DBFT: Efficient Byzantine Consensus with a Weak Coordinator and its Application to Consortium Blockchains

Tyler Crain[1], Vincent Gramoli[1,2], Mikel Larrea[1,3], and Michel Raynal[4]

1 **University of Sydney, Sydney, Australia**
{tyler.crain,vincent.gramoli}@sydney.edu.au
2 **Data61-CSIRO, Australia**
3 **University of the Basque Country UPV/EHU, Spain**
mikel.larrea@ehu.eus
4 **IUF and Université de Rennes, Rennes, France**
raynal@inria.fr

Abstract

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 a classic reduction from multivalue to binary consensus to illustrate how it is used in the Red Belly Blockchain.

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 Mostéfaoui, Moumem 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 our geo-distributed experiment, even when attacked by a well-engineered coalition of Byzantine processes.

Keywords and phrases Byzantine consensus, weak coordinator, geo-distribution

Digital Object Identifier 10.4230/LIPIcs...

1 Introduction and Related Work

To circumvent the impossibility of solving consensus in asynchronous message-passing systems [26] where processes can be faulty or *Byzantine* [42], researchers typically use randomization 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 [36]. A recent randomized algorithm without signature [49] solves consensus in $O(1)$ expected

time under a fair scheduler. The fair scheduler assumption was later relaxed in an extended version [50] 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 [23]. Interestingly, these solutions typically require a unique *coordinator* process, sometimes called a leader, to be non-faulty [22, 23, 18, 46, 37, 8, 2, 43]. 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. We 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 coordinator. 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 message from the coordinator. In our algorithm this allows the binary consensus to terminate in as few as 2 message delays even in the case of initial disagreement between non-faulty processes and a faulty coordinator.

To mitigate the problem of a slow or Byzantine coordinator, other approaches 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 [21, 3]. While this still requires a strong coordinator in each round, it favors the fastest coordinator in successive rounds. An exponential information gathering tree was used to terminate in $t+3$ rounds without a coordinator [9]. Another approach was to have only the coordinator of each round to decide [23], requiring at least $O(t)$ rounds.

Application to consortium blockchains. To motivate our algorithm, we study its applicability to the recent context of *blockchains* [54]. Blockchains originally aimed at tracking ownership of digital assets where any Internet users could solve a cryptopuzzle before proposing, for consensus, a block of asset transactions. The *consortium blockchains* [13] 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 [39, 62] where a sequence of commands must be decided by multiple processes. A slight difference 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 natural 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 generalize common definitions of Byzantine consensus, that either assume that no value proposed only by Byzantine processes can be decided [20, 50, 53], or that any value (i.e., possibly proposed by a Byzantine process) can be decided [23, 35, 45, 46, 61]. While the validity allows a

decided value to combine multiple proposals, it is less strict than interactive consistency [59] or vector consensus [57]: for example, it does not require the decided value to combine at least $t+1$ values proposed by correct processes. The HoneyBadger blockchain [48] was built upon the Coin randomized consensus to tolerate unexpected delays [49] whereas a more recent blockchain, called the Red Belly Blockchain[1], exploited the weak coordinator of our algorithm to reach high performance [30].

Geo-distributed experimentation with Byzantine coalition. To validate our expectations experimentally, we deployed our consensus algorithm on 100 Amazon VMs located 5 datacenters in different continents. We also implemented "Coin" the recent randomized algorithm from Moustéoui et al. [50] 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 increases the number of rounds needed to decide.

We also implement 4 different Byzantine attacks: Byz1 where Byzantine processes send a bit b where the protocol specification expect 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 exceed slightly the second only under the Byz3 attacks.

Finally, we combine our consensus algorithm with the reduction of multivalue to binary consensus by Ben-Or et al. [7] to propose a novel Democratic Byzantine Fault Tolerant (DBFT) consensus algorithm applicable to consortium blockchains. The Red Belly Blockchain [30] already builds upon DBFT and presents high performance.

Roadmap. Section 2 presents the model. Section 3 presents the binary Byzantine consensus algorithm. Section 4 presents the consensus definition and an application to the blockchain context. Section 5 presents the experiments and Section 6 concludes the paper. Appendix A presents additional experiments. Appendix B recalls the reliable broadcast abstraction. Appendix C presents proofs of saftey and liveness of the algorithms. Appendix D presents an implementation of the BV-broadcast abstraction, which is used in our algorithms.

2 A Byzantine 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. 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

[1] `http://redbellyblockchain.io`.

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.

Additional synchrony assumption. 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 [26]. 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 τ, there is an upper bound δ on message transfer delays. This eventual synchrony assumption is denoted $\diamond Synch$.

Notations. 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. The basic computation model strengthened with the additional constraint $t < n/3$ is denoted $\mathcal{BAMP}_{n,t}[t < n/3]$. The latter computation model strengthened with the eventual synchrony constraint $\diamond Synch$ is denoted $\mathcal{BAMP}_{n,t}[t < n/3, \diamond Synch]$.

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. Our algorithm terminates after $t + 1$ asynchronous rounds, which is optimal [25], but terminates in 2 message delays in the best case.

The algorithm is built incrementally. We first recall the binary consensus problem and the existing BV-broadcast abstraction on which our algorithm relies and present a safe binary consensus algorithm in the $\mathcal{BAMP}_{n,t}[t < n/3]$ model before presenting a safe and live consensus algorithm in the $\mathcal{BAMP}_{n,t}[t < n/3, \diamond Synch]$ model. The aim of this incremental approach is to facilitate the understanding.

3.1 The Binary Consensus Problem

Let $\mathcal{V}$ be the set of values that can be proposed. While $\mathcal{V}$ can contain any number (≥ 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.

3.2 The Binary Value Broadcast Communication Abstraction

Our binary consensus algorithm relies on a binary value all-to-all communication abstraction originally introduced for randomized consensus [50] and denoted BV-broadcast. A simple implementation is deferred to Appendix D. BV-broadcast 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)" with $m \in \{0, 1\}$.

In a BV-broadcast instance, each non-faulty process p_i BV-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.

3.3 A Safe Consensus Algorithm

Figure 1 describes a simple binary Byzantine consensus algorithm based on [50], which satisfies the BBC-Validity and BBC-Agreement properties in the model $\mathcal{BAMP}_{n,t}[t < n/3]$. This algorithm proceeds in rounds and relies on the BV-broadcast abstraction presented before.

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. Figure 1 describes a simple binary Byzantine consensus algorithm, which satisfies the BBC-Validity and BBC-Agreement properties in the system model $\mathcal{BAMP}_{n,t}[t < n/3]$. It provides the processes with the operation bin_propose(). The processes decide value v when invoking decide(v) at line 10.

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 = ∪_{1≤x≤n−t} b_val_x
                     and (ii) values_i ⊆ 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] ∪ {v};
```

Figure 1 A safe algorithm for the 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), 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: 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.
  - 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-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.

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 [50]. Instead we preserve the simplicity of this algorithm and postpone a deterministic terminating solution in Section 3.4.

3.4 A Safe and Live Consensus Algorithm in $\mathcal{BAMP}_{n,t}[t < n/3, \diamond Synch]$

To circumvent the consensus impossibility [26], we assume now eventual synchrony and present an algorithm solving the binary Byzantine consensus problem in the $\mathcal{BAMP}_{n,t}[t < n/3, \diamond Synch]$ model. While always terminating in $O(t)$ message delays, which is optimal [25], it actually terminates in $O(1)$ message delays if all non-faulty processes propose the same value and even without eventual synchrony.

The 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 *weak round coordinator*: the weak coordinator of round r is the process p_i such that $i = ((r - 1) \bmod n) + 1$. Note that this new round coordinator is only used to help agreement by suggesting a value and thus differs from the classic coordinator [19, 23] or the eventual leader that cannot be implemented in $\mathcal{BAMP}_{n,t}[t < n/3]$.

3.4.1 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 weak round coordinator.
- 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 COORD_VALUE$[r]()$ to broadcast the value it tries to favor to become the decided value.

3.4.2 Description of the extended algorithm.

The following items explain the new and modified statements that appear in Figure 2.

```
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;
(Opt1)   if (est_i = −1) then est_i ← 1; // "fast-path" for round 1, only used in the reduction in Sect. 4
(04)       else BV_broadcast EST[r_i](est_i);
         end if;
(New1)   wait_until (bin_values_i[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} = bin_values_i[r_i]; // 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)   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_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 = ∪_{1≤x≤n−t} b_val_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
             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;
(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;
(13)   end while.
```

Figure 2 A safe and live algorithm for the binary Byzantine consensus in $\mathcal{BAMP}_{n,t}[t < n/3, \diamond Synch]$. Line (Opt1) is only applied in the multivalued reduction presented in Section 4. Line (Opt2) is a mechanism to prevent unnecessary rounds from being executed

- At line New1, p_i waits until a value enters *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 not used here, and is only used along with the reduction to multivalued consensus presented in Section 4.
- Line New4 waits until $(n - t)$ AUX$[r]()$ messages are received from different processes before reseting 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 round coordinator (whose value is computed on line New2) to try to impose the first value that enters into its *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 $values_i$, 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 modification that minimizes the amount of extra rounds processes need to execute after deciding. The first condition, (**wait until** ($bin\_values_i[r_i] = \{0, 1\}$)), 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 *bin_values*, the process will not move on to the next round until this is true. The second condition, (**if** (decided in round $r_i - 2$)), halts the process 2 rounds after it has decided, as all non-faulty processes must have decided by this round.

As just seen, the idea made operational by these new or modified statements is the following: benefit from a non-faulty weak 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 weak 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-05.
- Then, assuming the previous item, all non-faulty processes set aux_i to $\{w\}$ (line New3), 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 $(r + 2)$.

4 From Multivalued to Binary Consensus in a Byzantine System

This section presents a reduction from multivalued to the binary consensus algorithm and how to adapt it for consortium blockchains. We consider a generalization 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 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 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 [15] 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 [38] 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 [16] required that the decided value be one of the proposed values.

```
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)  decide(proposals_i[j]).

(11) 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 BIN_CONS[j] end if. // BV-deliver 1 in rnd1 bin_propose

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

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

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 multivalued Byzantine consensus to binary Byzantine consensus is described in Figure 3 and is similar to an existing reduction [7], except that it combines the reliable broadcast RB-broadcast [11] 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. The behavior of a process p_i can be decomposed into 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_j, 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: p_i starts participating in a first set of binary consensus instances (lines 02-04).

Process p_i 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 04). (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. This allows the consensus algorithm to terminate in 4 message delays in the best case, i.e. 3 message delays to execute the reliable broadcast and 1 to complete the binary consensus by skipping the BV_broadcast step.

5 Experiments on 100 VMs on Distinct Continents

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

5.1 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 the consensus

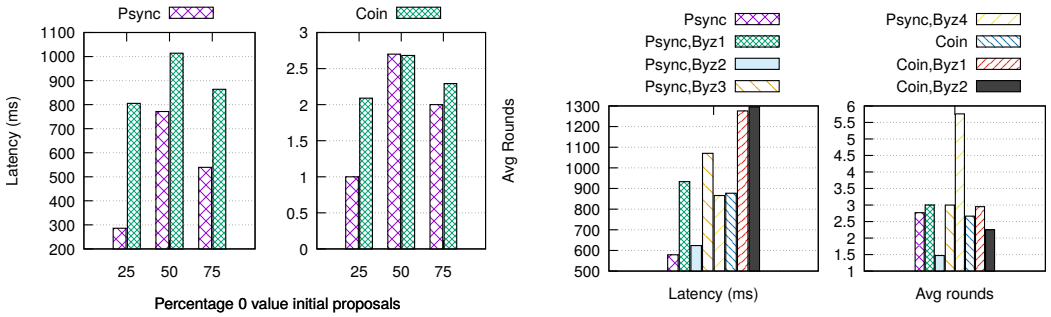

Figure 4 Geo-distributed experiments of latency and average number of rounds for our deterministic binary Byzantine consensus "Psync" and the randomized binary Byzantine consensus "Coin" with 4 different Byzantine attacks (Byz1, ..., Byz4): **(left)** with varying levels of initial disagreement, **(right)** with random initial proposals.

algorithm. The Red Belly Blockchain builds upon the same combination of DBFT and TLS by storing the necessary certificates in its blocks.

For the sake of comparison, we also implemented the randomized binary Byzantine consensus algorithm from Mostéfaoui et al. [49], called "Coin", as a baseline. Coin terminates in O(1) rounds in expectation and is at the heart of the HoneyBadger permissioned blockchain [48] but requires a fair scheduler [50]. All 100 machines are c4.xlarge machines of Amazon EC2 equipped with an Intel Xeon E5-2666 v3 processor 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 secure point-to-point channels with TLS and 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.

5.2 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 United States (Oregon, Northern California, and Ohio) and 2 in Europe (Ireland and Frankfurt). 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.

5.3 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 $\neg(r \bmod 2)$ to all non-faulty nodes. Finally, Byzantine nodes instantly send AUX message with value $\neg(r \bmod 2)$ to a single node and send AUX message with value $(r \bmod 2)$ to the remaining 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.

6 Conclusion

To conclude, our weak coordinator based Byzantine consensus is time optimal, resilience optimal, does rely on randomization or signatures and improves over the randomized Byzantine consensus algorithms [49, 50] 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.

We have demonstrated empirically that DBFT can be used on 100 machines distributed over distinct continents, which makes it a suitable solution for large-scale permissioned blockchain systems. 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 [1] without proof-of-work.

References

1 Abraham, I., Malkhi, D., Nayak, K., Ren, L., Spiegelman, A. Solida: A Blockchain Protocol Based on Reconfigurable Byzantine Consensus. *Proc. 21st International Conference on Principles of Distributed Systems*, pp. 1–19, (2017)

2 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)

3 Aublin P.-L., Ben Mokhtar, S., Quema V., RBFT: Redundant Byzantine Fault Tolerance. *Proc. 33rd Int'l Conference on Distributed Computing Systems* pp. 297–306, (2013)

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

5 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)

6 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)

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

8 Bessani, A., Sousa, J., Alchieri, E.A.P., State Machine Replication for the Masses with BFT-SMART. *Proc. 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*pp. 355-362 (2014)

9 Brief Announcement: A Leader-free Byzantine Consensus Algorithm. Fatemeh Borran and André Schiper. DISC 2009.

10 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)

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

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

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

14 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)

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

16 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

17 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)

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

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

20 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)

21 Clement, A., Wong, E., Alvisi, L., Dahlin, M. and Marchetti, M. Making Byzantine fault tolerant systems tolerate Byzantine faults. NSDI (2009).

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

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

24 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)

25 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)

26 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)

27 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)

28 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)

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

30 Vincent Gramoli. The Red Belly Blockchain. Invited talk. MIT, Cambridge, USA.

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

32 Hearn M., Corda: a distributed ledger. Version 0.5 (2016)

33 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)

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

35 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)

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

37 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)

38 Kursawe K., Optimistic asynchronous Byzantine agreement. Manuscript (2000)

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 Micali, S. ALGORAND: The Efficient and Democratic Ledger. arXiv:1607.01341v7 (2016).

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 Byzantine Consensus with $T < N/3$ and $O(N^2)$ Messages. *Proc. of the 2014 ACM Symposium on Principles of Distributed Computing*, p.2–9, (2014)

50 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)

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

55 Natoli C. and Gramoli V., The Balance Attack or Why Forkable Blockchains Are Ill-Suited for Consortium. *Proc. 47th IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'17)* (2017)

56 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)

57 Neves N. F. and Correia M. and Verissimo P., Solving vector consensus with a wormhole, IEEE Transactions on Parallel and Distributed Systems, 16(12):1120-1131 (2005)

58 Pass, R. and Shi, E. The Sleepy Model of Consensus. Cryptology ePrint Archive, 2016/918 (2018)

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

A Additional experiments

A.1 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) $\neg(r \bmod 2)$ to enter *bin_values* of non-faulty nodes and (b) no node must receive $n-t$ AUX messages with value $(r \bmod 2)$. Then to ensure (ii) we need (c) both 0 and 1 to enter *bin_values* of non-faulty nodes, (d) at least one node must receive receive $n-t$ AUX messages with value $\neg(r \bmod 2)$, and (e) at least one node must receive receive an AUX messages with value $(r \bmod 2)$.

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 $\neg(r \bmod 2)$ to all non-faulty nodes, this message is delivered instantly, as a result all non-faulty processes broadcast an AUX message with value $\neg(r \bmod 2)$. Then to ensure (d), Byzantine nodes instantly send AUX message with value $\neg(r \bmod 2)$ to a single node. Furthermore, to ensure (e), Byzantine nodes instantly send AUX message with value $(r \bmod 2)$ to the remaining nodes. Assuming both 0 and 1 entered *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 *bin_values* of non-faulty nodes. If $\neg(r \bmod 2)$ enters too late, a process may broadcast $(r \bmod 2)$ as its AUX message, and as a result we may fail with (d). Otherwise if $(r \bmod 2)$ enters *bin_values* too late, all non-faulty processes may terminate with $n-t$ AUX messages with value $\neg(r \bmod 2)$. Similar timing arguments can be made for other non-terminating scenarios that use different message patterns.

A.2 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 crypotgraphic 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.

B Reliable broadcast in Byzantine systems

This broadcast abstraction (in short, RB-broadcast) was proposed by G. Bracha [11]. 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

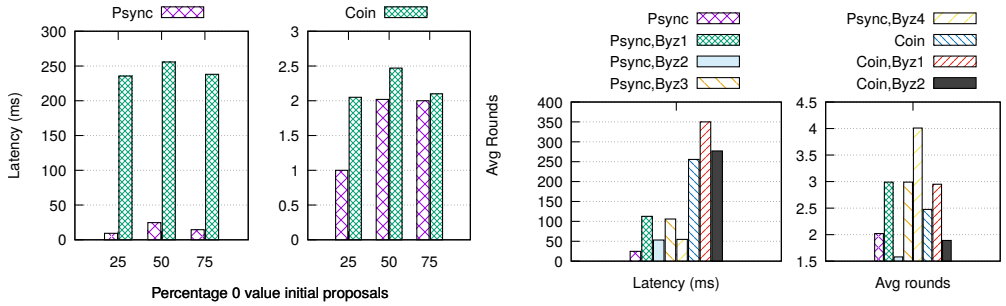

Figure 5 Single datacenter comparison of latency and average number of rounds to terminate of our deterministic binary Byzantine consensus against randomized binary Byzantine consensus: **(Left)**

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 [12] 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.

The algorithm described in [11] 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 [34].

C Proofs of safety and liveness of the algorithms

C.1 Safety proof of the binary Byzantine consensus (Figure 1)

The proof is describe from a point of view of a non-faulty process p_i. Let $values_i^r$ denote the value of the set $values_i$ 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.

▶ **Lemma 1.** *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 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. □$_{Lemma\ 1}$

▶ **Lemma 2.** *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 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 $v = w$. □$_{Lemma\ 2}$

▶ **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 $bin\_values_i[1]$ contains only values proposed by non-faulty processes. Consequently, the non-faulty processes broadcast at line 06 messages 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. □$_{Lemma\ 3}$

▶ **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 $v = (r \bmod 2)$.

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 $v = (r \bmod 2)$ 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 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 $(r \bmod 2) = v$ 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 1, 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 4*

▶ **Lemma 5.** *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. □*Lemma 5*

▶ **Lemma 6.** *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 $(r + 1)$. □*Lemma 6*

▶ **Lemma 7.** *Let the system model be $\mathcal{BAMP}_{n,t}[t < n/3]$. If every non-faulty process p_i terminates a round r with $values_i^r = \{0, 1\}$, then 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 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 $values_i^{r+2} = \{v\}$, at each non-faulty process p_i. It follows that each non-faulty process decides at line 10. □*Lemma 7*

▶ **Theorem 8.** *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 8*

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.[2]

C.2 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 9.** *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 $bin\_values_i[r]$ 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 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 $bin\_values_i[r] = \{v\}$ 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 $aux_i = bin\_values_i[r] = \{v\}$. It then follows that $values_i^r = \{v\}$ and the lemma remains true due to lines 09 and 10.
  - If the 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 COORD_VALUE$[r](\{1 - v\})$. As $(1 - v) \notin bin\_values_i[r]$, at line New3, 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 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.

□$_{Lemma\ 9}$

[2] In the case of the randomized binary consensus algorithm of [50], the common coin guarantees termination with probability 1, because eventually the singleton value in $values_i$ will match the coin.

▶ **Lemma 10.** *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[ ]() 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 τ from which (due to Lemma 13) 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 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. □$_{Lemma\ 10}$

▶ **Theorem 11.** *The algorithm described in Figure 2 solves the binary Byzantine consensus in the system model $\mathcal{BAMP}_{n,t}[t < n/3, \diamond Synch]$.*

Proof The proof follows directly from Lemma 9 (BBC-Validity and BBC-Agreement) and Lemma 10 (BBC-Termination). □$_{Theorem\ 11}$

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 10). 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 to the slow non-faulty processes to catch up to the most advanced non-faulty processes (as measured by their mini-round number).[3] To this end, when a process is in a mini-round ρ and receives messages corresponding to another mini-round ρ' 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 ρ, .., $(\rho' - 1)$. 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 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 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.

- 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.
- 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 (i.e. the slowest for that mini-round) 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 (an 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 New1 or New4 before starting its timer for that mini-round.
- Mini-round ρ_δ is the first mini-round where $timeout > \delta$ at any non-faulty process.

The proof is made up of two lemmas. Lemma 12 shows that processes will eventually reach a point where they remain no more than one mini-round apart. Lemma 13 builds upon this to show that the rounds eventually become synchronous.

▶ **Lemma 12.** *Consider the algorithm of Figure 2 enriched with the previous catch-up mechanism. There is a mini-round ρ_t such that in ρ_t 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 ρ_t until after they start waiting for the timeout of mini-round ρ_t.*

[3] Similar mechanisms are used by PBFT [18].

Proof Let us consider mini-round ρ_t 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 ρ_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:

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

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 $(\gamma_{fast_{\rho'}} + \theta_{fast} + timeout_{\rho'})$ in a mini-round ρ'. Given that timeouts start with value 0 in mini-round 0 and grows by 1 in each mini-round, *timeout* can be replaced with ρ for any mini-round ρ as a lower bound for the fastest process. We can then compute the time where the first non-faulty process arrives at mini-round ρ' (where $\rho' > \rho_\delta$) as:

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

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 how long it will take the slowest non-faulty process to execute mini-round ρ' when it does not wait at a timeout. By definition we know the process will spend no more time than θ_{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 $(bin\_values_i[r_i] \neq \perp)$. 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 ρ'. 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 6, 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 $bin\_values_i[r_i]$ 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 $2 \times \delta + \theta_{slow}$ 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 δ 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 $bin\_values_i[r_i]$ 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 $2 \times \delta + \theta_{slow}$ time to execute the mini-round.

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 (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 $timeout > (3\times\delta+\theta_{slow})$, once inequality (1) is true, it will remain true for all following mini-rounds. This will be done by induction. Consider $t_{last_{\rho_t}} < t_{first_{\rho_{t+1}}}$ is satisfied, let us now show that $t_{last_{\rho_t+1}} < t_{first_{\rho_{t+2}}}$ must also be satisfied. For this to not hold, the slowest non-faulty process must spend more time on mini-round ρ_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, $p_{last_{\rho_t}}$ must receive $(t + 1)$ messages from mini-round $(\rho_t + 1)$ after δ 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 $2 \times \delta + \theta_{slow}$ time to complete the mini-round. Thus, as long as $timeout > (3 \times \delta + \theta_{slow})$, which will eventually be true given $\diamond Synch$ and the growing timeout, process $p_{last_{\rho_t}}$ will reach mini-round $(\rho_t + 1)$ before $p_{first_{\rho_t+1}}$ reaches mini-round $(\rho_t + 2)$. $\square_{Lemma\ 12}$

▶ **Lemma 13.** *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 12 it is known that there exists a mini-round ρ_t 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 $timeout > (3\times\delta+\theta_{slow})$. 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 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$[r]()$). Thus, for a mini-round ρ'_t to by synchronous where $\rho'_t \geq \rho_t$, the following needs to be ensured:

$$t_{last_{\rho'_t}} + (3 \times \delta) + \theta_{slow} \leq t_{first_{\rho'_t}} + \gamma_{fast_{\rho'_t}} + timeout_{\rho'_t}. \tag{2}$$

Let us now compute $t_{last_{\rho'_t}}$. 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 $(t_{first_{\rho'_t}} + \gamma_{fast_{\rho'_t}} + \theta_{fast})$ at least one process has satisfied the condition on line New1 or New4 (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 12 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 12 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 ρ'_t is given by:

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

Now plugging this into inequality (2) leads to $timeout_{\rho'_t} \geq (7 \times \delta) + (2 \times \theta_{slow}) + \theta_{fast}$ (note that $2 \times \theta_{slow}$ is included to account for possible processing times in both mini-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 12 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 12 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. □$_{Lemma\ 13}$

C.3 Proof of the Blockchain Consensus (Figure 3)

▶ **Lemma 14.** *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 $proposals_i[j] \neq \perp$ 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.

□$_{Lemma\ 14}$

▶ **Lemma 15.** *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 only enter the bin_values of a BBC instance after validation at line 11, and (ii) the Intrusion-tolerance property of $BIN\_CONS[j]$, that at least one non-faulty process p_i inserted 1 into its $binvalues$ on line 12. Due to line 11, it follows that $proposals_i[j]$ contains a valid value.

□$_{Lemma\ 15}$

▶ **Lemma 16.** *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 $k1 = \min\{x \text{ such that } bin\_decisions_i[x] = 1\}$ and $k2 = \min\{x \text{ such that } bin\_decisions_j[x] = 1\}$.

On the one hand, it follows from line 07 that $(\bigwedge_{1 \leq x \leq n} bin\_decisions_i[x] \neq \bot)$ and $(\bigwedge_{1 \leq x \leq n} bin\_decisions_j[x] \neq \bot)$, 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 $k1 = k2 = \min\{x \text{ such that } dec[x] = 1\} = k$. 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 inserted 1 into its *binvalues* on line 12. As this invocation can be issued only at line 03, we conclude (from the predicate of line 02) that $proposals_\ell[k] = v \neq \bot$. 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. □*Lemma* 16

▶ **Lemma 17.** *Every non-faulty process decides a value.*

Proof It follows from Lemma 14 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 16, the predicate of line 09 is eventually satisfied at each non-faulty process, which concludes the proof of the lemma. □*Lemma* 17

▶ **Theorem 18.** *The algorithm described in Figure* 3 *implements multivalued Byzantine consensus (*VPBC*) in the system model* $\mathcal{BAMP}_{n,t}[t < n/3, \text{BBC}]$.

Proof Follows from Lemma 15 (VPBC-Validity), Lemma 16 (VPBC-Agreement), and Lemma 17 (VPBC-Termination). □*Theorem* 18

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

Figure 6 depicts the pseudocode of an existing implementation [50] of the BV-broadcast problem stated in Section 3.2.

```
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 BV-deliver B-VAL(v)    // local delivery of a value
(07)  end if.
```

Figure 6 An algorithm implementing BV-broadcast in $\mathcal{BAMP}_{n,t}[t < n/3]$ (from [50])