds. By contrast our weak coordinator
only helps agreement by suggesting a value while still allowing termination
in a constant number of message delays and thus differs from the classic
coordinator \cite{17,22} or the eventual leader that cannot be implemented in
$BAMP_{n,t}[t < n/3]$.

**Application to consortium blockchains.** To motivate our algorithm, we
study its applicability to the recent context of blockchains \cite{40}. Blockchains
originally aimed at tracking ownerships of digital assets where any Internet
user could solve a cryptopuzzle before proposing, for consensus, a block
of asset transactions. The consortium blockchains \cite{12} became promising
at reducing the amount of resources consumed by avoiding to resolve the
cryptopuzzle but restricting the set of proposers to $n$ known processes.

These consortium blockchains seem similar to replicated state machines \cite{30
44} where a sequence of commands must be decided by multiple processes.
Some blockchains already use Byzantine fault tolerant consensus, for exam-
ple, Hyperledger \cite{2} uses a consensus based on a classic coordinator \cite{8}, Hon-
eybadger \cite{36} uses a randomized algorithm \cite{37} and the Red Belly Blockchain \cite{25}
uses a preliminary version of the algorithm we introduce here \cite{19}. A slight
difference with state machine replication is that the block at index $x$ of a
blockchain must embed the hash of the block decided at instance number
($x - 1$). This relation between instances is interesting as it entails a natu-
ral mechanism during a consensus instance for discarding fake proposals or, instead, extracting a valid value out of various proposals.

We thus propose a variant of the consensus problem that allows us to extend common definitions of Byzantine consensus, that either assume that no value proposed only by Byzantine processes can be decided \[18, 38, 39\], or that any value (i.e., possibly proposed by a Byzantine process) can be decided \[22, 26, 33, 34, 43\]. Interestingly, the validity property we propose allows a decided value to combine multiple proposals but is less strict than interactive consistency \[42\] or vector consensus \[41\]: for example, it does not require the decided value to combine at least \(t + 1\) values proposed by correct processes.

Geo-distributed experimentation with Byzantine coalitions. To validate our expectations experimentally, we deployed our consensus algorithm on 100 Amazon VMs located in 5 datacenters on different continents. We also implemented “Coin” the recent randomized algorithm from Moustéoui et al. \[37\] used in the HoneyBadger blockchain \[36\] and demonstrated that under all our workloads, our algorithm outperforms “Coin” that is known to terminate in \(O(1)\) round in expectation. This is due to both the overhead of the coin implementation that slows down every round and the risks of being unlucky at tossing the coin by increasing the number of rounds needed to decide.

As Byzantine behaviors are known to affect drastically performance of (strong) coordinator-based consensus \[4, 20\], we also implemented 4 different Byzantine attacks: Byz1 where Byzantine processes send a bit \(b\) where the protocol specification expects them to send \(\neg b\); Byz2 where Byzantine processes are mute; Byz3 where Byzantine processes send a combination of random and flipped values and Byz4 where Byzantine processes form a coalition to limit the progress of non-faulty nodes from one round to another by exploiting a Byzantine coordinator and sending messages without waiting. Interestingly, the latency exceeds slightly the second only under the Byz3 attacks.

Finally, we combine our consensus algorithm with an optimized variant of the reduction of multivalue to binary consensus of Ben-Or et al. \[7\] to propose a novel Democratic Byzantine Fault Tolerant (DBFT) consensus algorithm applicable to consortium blockchains that terminates in 4 messages delays in the good case, when all non-faulty processes propose the same value.

Roadmap. Section 2 presents the model. Section 3 presents the binary
2 A Byzantine Computation Model

Asynchronous processes. The system is made up of a set $\Pi$ 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. Both notations $i \in Y$ and $p_i \in Y$ are used to say that $p_i$ belongs to the set $Y$.

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 “\text{send \ tag(m) to } p_j\text{ “}, where \text{tag} is the type of the message and \text{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 “\text{receive()}”. The macro-operation \text{broadcast \ tag(m)} is used as a shortcut for “\text{for each } p_i \in \Pi \text{ do } \text{send \ tag(m) to } p_j \text{ end for}”.

Failure model. Up to $t$ processes can exhibit a Byzantine behavior \cite{[42]}. 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.

**Additional synchrony assumption.** It is 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, we enrich the model with additional synchrony assumptions. It is assumed that after some finite time $\tau$, there is an upper bound $\delta$ on message transfer and process computation delays. This eventual (or partial) synchrony assumption is denoted $\diamond Synch$.

**Notations.** The acronym $BAMP_{n,t}[\emptyset]$ is used to denote the previous basic Byzantine Asynchronous Message-Passing computation model; $\emptyset$ means that there is no additional assumption. The basic computation model strengthened with the additional constraint $t < n/3$ is denoted $BAMP_{n,t}[t < n/3]$. The latter computation model strengthened with the eventual synchrony constraint $\diamond Synch$ is denoted $BAMP_{n,t}[t < n/3, \diamond Synch]$.

### 3 Binary Byzantine Consensus

In this section we propose a solution to the binary consensus using a weak coordinator that requires neither signatures, nor randomization. For the sake of simplicity, we build the algorithm incrementally by first recalling the binary consensus problem, then presenting a safe binary consensus algorithm in the $BAMP_{n,t}[t < n/3]$ model and finally presenting a safe and live consensus algorithm in the $BAMP_{n,t}[t < n/3, \diamond Synch]$ model.

Let $\mathcal{V}$ be the set of values that can be proposed by a process to the consensus. While $\mathcal{V}$ can contain any number ($\geq 2$) of values in multivalued consensus, it contains only two values in binary consensus, e.g., $\mathcal{V} = \{0, 1\}$. Assuming that each non-faulty process proposes a value, the binary Byzantine consensus (BBC) problem is for each of them to decide on a value in such a way that the following properties are satisfied:

- **BBC-Termination.** Every non-faulty process eventually decides on a value.
- **BBC-Agreement.** No two non-faulty processes decide on different values.
- **BBC-Validity.** If all non-faulty processes propose the same value, no other value can be decided.
3.1 The Binary Value Broadcast Communication Abstraction

Our binary consensus algorithm relies on a binary value all-to-all communication abstraction, denoted BV-broadcast, originally introduced for randomized consensus [38], and restated in the appendix.

In a BV-broadcast instance, each non-faulty process \( p_i \) broadcasts a binary value and obtains (BV-delivers) 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.

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.

3.2 Local variables and message types

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 asynchronous 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.

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).

3.3 A safe asynchronous binary Byzantine consensus algorithm

For the sake of simplicity, we first introduce a new leaderless algorithm ensuring BBC-Validity and BBC-Agreement properties in the system model $BAMP_{n,t}[t < n/3]$ but not BBC-termination. The algorithm is depicted in Figure 1 and provides the process $p_i$ with the operation $bin.propose(v_i)$ to propose its initial value $v_i$. Process $p_i$ proceeds in asynchronous rounds and decides value $v$ when invoking decide($v$) at line $10$.

```
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 $est[r_i](est_i)$; // add to $bin.values[r_i]$ upon BV-delivery
(05)    wait until ($bin.values[r_i]$ ≠ {});
(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 $values_i$ where (i) $values_i = \cup_{1≤x≤n−t}b.val_{p(x)}$
    and (ii) $values_i \subseteq bin.values_i[r_i]$);
(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.
(14)  when b-val[$r$]($v$) is BV-delivered by BV.broadcast[$r$] do
    $bin.values_i[r] ← bin.values_i[r] \cup \{v\}$;
```

Figure 1: A safe algorithm for the binary Byzantine consensus in $BAMP_{n,t}[t < n/3]$. 

8
After it has deposited its binary proposal in $est_i$ (line 01), each non-faulty process $p_i$ enters a sequence of asynchronous rounds. During a round $r$, each non-faulty process $p_i$ proceeds in three phases.

Phase 1: Binary value broadcast to filter out the values of Byzantine processes. Process $p_i$ first progresses to the next round, and binary value broadcasts (BV-broadcast) its current estimate (line 04).

At each process $p_i$, within the BV广播() algorithm, after receiving the same value from $t + 1$ processes, process $p_i$ then rebroadcasts this value. Each process $p_i$ BV-delivers a value $v$ by adding it to its $bin\_values_i$ set only if it receives $v$ from $2t + 1$ distinct processes. Eventually the sets $bin\_values$ of all non-faulty processes become non-empty, equal, and contain exclusively all values broadcast by non-faulty processes [19]. When a value is BV-delivered it is then added to $bin\_values_i[r]$ (line 14). 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: Exchanging estimates to converge to an agreement. This second phase runs between line 06 and line 07). In this phase, $p_i$ broadcasts normally 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.

- The values in $values_i$ come from the messages $aux[r]()$ of at least $(n - t)$ different processes.
- $values_i \subseteq bin\_values_i[r]$. Thanks to the BV-broadcast that filters out Byzantine value, 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.

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: Deciding upon estimate convergence to round number modulo 2. The third phase runs between line 08 and line 12. This phase is a purely local computation phase, during which (if not yet done) $p_i$ tries to decide the value $b = r \mod 2$ (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 a candidate for the consensus decision. To ensure
BBC-Agreement, \( v \) can be decided only if \( v = b \). The decision is realized by the statement \texttt{decide}(v) (line 10).

- If \( \text{values}_i = \{0, 1\} \), 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 for the same round) as its new estimate (line 11).

Let us observe that the invocation of \texttt{decide}(v) by \( p_i \) does not terminate the participation of \( p_i \) in the algorithm, namely \( p_i \) continues looping forever. This is because a deciding process may need to help other processes converging to the decision in the two subsequent rounds. This algorithm can be modified to avoid this infinite loop, but to preserve the simplicity in the presentation, we postpone a deterministic terminating solution to Section 3.4.

The proof of correctness of algorithm 1 is deferred to the appendix.

### 3.4 Psync: Safe and Live Consensus in \( BAMP_{n,t}[t < n/3, \Diamond Synch] \)

We now present \( Psync \), an algorithm solving the binary Byzantine consensus problem in the \( BAMP_{n,t}[t < n/3, \Diamond Synch] \) model. Similar to the safe algorithm (Section 3.3), \( Psync \) does not use signatures or randomization and has the following additional characteristics:

- \( Psync \) is time optimal [23] in that it terminates in \( O(t) \) message delays.
- When all non-faulty processes propose the same value, \( Psync \) terminates in \( O(1) \) message delays, even under asynchrony.
- \( Psync \) does not wait for a message from its coordinator and does not need recovery.

The \( Psync \) algorithm is presented in Figure 2 as an extension of the safe algorithm in Figure 1, with new and modified lines prefixed with “New” and “M-”, respectively. Lines prefixed by “Opt” are optional optimizations. In addition to the use of local timers, to eventually benefit from the \( \Diamond Synch \) assumption, the algorithm uses a \textit{weak coordinator}: the weak coordinator of round \( r \) is the process \( p_i \) such that \( i = ((r - 1) \mod n) + 1 \). Note that this new round coordinator is only used to help agreement by suggesting a value and thus differs from the classic coordinator [17,22].

**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.
Figure 2: A safe and live algorithm for the binary Byzantine consensus in $BAMP_{n,t}[t < n/3, \Diamond{Synch}]$; line (Opt1) is an optimization only applied in the multivalued reduction presented in Section 4; line (Opt2) is a mechanism to prevent unnecessary rounds from being executed.

```
operation bin_propose(v_i) is
  est_i ← v_i; r_i ← 0;
  timeout_i ← 0;
  while (true) do
    r_i ← r_i + 1;
    if (est_i = -1) then est_i ← 1; // "fast-path" for round 1, only used in the reduction in Sect. 4
      else BV.broadcast est[r_i](est_i);
    end if:
    (Opt1)
    wait_until (bin_values[r_i] ≠ ∅);
      timeout_i ← timeout_i + 1; set timer_i to timeout_i;
    (New2)
    coord_i ← ((r_i - 1) mod n) + 1;
      if (i = coord_i) then
        // 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[r_i] ≠ ∅) ∧ (timer_i expired)));
    (New3)
      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);
    (New4)
      wait_until (a message AUX[r_i]() has been received from (n - t) different processes);
      set timer_i to timeout_i;
    (M-07)
      wait_until ((messages AUX[r_i][b_val_i(1)], ..., AUX[r_i][b_val_i(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, where (i) values_i = ∪_{1≤x≤n-t} b_val_i(x)
        and (ii) values_i ⊆ bin_values_i[r_i]) ∧ (timer_i expired));
    (New5)
      if (when considering the whole set of the messages AUX[r_i]() received, several sets
        values_1, values_2, ... satisfy the previous wait predicate) ∧ (one of them is aux_i)
        then values_i ← aux_i; end if:
    (Opt2)
      if (decided in round r_i) then // the following are termination conditions
        wait until (bin_values[i][r_i] = {0, 1}) // only go to the next round when necessary
      else if (decided in round r_i - 2) then halt end if; // everyone has decided by now
      end if:
  end while.
```

• \( \text{timer}_i \) is a local timer, and \( \text{timeout}_i \) a timeout value, both used to exploit the assumption \( \Diamond \text{Synch} \).

• \( \text{coord}_i \) is the index of the current weak round coordinator.

• \( \text{aux}_i \) is an auxiliary set of values, used to store the value (if any) that the current weak coordinator strives to impose as decision value.

The weak coordinator of round \( r \), uses the message type \( \text{COORD\_VALUE}[r]() \) to broadcast the value it suggests for decision.

**Description of the extended algorithm.** We now list the new and modified lines that were added in Figure 2.

• At line New1, \( p_i \) waits until a value enters \( \text{bin\_values} \), then 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 Opt1 is an optimization only used along with the reduction to multivalued consensus presented in Section 4.

• Line New4 waits until \((n-t) \text{AUX}[r]()\) messages are received from different processes before resetting the timer, whose expiry is used in the predicate of the modified line M-07.

• Lines New2, New3, M-06, and New5 realize a mechanism that allows the current weak coordinator (whose value is computed on line New2) to try to impose the first value that enters into its \( \text{bin\_values} \) set as the decided value. 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 \( \text{values}_i \), which entails their decision.

• Modified lines M-05 and M-07: addition of the timer expiration in the predicate considered at the corresponding line.

• Line Opt2 is an optional optimization to minimize the amount of extra rounds processes need to execute after deciding. The first condition \((\text{wait until } (\text{bin\_values}_i[r_i] = \{0, 1\}))\) ensures that, after decision, a process only continues to the next round if some other non-faulty process did not decide in the current round. As this can only happen if both 0 and 1 enter \( \text{bin\_values} \), the process will not move on to the next round until this is true. The second condition, \((\text{if } (\text{decided in round } r_i - 2))\), halts the process 2 rounds after it has decided, as all non-faulty processes are guaranteed to have decided by this round.

The aforementioned modifications exploit the weak coordinator that only helps resolving disagreement by broadcasting a value that all non-faulty
adopt, as opposed to leaders or classic (strong) coordinators [17,22]. To this end:

- The weak coordinator $p_k$ broadcasts the message $\text{COORD\_VALUE}[r_i](w)$, where $w$ is the first value that enters its $\text{bin\_values}$ 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, so that all non-faulty processes will receive it before their timer expiration at line M-05.

- Then, assuming the previous item, all non-faulty processes set $\text{aux}_i$ to \{w\} (line New3), and broadcast it (line M-06). The predicate $w \in \text{bin\_values}_i[r_i]$ 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 $\text{AUX}[r_i](\{w\})$ from $(n - t)$ different processes, and, by line New5, will set $\text{values}_i = \{w\}$. This entails their decision during the round $(r + 1)$ or $(r + 2)$.

To ensure that slow processes catch up to faster processes that have reached later rounds, once a process has received at least $t + 1$ messages belonging to a round $r$, the process does wait for timeouts in rounds less than $r$. In the presence of $\Diamond \text{Synch}$, this ensures that all non-faulty processes eventually execute synchronous rounds. The proof of liveness of algorithm 2 is deferred to the appendix.

4 DBFT: From Binary Byzantine Consensus to Blockchain Consensus

This section presents a Democratic Binary Fault Tolerant algorithm, called DBFT. It relies on a reduction from the binary Byzantine consensus $\text{Psync}$ to the multivalue consensus and is also time optimal, resilience optimal and does not use classic (strong) coordinator, which means that it does not wait for a particular message. In addition, it finishes in only 4 messages delays in the good case, when all non-faulty processes propose the same value.

We consider a variant of the classical Byzantine consensus problem, called the Validity Predicate-based Byzantine Consensus (denoted VPBC). Its validity requirement relies on an application-specific valid() predicate that is used by blockchains to indicate whether a value is valid. 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.

- 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, i.e., it satisfies the predefined predicate denoted \( \text{valid}() \), and if all non-faulty processes propose the same value \( v \) then they decide \( v \).

This definition generalizes the classical definition of Byzantine consensus, which does not include the predicate \( \text{valid}() \). 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 or contains invalid transactions. There exist similar problem definitions whose validity also relies on the notion of a predicate. The validated Byzantine consensus \cite{13} differs in that the same valid value proposed by non-faulty processes has to be decided if all processes are non-faulty. The asynchronous Byzantine agreement \cite{29} defines a legal value similar to our valid value, however, its validity does not require a legal value to be decided if multiple ones exist, while we require that any decided value must be valid. A probabilistic variant \cite{14} required that the decided value be one of the proposed values, something we do not require.

```
operation mv_propose(v_i) is
(01) RB_broadcast VAL(v_i);
(02) repeat if (exists k : (proposals_i[k] ≠ ⊥) ∧
     \( \text{BIN}_\text{CONS}[k].\text{bin_propose}() \text{ not invoked} ))
     then invoke \( \text{BIN}_\text{CONS}[k].\text{bin_propose}(-1) \) end if;
(03) until (exists f : bin_decisions_i[f] = 1) end repeat;
(04) for each k s.t. \( \text{BIN}_\text{CONS}[k].\text{bin_propose}() \text{ not yet invoked} \)
    do invoke \( \text{BIN}_\text{CONS}[k].\text{bin_propose}(0) \) end for;
(05) wait_until (\( \bigwedge_{1 ≤ x ≤ n} \text{bin_decisions}_i[x] ≠ ⊥ \));
(06) \( j ← \min \{ x \text{ such that } \text{bin_decisions}_i[x] = 1 \} \);
(07) wait_until (proposals_i[j] ≠ ⊥);
(08) decide(proposals_i[j]).
(09) when VAL(v) is RB-delivered from p_j do
    if valid(v) then
        proposals_i[j] ← v;
        BV-deliver b-val[1](1) to BINCONS[j] end if.
(10) when \( \text{BIN}_\text{CONS}[k].\text{bin_propose}() \) decides a value b
    do \( \text{bin_decisions}_i[k] ← b \).
```

Figure 3: From multivalued to binary Byzantine consensus in \( BAMP_{n,t}[t < n/3, \text{BBC}] \)
Binary consensus objects. 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 3.4. To simplify the presentation, we consider that a process \( p_i \) launches its participation in \( BIN\_CONS[k] \) by invoking \( BIN\_CONS[k].bin\_propose(v) \), where \( v \in \{0, 1\} \). Then, it executes the corresponding code in a specific thread, which eventually returns the value decided by \( BIN\_CONS[k] \).

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 \( BIN\_CONS[k] \).

The algorithm. The algorithm reducing from the binary Byzantine consensus to multivalue Byzantine consensus is described in Figure 3 and is similar to an existing reduction [7], except that it combines the reliable broadcast, RB-broadcast [10], restated in the appendix, with our binary consensus messages to finish in 4 message delays in the good case. Initially, a process invokes the operation \( mv\_propose(v) \), where \( v \) is the value it proposes to the multivalued consensus. Process \( p_i \) executes four phases.

Phase 1: \( p_i \) disseminates its value (lines [01] and [11]). Process \( p_i \) first sends its value to all the processes by invoking the RB-broadcast operation (line [01]). If a process RB-delivers a valid value \( v \) RB-broadcast by a process \( p_k \), then the process stores it in \( proposals_i[j] \) and BV-delivers 1 directly to round one of instance \( BIN\_CONS[j] \) (line [11]), placing 1 in its \( bin\_values_i \) for that instance.

Phase 2: Process \( p_i \) starts participating in a first set of binary consensus instances (lines [02][04]). It enters a loop in which it starts participating in the binary consensus instances. Process \( p_i \) invokes a binary consensus instance \( k \) with value \( -1 \) for each value RB-broadcast by process \( p_k \) that \( p_i \) RB-delivered. \( -1 \) is a special value that allows the binary consensus to skip the \( BV\_broadcast \) step (line (Opt1)) and immediately send an \( AUX \) message with value 1, allowing the binary consensus to terminate with value 1 in a single
message delay. (Note that the timeout of the first round is set to 0 so the binary consensus proceeds as fast as possible.) The direct delivery of 1 into bin.values is possible due to an overlap in the properties of BV.broadcast and RB-broadcast, allowing us to skip a message step of our binary consensus algorithm. In other words, all non-faulty processes will RB-deliver the proposed value, and as a result will also BV-deliver 1. This loop stops as soon as $p_i$ discovers a binary consensus instance $BIN\_CONS[\ell]$ in which 1 was decided (line 01). (As all non-faulty processes will only have 1 in their bin.values until an instance terminates, the first instance to decide 1 will terminate in one message delay following the RB-delivery.)

**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 with bin.propose(0) 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 participation is to ensure that all binary consensus instances eventually terminate.

**Phase 4:** $p_i$ decides a value (lines 07-10 and 12). Process $p_i$ considers the first (according to the process index order) among the successful binary consensus objects, i.e., the ones that returned 1 (line 08). 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). Observe that this value is eventually RB-delivered by every non-faulty process. Consequently, $p_i$ decides it (lines 09-10). Notice that as soon as the binary consensus instance with the smallest process index terminates with 1, the reduction can return as soon as the associated value is RB-delivered. This is due to the observation that the values associated with the larger indices will not be used.

**Complexity.** This eager termination allows the consensus algorithm to terminate in 4 message delays in the good scenario, i.e., 3 message delays to execute the reliable broadcast and 1 to complete the binary consensus by skipping the BV.broadcast step. In this case the reliable broadcast and binary consensus each have $O(n^2)$ message complexity for a total of $O(n^3)$ including all $n$ instances. In the case of faulty processes or asynchrony the algorithm will need at least 3 additional message delays for binary consensus instances to terminate with 0.

**Theorem 1.** *The algorithm described in Figure 3 implements the multi-valued Byzantine consensus (VPBC) in the system model BAMP_{n,t}[t <
The proof of correctness of DBFT is deferred to the appendix.

5 Conclusion

To conclude, our weak coordinator based Byzantine consensus is time optimal, resilience optimal, does not rely on randomization or signatures and improves over the randomized Byzantine consensus algorithms \cite{37, 38} by terminating faster in various geo-distributed experiments. We presented how it can be used for consortium blockchains by generalizing the Byzantine consensus problem and presenting a solution that combines an existing reduction with our binary Byzantine consensus algorithm.

DBFT is now at the heart of the Red Belly Blockchain, a fast permissioned blockchain. Future work involves extending this permissioned blockchain into a public blockchain using DBFT for reconfiguration to periodically change at runtime the subset of machines running the consensus, similar to Solida \cite{1} but without proof-of-work.

Acknowledgments. We wish to thank Christian Cachin and Seth Gilbert for their constructive feedback on earlier versions of this paper and for pointing us to relevant papers.

References


[38] 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)


A  Experiments on 100 VMs on Distinct Continents

In this section, we evaluate the performance of our consensus algorithm against a randomized consensus applied to blockchains on 100 Amazon machines located in 5 distinct data centers across different continents.

A.1  Experimental setup

To measure the performance of our consensus algorithm in a real network setting, we deployed our binary consensus algorithm called “Psync” on 100 machines distributed across different continents.

To implement point-to-point reliable channels over the Internet, we implemented secure channels using TLS on top of TCP/IP. Note that TLS uses a public key cryptosystem (and signatures) only to exchange secret keys, but no signatures are used by our consensus algorithm. Note that the Red Belly Blockchain builds upon the same combination of DBFT and TLS by storing the necessary certificates in its blocks [25].

For the sake of comparison, we also implemented the randomized binary Byzantine consensus algorithm from Mostéfaoui et al. [37], called “Coin”, as a baseline. Coin terminates in $O(1)$ rounds in expectation and is at the heart of the HoneyBadger permissioned blockchain [36] but requires a fair scheduler [38]. Our implementation reuses the common coin implementation of HoneyBadger [36] that consists of a one step message exchange and threshold signatures. All 100 machines are c4.xlarge of Amazon EC2 equipped with an Intel Xeon E5-2666 v3 with 4 vCPUs, 7.5 GiB RAM, and “moderate” network performance.

We set the timeouts of Psync to be null in the first $t$ rounds before incrementing exponentially. We implemented reliability using sequence numbers and negative acknowledgments at the application level. All consensus decisions are stored to disk in an append only log. Results are taken as the average of 100 instances of consensus.

A.2  Geo-distributed experiments between 5 datacenters

Figure 4 compares the average latency and number of rounds needed to terminate Psync and Coin in 5 Amazon datacenters, 3 in the US (Oregon, Northern California, and Ohio) and 2 in Europe (Ireland and Frankfurt). Our ping latency across continents is between 91 ms and 164 ms and within one continent between 22 ms and 71 ms. In Figure 4(left) the x-axis denotes
the approximate percentage of processes that have an initial proposal of 0 (others proposing 1). Psync terminates in at most three rounds on average.

Given that Psync is designed to terminate with 1 in the first round and 0 in the second round, the best performance is reached when the majority of proposals are 1. In all cases the latency of Psync is lower than Coin due to the coin needing an extra message step, additional computation complexity, and randomness.

A.3 Tolerance to various Byzantine attacks

Figure 4(right) compares the algorithms with the following Byzantine behaviors: (Byz1) Byzantine processes flip the binary values of their messages; (Byz2) Byzantine processes are mute; (Byz3) extends Byz1 with Byzantine coordinators that send random binary values in their COORD_VALUE messages; (Byz4) Byzantine processes form a coalition to limit the progress within rounds by sending their own messages without waiting so they can be processed before others. Both Byz3 and Byz4 are specific to Psync.

More precisely, Byz4 mimics a behavior where the coordinator is faulty to limit progress during rounds by trying to have (i) no non-faulty processes to decide in round \( r \) and (ii) have two non-faulty processes starting round \( r + 1 \) with distinct estimates. To this end, the faulty nodes start the round by broadcasting both 1 and 0 in their BV-broadcast. Then, the Byzantine coordinator sends a message coord.value with \(- (r \mod 2)\) to all non-faulty nodes. Finally, Byzantine nodes instantly send AUX message with value
to a single node and send AUX message with value \((r \mod 2)\) to the remaining nodes. Faulty nodes in Byz4 have the power to send their messages instantly and to observe the messages received at non-faulty nodes, giving them more power to delay termination. They do not control the speed or order of messages from non-faulty nodes.

In Psync, the Byzantine processes are chosen as the first \(t\) coordinators. Coin has the highest latency with the Byzantine behaviors, but its number of rounds is least affected. Byzantine behavior Byz3 is the slowest to terminate for Psync because it allows Byzantine processes to force the most disagreement. While theoretically Byz4 could always prevent termination in the first \(t\) rounds, the average number of rounds is only increased to 6 (but has a maximum of 35). This is due to the fact that they do not control the speed of messages of non-faulty processes in the network preventing the non-terminating case. Furthermore, given that the Byzantine processes have to act fast to ensure their messages are processed first, the average latency is lower than Byz1 and Byz2.

### A.4 Detailed description of Byzantine behavior Byz4

In the presence of a faulty coordinator it is possible to execute repeated rounds in which there is no termination, behavior Byz4 tries to capture this behavior. Note that we allow Byzantine messages to be delivered instantly by computing them directly at the non-faulty nodes when needed. We will now describe the Byz4 behavior. Assume we are in a round \(r\). There are two main things we need to ensure: (i) no non-faulty process decides in round \(r\) (ii) at least one non-faulty node must start round \(r+1\) with an estimate of 0 and another start with the estimate of 1.

To ensure (i) we need (a) \(\neg (r \mod 2)\) to enter bin\_values\_nodes and (b) no node must receive \(n-t\) AUX messages with value \((r \mod 2)\). Then to ensure (ii) we need (c) both 0 and 1 to enter bin\_values\_nodes, (d) at least one non-faulty node must receive \(n-t\) AUX messages with value \(\neg (r \mod 2)\), and (e) at least one node must receive an AUX messages with value \((r \mod 2)\).

Thus, Byzantine nodes start the round by broadcasting both 1 and 0 in their BV-broadcast to ensure (a) and (c). To try to ensure (b), the Byzantine coordinator sends a message coord\_value with \(\neg (r \mod 2)\) to all non-faulty nodes, this message is delivered instantly, as a result all non-faulty processes broadcast an AUX message with value \(\neg (r \mod 2)\). Then to ensure (d), Byzantine nodes instantly send AUX message with value \(\neg (r \mod 2)\) to a single node. Furthermore, to ensure (e), Byzantine nodes instantly send AUX
message with value \((r \mod 2)\) to the remaining nodes. Assuming both 0 and 1 entered \textit{bin\_values} at appropriate times at non-faulty nodes, termination will be prevented for this round.

The difficulty in ensuring this non-termination scenario is that the Byzantine nodes do not control the time that both 1 and 0 enter \textit{bin\_values} of non-faulty nodes. If \(\neg(r \mod 2)\) enters too late, a process may broadcast \((r \mod 2)\) as its \textit{aux} message, and as a result we may fail with (d). Otherwise if \((r \mod 2)\) enters \textit{bin\_values} too late, all non-faulty processes may terminate with \(n - t\) \textit{aux} messages with value \(\neg(r \mod 2)\). Similar timing arguments can be made for other non-terminating scenarios that use different message patterns.

\section*{A.5 Different experiment configurations}

Figure 5 uses the same experimental settings as Figure 4, except is run with 100 nodes within a single datacenter. Here we see a much larger gap in latency between Psync and Coin as the computation of the cryptographic operations of the random coin is much larger than the network latency. Note that the latency of both algorithms could be decreased through the use of message authentication codes (MACs) with datagram broadcasts, but we expect the latency to still be dominated by the cryptographic operations of the coin.

Figure 5 uses the same experimental settings as Figure 4, except is run with 1 node in each of Amazon’s 14 EC2 data centers. The results are similar to the 5 datacenter case of Figure 4, but with higher latency in most cases due to the increased geo-distribution.

\section*{B Proofs of safety and liveness of the algorithms}

\subsection*{B.1 Safety proof of the binary Byzantine consensus (Figure 1)}

The proof is described from a point of view of a non-faulty process \(p_i\). Let \(values^r_i\) denote the value of the set \textit{values}_i which satisfies the predicate of line 07 during a round \(r\). Moreover, let us recall that, given a run, \(C\) denotes the set of non-faulty processes in this run.

\begin{lemma}
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.
\end{lemma}
Figure 5: Single datacenter comparison of latency and average number of rounds to terminate of our deterministic binary Byzantine consensus against randomized binary Byzantine consensus: (Left)

**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 $est[r](v)$ 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 $aux[r](\{v\})$ 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 $v$.

\[\Box\text{Lemma 1}\]

**Lemma 2.** Let $t < n/3$. $((p_i, p_j \in C) \land (values_i^r = \{v\}) \land (values_j^r = \{w\})) \Rightarrow (v = w)$.

**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 $aux[r](\{v\})$ 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 $aux[r](\{v\})$ 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 $aux[r](\{w\})$ 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 $aux[r](\{\})$ to $p_i$ and $p_j$, and we consequently have $v = w$. \[\Box\text{Lemma 2}\]

25
Lemma 3. 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 \( \text{bin\_values}_i[1] \) 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 (\( values_1^i \subseteq \text{bin\_values}_i[1] \)), and the BV-Justification property of the BV-broadcast abstraction, that the set \( values_1^i \) 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.

Lemma 4. 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 \( v = (r \mod 2) \).

If another non-faulty process \( p_j \) decides during round \( r \), we have \( values_j^r = \{ w \} \), and, due to Lemma 2, 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 \( v = (r \mod 2) \) 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 2 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 \( (r \mod 2) = v \) to its local estimate \( est_j \).

It follows that all non-faulty processes start round \( (r + 1) \) with the same local estimate \( v = r \mod 2 \). Due to Lemma 3, 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.

Lemma 5. Let the system model be \( \text{BAMP}_{n,t}[t < n/3] \). 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 $\text{aux}[r]()$ (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 $r$. □Lemma 5

**Lemma 6.** Let the system model be $\text{BAMP}_{n,t}[t < n/3]$. If all non-faulty processes $p_i$ terminate a round $r$ with $\text{values}_r = \{v\}$, they all decide by round $(r + 1)$.

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

If $r$ is such that $v \neq (r \mod 2)$, each non-faulty process sets its current estimate to $v$ (line 10). As during the next round we have $v = ((r + 1) \mod 2)$, and $\text{values}_{r+1} = \text{bin.values}_r[r + 1] = \{v\}$ at each non-faulty process $p_i$, each non-faulty process decides during round $(r + 1)$. □Lemma 6

**Lemma 7.** Let the system model be $\text{BAMP}_{n,t}[t < n/3]$. If every non-faulty process $p_i$ terminates a round $r$ with $\text{values}_r = \{0, 1\}$, then it decides by round $(r + 2)$.

**Proof** If every non-faulty processes $p_i$ is such that $\text{values}_r = \{0, 1\}$, it executes line 11 during round $r$, and we have $\text{est}_i = (r \mod 2) = v$ when it starts round $(r + 1)$. Due to Lemma 1, it keeps this estimate forever. As all non-faulty processes execute rounds $(r + 1)$ and $(r + 2)$ (Lemma 5), and $v = ((r + 2) \mod 2)$, we have $\text{values}_{r+2} = \{v\}$, at each non-faulty process $p_i$. It follows that each non-faulty process decides at line 10. □Lemma 7

**Theorem 2.** The algorithm described in Figure 1 satisfies the safety consensus properties.

**Proof** The proof follows from Lemma 3 (BBC-Validity) and Lemma 4 (BBC-Agreement). □Theorem 2

**Decision** The algorithm described in Figure 1 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.

### B.2 Why the safe algorithm does not terminate with \( \diamond Synch \)

To circumvent the consensus impossibility [24] and find a terminating solution, one could be tempted to consider the \( BAMP_{n,t}[t < n/3, \diamond Synch] \) model and setting a timer, that increases in each round, by replacing line 05 in Figure 1 with a new line called “New1” and a modified line 05 called “M-05”:

```plaintext
... (New1) timeout_i ← timeout_i + 1; set timer_i to timeout_i;
(M-05) wait_until ((bin_values_i[r] ≠ ∅) ∧ (timer_i expired));
... 
```

In fact, this could seem sufficient to eventually give enough time for messages to be delivered. As we explain below, it would still be possible for a Byzantine process to wait depending on the timer of the current round to send a message \( BVAL(v) \) to a non-faulty process early enough so that this non-faulty process receives the message before its local timer expires but too late for this non-faulty process to rebroadcast it and for other non-faulty processes to deliver it before their timers expire.

As an example, consider a counter-example of \( n = 4 \) processes among which \( t = 1 \) process is Byzantine that starts from a round \( r \) such that \( r \mod 2 = 1 \) with non-faulty processes with estimates 0, 0 and 1. There is an execution leading to a round \( r + 1 \) where \( (r + 1) \mod 2 = 0 \) and non-faulty processes have estimates 0, 1 and 1. The symmetric of this counter-example can then be used from round \( r + 1 \) where non-faulty processes have estimates 0, 1 and 1 to round \( r + 2 \) where non-faulty processes have estimates 0, 0 and 1. An infinite sequence alternating this counter-example and its symmetric example illustrates an infinite execution where no non-faulty process decides.

The counter-example is represented as a distributed execution in Figure 6 where \( p_1, p_2 \) and \( p_3 \) are non-faulty processes while \( p_4 \) is a Byzantine process, as a distributed execution where time increases from left to right.

---

1In the case of the randomized binary consensus algorithm of [38], the common coin guarantees termination with probability 1, because eventually the singleton value in \( values_i \), will match the coin.
Figure 6: An execution of $n = 4$ processes exchanging broadcast messages (represented by triangles between $n - t = 3$ non-faulty processes) in round $r$ ($r \mod 2 = 1$) where $est_1 = est_2 = 0$ and $est_3 = 1$ leading to a round $r + 1$ where $est_1 = 0$ and $est_2 = est_3 = 1$

where arrows represent messages sent by the Byzantine process $p_4$ and triangles represent the broadcast messages among non-faulty processes: the left angle of each triangle indicates the source of the broadcast while the right edge indicates the processes where messages are delivered. (The receipt of messages by the Byzantine process $p_4$ are omitted for the sake of clarity in the presentation.)

The first four triangles represent the BV-broadcast (Figure 7) where $p_1$ and $p_2$ broadcasts $BVAL(0)$ while $p_3$ broadcasts $BVAL(1)$ according to their initial estimates. Once $p_3$ delivers $BVAL(0)$ from $t + 1 = 2$ non-faulty processes, it broadcasts the value 0 that it never broadcast before as specified in the code of Figure 7.

During BV-broadcast, all non-faulty processes receive from $2t + 1 = 3$ non-faulty processes. Now consider that the Byzantine process $p_4$ sends $BVAL(1)$ to $p_2$ which makes $p_2$ rebroadcast it as part of the BV-broadcast because it has now received message $BVAL(1)$ from $2 = t + 1$ distinct processes, namely $p_3$ and $p_4$. We can thus obtain that $bin\_values_1 = \{0\}$, $bin\_values_2 = \{0, 1\}$ and $bin\_values_3 = \{0\}$ at the time non-faulty processes broadcast their AUX messages. By sending $AUX(\{0\})$ to $p_1$, the Byzantine process $p_4$ allows $p_1$ to choose $values_1 = \{0\}$ that has received $AUX(\{0\})$ from $n - t = 3$ distinct processes ($p_1$, $p_3$ and $p_4$), while the others have to choose $values_2 = values_3 = \{0, 1\}$ as they receive $AUX(\{0\})$, $AUX(\{0, 1\})$, $AUX(\{0\})$ from $p_1$, $p_2$ and $p_3$, respectively. As $b = r \mod 2 = 1$, it results
from line 11 that \( p_1, p_2 \) and \( p_3 \) have estimates 0, 1, 1, respectively when starting the round \( r + 1 \).

Applying the symmetric example would lead to round \( r + 2 \) with the same estimates 0, 0, 1 as in round \( r \), indicating the existence of an infinite execution.

### B.3 Proof of Safety and Liveness of the \( \Diamond \) Synch-based Binary Byzantine Consensus (Figure 2)

The proof consists of two parts: (i) show that the added statements preserve the consensus safety properties proved for the algorithm of Figure 1, and (ii) show that all non-faulty processes eventually decide.

**Lemma 8.** The algorithm described in Figure 2 satisfies the BBC-Validity and BBC-Agreement properties.

**Proof** The proof consists in showing that the Lemmas 1, 2, 3 and 4 remain correct when considering the algorithm of Figure 2. Basically, these proofs remain correct because, as the new and modified statements do not assign values to the sets \( \text{bin\_values}_i[r] \) at the non-faulty processes, and no property of \( \text{bin\_values}_i \) is related to a timing assumption, the set \( \text{bin\_values}_i[r] \) 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 \( \text{est}_i \) and \( \text{values}_i \) of any non-faulty process \( p_i \) (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 1** Let \( r \) be the considered round, and \( v \) be the current estimate of the non-faulty processes. We then have \( \text{bin\_values}_i[r] = \{ v \} \) at line M-05 of every non-faulty process \( p_i \).
  
  - If the weak round coordinator \( p_k \) is non-faulty, we have at every non-faulty process \( \text{aux}_i = \text{bin\_values}_i[r] = \{ v \} \). It then follows that \( \text{values}_i^r = \{ v \} \) and the lemma remains true due to lines 09 and 10.
  
  - If the weak 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 \( \text{COORD\_VALUE}[r](\{1 - v\}) \). As \((1 - v) \notin \text{bin\_values}_i[r] \), at line New3, \( p_i \) executes the “else” part where it sets \( \text{aux}_i \) to \( \{ v \} \) (the only value in \( \text{bin\_values}_i[r] \)), and the lemma follows.
Lemma 2. 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 3. 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 4. As it relies only on the set $values_i^r$ of each non-faulty process $p_i$, this proof remains correct.

Lemma 9. The algorithm described in Figure 2 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 5 remains true. The reader can also check that the proof of Lemma 6 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 6). The proof shows that, due to (a) the eventual synchrony assumption, (b) the weak round coordinator mechanism, and (c) the messages $COORD\_VALUE[r](\cdot)$ sent by the weak round coordinators, there is a round $r$ such that $values_i^r = \{w\}$ at each non-faulty process.

Let us consider a time $\tau$ from which (due to Lemma 11) 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 $\tau$. 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 New3. 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 6 all non-faulty processes decide $w$ by round $r+1$, which concludes the proof of the lemma.

Theorem 3. The algorithm described in Figure 2 solves the binary Byzantine consensus in the system model $BAMP_{n,t}[t < n/3, \Diamond Synch]$. 

31
**Proof**  The proof follows directly from Lemma 8 (BBC-Validity and BBC-Agreement) and Lemma 9 (BBC-Termination).

**Theorem**

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 (as assumed by Lemma 9). 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. It is assumed that non-faulty processes may observe time at different rates and processing time is non-negligible, but is bounded by some unknown constant. By using a timeout that grows by 1 each round the following proof shows that all processes eventually reach a round from which they behave synchronously.

For the proof we will need to use a mini-round notation and a catch-up mechanism.

- **Mini-round**: Each round $r$ is split into two mini-rounds, with the first mini-round representing lines 03 to M-05 and the second representing lines (New3) to 12. Thus, round 0 is made up of mini-rounds 0 and 1, round 1 is made up of mini-rounds 2 and 3, and so on. The reason behind splitting the rounds is so that each mini-round includes a single execution of the timer.

- **Catch-up mechanism**: A catch-up mechanism is used to help the slow non-faulty processes to catch up to the most advanced non-faulty processes (as measured by their mini-round number). To this end, when a process is in a mini-round $\rho$ and receives messages corresponding to another mini-round $\rho'$ from $(t + 1)$ different processes (i.e., from at least one non-faulty process) such that $\rho' > \rho$, the process no longer waits for timers in mini-rounds $\rho$, ..., $(\rho' - 1)$. It still completes these mini-rounds, but does so without waiting for timers expiration.

We assume that each process has a local clock that allows it to measure time units as 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

\[^2\text{Similar mechanisms are used by PBFT [16].}\]
notation \( t \) with a subscript (for example \( t_{\text{first}} \)) 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 \( \diamond \text{Synch} \), processes are able to observe time at different rates, but within an unknown fixed bound. For simplicity we assume that the fastest non-faulty process observes time at a rate no faster than observed by the global observer \( G \), thus all other processes observe time at this rate or slower. The timeouts used in the following proof are relative to the timeouts of the fastest process.

**Definitions**  The following definitions will be used in the proofs.

- \( \delta \) is a fixed, but unknown bound on message transfer delays as ensured by \( \diamond \text{Synch} \) and measured in time units as observed by \( G \).
- \( t_{\text{first}} \) is the time, as measured by \( G \), at which the first non-faulty process \( p_{\text{first}} \) reaches mini-round \( \rho \) (\( t_{\text{first}} \) is the time at which the first non-faulty process starts the consensus).
- \( t_{\text{last}} \) is the time, as measured by \( G \), at which the last (i.e. the slowest for that mini-round) non-faulty process \( p_{\text{last}} \) reaches mini-round \( \rho \) (\( t_{\text{last}} \) is the time at which the last non-faulty process starts the consensus).
- \( \theta_{\text{fast}} \) (resp. \( \theta_{\text{slow}} \)) is the minimum (resp. maximum) amount of time, as observed by \( G \), for any process to perform the computation of any mini-round (an unknown bounded difference between \( \theta_{\text{fast}} \) and \( \theta_{\text{slow}} \) is ensured by \( \diamond \text{Synch} \)).
- \( \gamma_{\text{fast}} \) is the minimum amount of time, as observed by \( G \), in a mini-round \( \rho \) that any process waits on line New1 or New4 before starting its timer for that mini-round.
- Mini-round \( \rho_{\delta} \) is the first mini-round where \( \text{timeout} > \delta \) at any non-faulty process.

The proof is made up of two lemmas. Lemma 10 shows that processes will eventually reach a point where they remain no more than one mini-round apart. Lemma 11 builds upon this to show that the rounds eventually become synchronous.

**Lemma 10.** Consider the algorithm of Figure 2 enriched with the previous catch-up mechanism. There is a mini-round \( \rho_{\delta} \) such that in \( \rho_{\delta} \) and for 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 mini-round larger
than $\rho_t$ until after they start waiting for the timeout of mini-round $\rho_t$.

**Proof** Let us consider mini-round $\rho_t$ where $\rho_t > \rho_\delta$. For all non-faulty
processes to wait at a timeout in a mini-round $\rho_t$, the last non-faulty process
to arrive at $\rho_t$ must arrive before it receives a message from some other non-
faulty process that has already started executing a later mini-round (note
that given $\rho_t > \rho_\delta$, this can only occur when the non-faulty processes are no
more than 1 mini-round apart). Thus, to satisfy the lemma, a mini-round is
needed where the following inequality holds at that and all following mini-
rounds:

$$t_{\text{last}, \rho_t} < t_{\text{first}, \rho_{t+1}}.$$  (1)

To find out when this is satisfied first we will compute the minimum and
maximum times at which non-faulty processes can arrive at a mini-round.
By definition, a non-faulty process can spend no less time than $(\gamma_{\text{fast}} + \theta_{\text{fast}} + \text{timeout})$ in a mini-round $\rho'$. Given that timeouts start with value 0
in mini-round 0 and grows by 1 in each mini-round, timeout can be replaced
with $\rho$ for any mini-round $\rho$ as a lower bound for the fastest process. We
can then compute the time where the first non-faulty process arrives at
mini-round $\rho'$ (where $\rho' > \rho_\delta$) as:

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

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

Now consider how long it will take the slowest non-faulty process to
execute mini-round $\rho'$ when it does not wait at a timeout. By definition we
know the process will spend no more time than $\theta_{\text{slow}}$ on computation. Thus,
the remaining time will be spent waiting until the wait_until() conditions in
the algorithm are satisfied. We will now examine how much time a non-
faulty process can spend waiting during a mini-round on either line M-05 or
M-07 (we only consider these wait_until() conditions as they encompass the
others within a mini-round).

First consider line M-05. Its condition requires $(\text{bin} \_\text{values}[i] \neq \bot)$. Given that the process is not waiting at a timeout, it must have received
$(t + 1)$ messages corresponding to a later mini-round, meaning that some
non-faulty process has already completed $\rho'$. Furthermore, given that this is
the slowest non-faulty process, we know that all non-faulty processes have already executed the BV\_broadcast() operation on line 04. As we can see in Figure 7 in the BV\_broadcast() operation all non-faulty processes will perform at most 2 broadcast operations. Thus, by the BV-Uniformity property, all non-faulty processes will have a value in their bin\_values_i[ri] after at most 2 message delays following the slowest non-faulty processes invocation of the BV\_broadcast(). As a result, the process takes at most $2 \times \delta + \theta_{\text{slow}}$ time to execute the mini-round.

Now consider line M[07]. By the time the slowest non-faulty process has reached this line all non-faulty processes have broadcast their aux messages, thus the slowest non-faulty process will receive these aux messages in at most $\delta$ time. The process may then need to wait for another message delay to satisfy all the conditions of line M[07] in the case where a non-faulty process had a value enter its bin\_values_i[ri] immediately before broadcasting its aux message (recall that the BV\_broadcast() may take up to 2 message delays). Thus, as before, the process takes at most $2 \times \delta + \theta_{\text{slow}}$ time to execute the mini-round.

We then have:

$$t_{\text{last}_{\rho'}} \leq t_{\text{last}_s} + \left( \sum_{x=s}^{\rho'-1} 2 \times \delta + \theta_{\text{slow}} \right).$$

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

Now given $t_{\text{first}_{\rho'}}$ is quadratic while $t_{\text{last}_{\rho'}}$ is linear, inequality (1) must eventually be satisfied and there will be a mini-round where all non-faulty processes wait for at least part of their timeout.

It will now be shown that for mini-rounds where timeout > $(3 \times \delta + \theta_{\text{slow}})$, once inequality (1) is true, it will remain true for all following mini-rounds. This will be done by induction. Consider $t_{\text{last}_{\rho t}} < t_{\text{first}_{\rho t+1}}$ is satisfied, let us now show that $t_{\text{last}_{\rho t+1}} < t_{\text{first}_{\rho t+2}}$ must also be satisfied. For this to not hold, the slowest non-faulty process must spend more time on mini-round $\rho_t$ than the fastest non-faulty process spends on mini-round $(\rho_t + 1)$, but this is impossible because once the fastest process completes the condition on line New1 or New4 and starts its timer, $p_{\text{last}_{\rho t}}$ must receive $(t + 1)$ messages from mini-round $(\rho_t + 1)$ after $\delta$ time. Once these messages are received, the process will not wait at any timeout, and as we have already seen, the this process will take no more than $2 \times \delta + \theta_{\text{slow}}$ time to complete the mini-round. Thus, as long as timeout > $(3 \times \delta + \theta_{\text{slow}})$, which will eventually be true given $\Diamond \text{Synch}$ and the growing timeout, process $p_{\text{last}_{\rho t}}$ will reach mini-round $(\rho_t + 1)$ before $p_{\text{first}_{\rho t+1}}$ reaches mini-round $(\rho_t + 2)$.

\begin{lemma}
\end{lemma}
Lemma 11. Consider the algorithm of Figure 2 enriched with the previous catch-up mechanism. Eventually the non-faulty processes attain a mini-round from which they behave synchronously.

Proof By Lemma 10 it is known that there exists a mini-round $\rho$ where at that and all following mini-rounds all non-faulty processes wait for at least part of their timeout. Additionally, this must happen at some mini-round where $\text{timeout} > (3 \times \delta + \theta_{\text{slow}})$. Consider we are in such mini-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 their 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 mini-round is the weak coordinator, it may take up to 3 message delays before its $\text{COORD\_VALUE}[r]()$ message is received by all non-faulty processes (this includes up to 2 message delays until a value enters its $\text{bin\_values}[r]$ and an additional message delay to broadcast $\text{COORD\_VALUE}[r]()$). Thus, for a mini-round $\rho'$ to be synchronous where $\rho' \geq \rho$, the following needs to be ensured:

$$t_{\text{last}, \rho'} + (3 \times \delta) + \theta_{\text{slow}} \leq t_{\text{first}, \rho'} + \gamma_{\text{fast}, \rho'} + \text{timeout}_{\rho'}.$$  \hspace{1cm} (2)

Let us now compute $t_{\text{last}, \rho'}$. First, notice that before a non-faulty process starts its timer for a mini-round it must wait until the condition on line New1 or New4 is satisfied. Also note that by time $(t_{\text{first}, \rho'} + \gamma_{\text{fast}, \rho'} + \delta_{\text{fast}})$ at least one process has satisfied the condition on line New1 or New4 (this is given by the definition of $\gamma$). As a result all processes will receive $(t + 1)$ messages from mini-round $\rho'$ by time $(t_{\text{first}, \rho'} + \gamma_{\text{fast}, \rho'} + \delta_{\text{fast}})$. Now given Lemma 10 and that $(\rho' - 1) > \delta$, it is known that that the slowest process is no further behind than waiting at the timeout of mini-round $(\rho' - 1)$. After getting these $(t + 1)$ messages from mini-round $\rho'$ the slow process will then skip the timeout of mini-round $(\rho' - 1)$ 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 10 to satisfy the condition line M-05 or M-07) plus any processing time. Thus, the time at which the slowest process reaches mini-round $\rho'$ is given by:

$$t_{\text{last}, \rho'} \leq t_{\text{first}, \rho'} + \gamma_{\text{fast}, \rho'} + \delta_{\text{fast}} + \theta_{\text{slow}} + (3 \times \delta).$$

Now plugging this into inequality (2) leads to $\text{timeout}_{\rho'} \geq (7 \times \delta) + (2 \times \theta_{\text{slow}}) + \delta_{\text{fast}}$ (note that $2 \times \theta_{\text{slow}}$ is included to account for possible processing time).
times in both mini-rounds \((\rho_i' - 1)\) and \(\rho'_t\). But given that the timeout grows in each mini-round and that \(\delta, \theta_{fast}, \) and \(\theta_{slow}\) 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 10 holds then the above argument is valid for any mini-round. Now given that \(timeout \geq (7 \times \delta) + (2 \times \theta_{slow}) + \theta_{fast}\) is larger than the timeout needed for Lemma 10 to hold for every following mini-round, once inequality 2, i.e. synchrony, it true for one mini-round, it will also hold for every following mini-round.

\[ \Box_{Lemma 13} \]

### B.4 Proof of the Blockchain Consensus (Figure 3)

**Lemma 12.** There is at least one binary consensus instance that decides value 1, and all non-faulty processes exit the repeat loop.

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 \(proposals_i[j] \neq \bot\) at each non-faulty process \(p_i\) (line 11).

It follows from the first sub-predicate of line 02 and the RB-Termination-2 property that all non-faulty processes \(p_i\) invokes \(bin\_propose(-1)\) on the BBC object \(BIN\_CONS[j]\) and by line 11 they all BV-deliver 1 to round one. Notice that by using the RB-delivery to trigger the BV-delivery of 1 (instead of calling \(BV\_broadcast\)) the lemma relies on the fact that the properties of \(RB\_broadcast\) also ensure the properties of \(BV\_broadcast\). Namely that RB-Termination-1 ensures BV-Obligation, RB-Validity ensures BV-Justification, and RB-Termination-2 ensures BV-Uniformity and BV-Termination. It follows that the properties of the binary consensus are maintained. Hence, from its BBC-Termination, BBC-Agreement, BBC-Validity, and Intrusion-tolerance properties (as no non-faulty process has proposed 0), this BBC instance returns the value 1 to all non-faulty processes, which exit the “repeat” loop.

\[ \Box_{Lemma 12} \]
Lemma 13. A decided value is a valid value (i.e., it satisfies the predicate \textit{valid}).

**Proof** Let us first observe that, for a value \textit{proposals}_i[j] to be decided by a process \textit{p}_i, we need to have \textit{bin\_decisions}_i[j] = 1 (lines 08-10).

If the value 1 is decided by \textit{BIN\_CONS}[j], \textit{bin\_decisions}_i[j] = 1 is eventually true at each non-faulty process \textit{p}_i (line 12). If follows from (i) the fact that the value 1 can only enter the \textit{bin\_values} of a BBC instance after validation at line 11, and (ii) the Intrusion-tolerance property of \textit{BIN\_CONS}[j], that at least one non-faulty process \textit{p}_i inserted 1 into its \textit{bin\_values} on line 12. Due to line 11 it follows that \textit{proposals}_i[j] contains a valid value. □

Lemma 14. No two non-faulty processes decide different values.

**Proof** Let us consider any two non-faulty processes \textit{p}_i and \textit{p}_j, such that \textit{p}_i decides \textit{proposals}_i[k1] and \textit{p}_j decides \textit{proposals}_j[k2]. It follows from line 08 that \( k1 = \min\{x \text{ such that } \textit{bin\_decisions}_i[x] = 1\} \) and \( k2 = \min\{x \text{ such that } \textit{bin\_decisions}_j[x] = 1\} \).

On the one hand, it follows from line 07 that \( (\bigwedge_{1 \leq x \leq n} \textit{bin\_decisions}_i[x] \neq \bot) \) and \( (\bigwedge_{1 \leq x \leq n} \textit{bin\_decisions}_j[x] \neq \bot) \), from which we conclude that both \textit{p}_i and \textit{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 \( \forall x : \textit{bin\_decisions}_i[x] = \textit{bin\_decisions}_j[x] \). Let \( \textit{dec}_i = \textit{bin\_decisions}_i[x] = \textit{bin\_decisions}_j[x] \). It follows then from line 08 that \( k1 = k2 = \min\{x \text{ such that } \textit{dec}_i = 1\} = k \). Hence, \( \textit{dec}_i[k] = 1 \).

On the other hand, it follows from the Intrusion-tolerance property of \textit{BIN\_CONS}[k] that a non-faulty process \textit{p}_k inserted 1 into its \textit{bin\_values} on line 12. As this invocation can be issued only at line 03, we conclude (from the predicate of line 02) that \( \textit{proposals}_i[k] = v \neq \bot \). As \textit{p}_k is non-faulty, it follows from the RB-Unicity and RB-Termination-2 properties that all non-faulty processes RB-delivers \( v \) from \textit{p}_k. Hence, we eventually have \( \textit{proposals}_i[k] = \textit{proposals}_j[k] \), which concludes the proof of the lemma. □

Lemma 15. Every non-faulty process decides a value.

**Proof** It follows from Lemma 12 that there is some \textit{p}_j such that we eventually have \textit{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 14, the predicate of line 09 is eventually satisfied at each non-faulty process, which concludes the proof of the lemma.

**Theorem 4.** The algorithm described in Figure 3 implements multivalued Byzantine consensus (VPBC) in the system model \( \text{BAMP}_{n,t}[t < n/3, \text{BBC}] \).

**Proof** Follows from Lemma 13 (VPBC-Validity), Lemma 14 (VPBC-Agreement), and Lemma 15 (VPBC-Termination).

### B.5 Complexity

The proposed reduction has constant time complexity.

**Lemma 16.** When \( \text{RB} \_\text{broadcast} \) is invoked by a non-faulty process, all non-faulty processes \( \text{RB} \_\text{deliver} \) the value in constant time.

**Proof** Within \( \text{RB} \_\text{broadcast} \), a process starts by calling \( \text{broadcast} \) with a value, which will then be echoed by all non-faulty processes, resulting in all processes delivering the value in 3 communication steps.

**Lemma 17.** For any process that \( \text{RB} \_\text{deliver} \) a value, within a constant amount of time following this all non-faulty processes have \( \text{RB} \_\text{deliver} \) the value.

**Proof** For a process to \( \text{RB} \_\text{deliver} \) a value it must have received the value from \( n - t \) processes, thus \( t + 1 \) non-faulty processes have \( \text{broadcast} \) this value, which means all non-faulty will echo that value and all non-faulty will receive the value from \( n - t \) processes in at most 2 communication steps following the first \( \text{RB} \_\text{deliver} \).

**Theorem 5.** The reduction presented in figure 3 is a constant time reduction.

**Proof** Let us show this by contradiction. Assume a non-faulty process does not decide in constant time, there are two possibilities how this could happen, either: (i) the process waits on line 09 for more than constant time, or (ii) the process has not invoked some instance \( \text{BIN} \_\text{CONS}[k] \) until after constant time had already passed.
First consider (i). Here the process is waiting to \texttt{RB\_deliver} a value that by lemma 13 has already been \texttt{RB\_deliver}d by some non-faulty process, but by lemma 17 we know this must happen in constant time.

Now consider (ii). By lemma 12 we know that at least one binary consensus instance decides 1. Once this happens all non-faulty processes invoke all remaining instances of \texttt{BIN\_CONS}[k] without waiting. Thus, for (ii) to be true, no instance of \texttt{BIN\_CONS}[k] must have terminated with 1 in constant time. But given that at least \(2t+1\) instances of \texttt{RB\_broadcast} will be invoked by non-faulty processes, and given lemma 16 all non-faulty processes will invoke all instances of \texttt{BIN\_CONS}[k] in constant time on line 03 or 06.

\(\square\) Theorem 5

C The BV-broadcast all-to-all communication implementation

Figure 7 depicts the pseudocode of an existing implementation 38 of the BV-broadcast problem stated in Section 3.1.

\begin{verbatim}
operation BV\_broadcast \texttt{msg(v_i)} is
(01) broadcast \texttt{b\_val(v_i)}.

when \texttt{b\_val(v)} is received
(02) if (\texttt{b\_val(v)} received from \((t+1)\) different processes and \texttt{b\_val(v)} not yet broadcast)
(03) then broadcast \texttt{b\_val(v)} // a process echoes a value only once
(04) if (\texttt{b\_val(v)} received from \((2t+1)\) different processes)
(05) then bin\_values, \leftarrow bin\_values, \cup \{v\}. // local delivery of a value
\end{verbatim}

Figure 7: An algorithm implementing BV-broadcast in \(BAMP_{n,t}\) \(t < n/3\) (from [37])

D Reliable broadcast in Byzantine systems

This broadcast abstraction (in short, RB-broadcast) was proposed by G. Bracha 10. It is a one-shot one-to-all communication abstraction, which provides processes with two operations denoted \texttt{RB\_broadcast()} and \texttt{RB\_deliver()}. When \(p_i\) invokes the operation \texttt{RB\_broadcast()} (resp., \texttt{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 [11] 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 \texttt{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.

The algorithm described in [10] 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.