DBFT: Efficient Leaderless Byzantine Consensus and its Application to Blockchains

Tyler Crain and Vincent Gramoli
University of Sydney
Australia
{*tyler.crain,vincent.gramoli*}@*sydney.edu.au*

Mikel Larrea
Univ. of the Basque Country UPV/EHU
Spain
mikel.larrea@ehu.eus

Michel Raynal
Université de Rennes, France
Polytechnic University, Hong Kong
raynal@inria.fr

***Abstract*—This paper introduces a new leaderless Byzantine consensus called the *Democratic Byzantine Fault Tolerance (DBFT)* for blockchains. While most blockchain consensus protocols rely on a correct leader or coordinator to terminate, our algorithm can terminate even when its coordinator is faulty.**

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 decentralization is particularly appealing for blockchains for two reasons: *(i)* each node plays a similar role in the execution of the consensus, hence making the decision inherently "democratic"; *(ii)* decentralization avoids bottlenecks by balancing the load, making the solution scalable.

DBFT is deterministic, assumes partial synchrony, is resilience optimal, time optimal and does not need signatures. 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 whose fast path terminates in 4 message delays.

***Index Terms*—Byzantine consensus, weak coordinator, geo-distribution**

I. Introduction and Related Work

To circumvent the impossibility of solving consensus in asynchronous message-passing systems [22] where processes can be faulty or *Byzantine* [30], researchers typically use randomization [3], [6], [14] 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 [26]. A recent randomized algorithm without signature [34] solves consensus in $O(1)$ expected time under a fair scheduler. The fair scheduler assumption was later relaxed in an extended version [35] 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 [21]. Interestingly, these solutions typically require a unique *coordinator*, or leader, to be non-faulty [4], [8], [15], [20], [21], [27], [31], [32]. 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 well-known drawback of this approach is that a faulty coordinator can dramatically impact the algorithm performance [1], [5], [17] by leveraging the power it has in a round and imposing its value to all.

In this paper, we present *Democratic Byzantine Fault Tolerance (DBFT)*, a Byzantine consensus algorithm that copes with this problem by not relying on a classic coordinator or leader. Instead, DBFT uses what we refer to as a *weak coordinator* that 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 values that might all be decided. Furthermore, having a weak coordinator allows rounds to be executed optimistically without waiting for a specific message. Finally, DBFT is time optimal, resilience optimal and does not need signatures.

To mitigate the limitations of leader-based Byzantine consensus, 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 [5], [17]. While this still requires a classic 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]. Other solutions [21], [43] require at least $O(t)$ rounds. By contrast our weak coordinator only helps agreement by suggesting a value while still allowing a fast path termination in a constant number of message delays, hence differing from the classic coordinator [16], [21] or the eventual leader approaches that cannot be implemented in $\mathcal{BAMP}_{n,t}[t < n/3]$.

Application to blockchains. To motivate our algorithm, we study its applicability to the recent context of *blockchains* [37]. 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. New blockchain models became promising at reducing the amount of resources consumed by avoiding to resolve the cryptopuzzle but restricting the set of proposers to a subset of known processes. The *consortium blockchains*[1] rely on a preselected set of n known block proposers. The *community blockchains* [44] introduces dynamism by allowing different set of n_i participants to propose blocks for different index i of the chain. This is why, various blockchains started exploring the use of Byzantine fault tolerant consensus, like the ordering service of Hyperledger Fabric [2] that relies on BFTSmart [8] or Tendermint [11] that relies on a variant of PBFT [15]. Unfortunately, these approaches are leader-based and cannot scale beyond few nodes [8], [11]. Honeybadger [33] uses a randomized algorithm [34] which requires a fair scheduler. As far as we know, only the Red Belly Blockchain [24] uses a similar deterministic leaderless solution.

These blockchains seem similar to replicated state machines [29], [42] where a sequence of commands must be decided by multiple processes. 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 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 extend common definitions of Byzantine consensus, that either assume that no value proposed only by Byzantine processes can be decided [18], [35], [36], or that any value (i.e., possibly proposed by a Byzantine process) can be decided [21], [25], [32], [38], [41]. Interestingly, our definition is less strict than interactive consistency [40] or vector consensus [39]: for example, it does not require the decided value to combine at least $t + 1$ values proposed by correct processes.

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 blockchains that terminates in 4 messages delays in the good case, when all non-faulty processes propose the same value.

Roadmap. Section II presents the model. Section III presents the binary Byzantine consensus algorithm. Section IV presents the consensus definition and an application to the blockchain context and Section V concludes the paper. The proofs of safety and termination as well as experimental results are deferred to the companion technical report [19].

II. A Byzantine Computation Model

A. Asynchronous processes

The system consists 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.

[1]https://blog.ethereum.org/2015/08/07/on-public-and-private-blockchains/.

B. Communication network

The processes communicate by exchanging messages through an asynchronous reliable point-to-point network. "Asynchronous" means that there is no bound on message transfer delays, but these delays are finite. "Reliable" means that the network does not lose, duplicate, modify, or create messages. "Point-to-point" means that any pair of processes is connected by a bidirectional channel. Hence, when a process receives a message, it can identify its sender. A process p_i sends a message to a process p_j by invoking the primitive "send $\text{TAG}(m)$ 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 $\text{TAG}(m)$ is used as a shortcut for "**for each** $p_i \in \Pi$ **do** send $\text{TAG}(m)$ to p_j **end for**".

C. Failure model

Up to t processes can exhibit a *Byzantine* behavior [40]. 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.

D. 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 [22]. 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 and process computation delays. This eventual (or partial) synchrony assumption is denoted $\Diamond Synch$.

E. 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 $\mathcal{BAMP}_{n,t}[t < n/3, \diamond Synch]$.

III. 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 $\mathcal{BAMP}_{n,t}[t < n/3]$ model and finally presenting a safe and live consensus algorithm in the $\mathcal{BAMP}_{n,t}[t < n/3, \diamond Synch]$ 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 (≥ 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.

A. 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 [35], and restated in the companion technical report [19].

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.

B. 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.

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

C. 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 $\mathcal{BAMP}_{n,t}[t < n/3]$ but not BBC-termination. The algorithm is depicted in Figure 1 and provides the process p_i with the operation $\mathsf{bin\_propose}(v_i)$ to propose its initial value v_i. Process p_i proceeds in asynchronous rounds and decides value v when invoking $\mathsf{decide}(v)$ 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. 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 $\mathsf{BV\_broadcast}()$ 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

```
operation bin_propose($v_i$) is
(01) $est_i \leftarrow v_i$; $r_i \leftarrow 0$;
(02) while (true) do
(03)     $r_i \leftarrow r_i + 1$;
(04)     BV_broadcast EST$[r_i](est_i)$; // add to $bin\_values[r_i]$ upon $BV\_delivery$
(05)     wait_until $(bin\_values_i[r_i] \neq \emptyset)$;
(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 $p(x)$, $1 \leq x \leq n-t$, and their contents are
                 such that $\exists$ a non-empty set $values_i$ where (i) $values_i = \cup_{1 \leq x \leq n-t} b\_val_{p(x)}$
                 and (ii) $values_i \subseteq bin\_values_i[r_i]$);
(08)     $b_i \leftarrow r_i$ mod 2;
(09)     if $(values_i = \{v\})$ // $values_i$ is a singleton whose element is $v$
(10)         then $est_i \leftarrow v$; if $(v = b_i)$ then decide($v$) if not yet done end if;
(11)         else $est_i \leftarrow b_i$
(12)     end if;
(13) end while.

(14) when B-VAL$[r](v)$ is BV-delivered by BV_broadcast$[r]$ do
         $bin\_values_i[r] \leftarrow bin\_values_i[r] \cup \{v\}$;
```

Fig. 1. A safe algorithm for the binary Byzantine consensus in $\mathcal{BAMP}_{n,t}[t < n/3]$

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 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 for the same round) 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. 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 III-D. The proof of correctness of algorithm 1 is deferred to the companion technical report [19].

D. Psync: Safe and Live Consensus in $\mathcal{BAMP}_{n,t}[t < n/3, \diamond Synch]$

We now present *Psync*, an algorithm solving the binary Byzantine consensus problem in the $\mathcal{BAMP}_{n,t}[t < n/3, \diamond Synch]$ model. Similar to the safe algorithm (Section III-C), 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 *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 [16], [21].

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.

```
operation bin_propose($v_i$) is
(01)    $est_i \leftarrow v_i$; $r_i \leftarrow 0$;
        $timeout_i \leftarrow 0$;
(02)    while (true) do
(03)        $r_i \leftarrow r_i + 1$;
(Opt1)      if $(est_i = -1)$ then $est_i \leftarrow 1$; // "fast-path" for round 1, only used in the reduction in Sect. IV
(04)            else BV_broadcast EST$[r_i](est_i)$;
            end if;
(New1)      wait_until $\big(bin\_values_i[r_i] \neq \emptyset\big)$;
            $timeout_i \leftarrow timeout_i + 1$; set $timer_i$ to $timeout_i$;
(New2)      $coord_i \leftarrow ((r_i - 1) \mod n) + 1$;
            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 $\big((bin\_values_i[r_i] \neq \emptyset) \wedge (timer_i$ expired$)\big)$;
(New3)      if $\big(($COORD_VALUE$[r_i](w)$ received from $p_{coord_i}) \wedge (w \in bin\_values_i[r_i])\big)$
                then $aux_i \leftarrow \{w\}$
                else $aux_i \leftarrow bin\_values_i[r_i]$
            end if;
(M-06)      broadcast AUX$[r_i](aux_i)$;
(New4)      wait_until $\big($a message AUX$[r_i]()$ has been received from $(n-t)$ different processes$\big)$;
            set $timer_i$ to $timeout_i$;
(M-07)      wait_until $\big($(messages AUX$[r_i](b\_val_{p(1)})$, ..., AUX$[r_i](b\_val_{p(n-t)})$ have been received
                       from $(n-t)$ different processes $p(x)$, $1 \leq x \leq n-t$, and their contents are
                       such that $\exists$ a non-empty set $values_i$ where (i) $values_i = \cup_{1 \leq x \leq n-t} b\_val_{p(x)}$
                       and (ii) $values_i \subseteq bin\_values_i[r_i]) \wedge (timer_i$ expired$)\big)$;
(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) $\wedge$ (one of them is $aux_i$)
                       then $values_i \leftarrow aux_i$ end if; // $values_i$ is either defined here or at line M07
(08)        $b_i \leftarrow r_i \mod 2$;
(09)        if $(values_i = \{v\})$ // $values_i$ is a singleton whose element is $v$
(10)            then $est_i \leftarrow v$; if $(v = b_i)$ then decide($v$) if not yet done end if;
(11)            else $est_i \leftarrow b_i$
(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.
```

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

The weak coordinator of round r, uses the message type 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 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 IV.
- 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 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 optimization to minimize 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 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 [16], [21]. To this end:

- The weak 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, 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 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 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 $\diamond Synch$, this ensures that all non-faulty processes eventually execute synchronous rounds. The proof of liveness of Algorithm 2 is deferred to the companion technical report [19].

IV. 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 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 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 blockchains, and possibly other specific Byzantine consensus problems. In the context of blockchains, a proposal is not valid if either it does not contain an appropriate hash of the last block added to the Blockchain or it contains invalid transactions. There exist similar problem definitions whose validity also relies on the notion of a predicate. The validated Byzantine consensus [12] 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 [28] 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 [13] required that the decided value be one of the proposed values, something we do not require. Finally, vector consensus [39] and interactive consistency [30] both require a minimal number of proposals to be decided.

```
operation mv_propose($v_i$) is
(01) RB_broadcast VAL($v_i$);
(02) repeat if $(\exists\ k : (proposals_i[k] \neq \bot) \wedge$
              $(BIN\_CONS[k]$.bin_propose() not invoked$))$
(03)              then invoke $BIN\_CONS[k]$.bin_propose($-1$) end if;
(04) until $(\exists \ell :\ bin\_decisions_i[\ell] = 1)$ end repeat;
(05) for each $k$ s.t. $BIN\_CONS[k]$.bin_propose() not yet invoked
(06)         do invoke $BIN\_CONS[k]$.bin_propose(0) end for;
(07) wait_until $(\bigwedge_{1\leq x\leq n} bin\_decisions_i[x] \neq \bot)$;
(08) $j \leftarrow$ min$\{x$ such that $bin\_decisions_i[x] = 1\}$;
(09) wait_until $(proposals_i[j] \neq \bot)$;
(10) decide($proposals_i[j]$).

(11) when VAL($v$) is RB-delivered from $p_j$ do
     if valid($v$) then
         $proposals_i[j] \leftarrow v$;
         BV-deliver B-VAL[1](1) to $BIN\_CONS[j]$ end if.

(12) when $BIN\_CONS[k]$.bin_propose() decides a value $b$
     do $bin\_decisions_i[k] \leftarrow b$.
```

Fig. 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 III-D. 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; $\bot$ denotes a default value that cannot be proposed by a (faulty or non-faulty) process.

- An array $proposals_i[1..n]$ initialized to $[\bot, \cdots, \bot]$. 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 companion technical report [19], 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_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:** 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 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.

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

The proof of correctness of DBFT is deferred to the companion technical report [19].

V. 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 [34], [35]. We presented how it can be used for blockchains by generalizing the Byzantine consensus problem and presenting a solution that combines an optimized reduction with our binary Byzantine consensus algorithm.

A variant of DBFT is now at the heart of the Red Belly Blockchain, one of the fastest blockchains to date [24]. Future work involves adapting DBFT to support rational processes rather than simply either Byzantine or non-faulty processes.

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. This research is in part supported under Australian Research Council Discovery Projects funding scheme (project number 180104030) entitled “Taipan: A Blockchain with Democratic Consensus and Validated Contracts” and grants IT980-16 and TIN2016-79897-P.

References

[1] Yair Amir, Brian Coan, Jonathan Kirsch, and John Lane. Prime: Byzantine replication under attack. *IEEE Transactions on Dependable and Secure Computing*, 8(4):564–577, July 2011.

[2] Jo ao Sousa, Alysson Bessani, and Marko Vukolić. A byzantine fault-tolerant ordering service for the Hyperledger Fabric blockchain platform. In *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 51–58, June 2018.

[3] James Aspnes. Randomized protocols for asynchronous consensus. *Distrib. Comput.*, 16(2-3):165–175, September 2003.

[4] Pierre-Louis Aublin, Rachid Guerraoui, Nikola Knežević, Vivien Quéma, and Marko Vukolić. The next 700 BFT protocols. *ACM Trans. Comput. Syst.*, 32(4):12:1–12:45, January 2015.

[5] Pierre-Louis Aublin, Sonia Ben Mokhtar, and Vivien Quéma. RBFT: Redundant byzantine fault tolerance. In *2013 IEEE 33rd International Conference on Distributed Computing Systems*, pages 297–306, July 2013.

[6] Michael Ben-Or and Ran El-Yaniv. Resilient-optimal interactive consistency in constant time. *Distributed Computing*, 16(4):249–262, Dec 2003.

[7] Michael Ben-Or, Boaz Kelmer, and Tal Rabin. Asynchronous secure computations with optimal resilience (extended abstract). In *Proceedings of the Thirteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '94, pages 183–192, New York, NY, USA, 1994. ACM.

[8] Alysson Bessani, Joo Sousa, and Eduardo E.P. Alchieri. State machine replication for the masses with bft-smart. In *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 355–362, June 2014.

[9] Fatemeh Borran and André Schiper. A leader-free byzantine consensus algorithm. In Krishna Kant, Sriram V. Pemmaraju, Krishna M. Sivalingam, and Jie Wu, editors, *Distributed Computing and Networking*, pages 67–78, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[10] Gabriel Bracha. Asynchronous byzantine agreement protocols. *Inf. Comput.*, 75(2):130–143, November 1987.

[11] Ethan Buchman. Tendermint: Byzantine fault tolerance in the age of blockchains, 2016. MS Thesis.

[12] Christian Cachin, Rachid Guerraoui, and Luís Rodrigues. *Introduction to Reliable and Secure Distributed Programming*. Springer Publishing Company, Incorporated, 2nd edition, 2011.

[13] Christian Cachin, Klaus Kursawe, Frank Petzold, and Victor Shoup. Secure and efficient asynchronous broadcast protocols. In *Proceedings of the 21st Annual International Cryptology Conference on Advances in Cryptology*, CRYPTO '01, pages 524–541, London, UK, UK, 2001. Springer-Verlag.

[14] Ran Canetti and Tal Rabin. Fast asynchronous byzantine agreement with optimal resilience. In *Proceedings of the Twenty-fifth Annual ACM Symposium on Theory of Computing*, STOC '93, pages 42–51, New York, NY, USA, 1993. ACM.

[15] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance and proactive recovery. *ACM Trans. Comput. Syst.*, 20(4):398–461, November 2002.

[16] Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *J. ACM*, 43(2):225–267, March 1996.

[17] Allen Clement, Edmund Wong, Lorenzo Alvisi, Mike Dahlin, and Mirco Marchetti. Making byzantine fault tolerant systems tolerate byzantine faults. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'09, pages 153–168, Berkeley, CA, USA, 2009. USENIX Association.

[18] Miguel Correia, Nuno Ferreira Neves, and Paulo Veríssimo. From consensus to atomic broadcast: Time-free byzantine-resistant protocols without signatures. *Comput. J.*, 49(1):82–96, January 2006.

[19] Tyler Crain, Vincent Gramoli, Mikel Larrea, and Michel Raynal. DBFT: Efficient byzantine consensus with a weak coordinator and its application to consortium blockchains. Technical Report 1702.03068, arXiv, 2017.

[20] Danny Dolev, Cynthia Dwork, and Larry Stockmeyer. On the minimal synchronism needed for distributed consensus. *J. ACM*, 34(1):77–97, January 1987.

[21] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *J. ACM*, 35(2):288–323, April 1988.

[22] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM 32, no. 2, 374-382.*, 1985.

[23] Michael J. Fischer and Nancy A. Lynch. A lower bound for the time to assure interactive consistency. *Inf. Process. Lett.*, 14(4):183–186, 1982.

[24] Vincent Gramoli. The Red Belly Blockchain, July 2017. Personal Communication.

[25] Kim Potter Kihlstrom, Louise E. Moser, and P. M. Melliar-Smith. Byzantine fault detectors for solving consensus. *The Computer Journal*, 46(1):16–35, 2003.

[26] Valerie King and Jared Saia. Byzantine agreement in expected polynomial time. *J. ACM*, 63(2):13, 2016.

[27] Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. Zyzzyva: Speculative byzantine fault tolerance. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, pages 45–58, New York, NY, USA, 2007. ACM.

[28] Klaus Kursawe. Optimistic asynchronous byzantine agreement, 2000. manuscript.

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

[30] Leslie Lamport, Robert Shostak, and Marshall Pease. The byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401, July 1982.

[31] Shengyun Liu, Paolo Viotti, Christian Cachin, Vivien Quéma, and Marko Vukolic. XFT: practical fault tolerance beyond crashes. In *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016.*, pages 485–500, 2016.

[32] Jean-Philippe Martin and Lorenzo Alvisi. Fast byzantine consensus. *IEEE Transactions on Dependable and Secure Computing*, 3(3):202–215, July 2006.

[33] Andrew Miller, Yu Xia, Kyle Croman, Elaine Shi, and Dawn Song. The honey badger of BFT protocols. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 31–42, New York, NY, USA, 2016. ACM.

[34] Achour Mostéfaoui, Hamouma Moumen, and Michel Raynal. Signature-free asynchronous byzantine consensus with t ¿ n/3 and o(n2) messages. In *Proceedings of the 2014 ACM Symposium on Principles of Distributed Computing*, PODC '14, pages 2–9, New York, NY, USA, 2014. ACM.

[35] Achour Mostéfaoui, Hamouma Moumen, and Michel Raynal. Signature-free asynchronous binary byzantine consensus with t¿ n/3, o(n2) messages, and o(1) expected time. *J. ACM*, 62(4):31:1–31:21, September 2015.

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

[37] Satoshi Nakamoto. Bitcoin: a peer-to-peer electronic cash system, 2008. http://www.bitcoin.org.

[38] Lynch Nancy. *Distributed algorithms*. Morgan Kaufmann, San Francisco (CA), 1996.

[39] Nuno F. Neves, Miguel Correia, and Paulo Verissimo. Solving vector consensus with a wormhole. *IEEE Trans. Parallel Distrib. Syst.*, 16(12):1120–1131, December 2005.

[40] Marshall Pease, Robert Shostak, and Leslie Lamport. Reaching agreement in the presence of faults. *J. ACM*, 27(2):228–234, April 1980.

[41] Michel Raynal. *Communication and Agreement Abstractions for Fault-tolerant Asynchronous Distributed Systems*. Morgan and Claypool Publishers, 1st edition, 2010.

[42] Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Comput. Surv.*, 22(4):299–319, December 1990.

[43] T. K. Srikanth and Sam Toueg. Simulating authenticated broadcasts to derive simple fault-tolerant algorithms. *Distributed Computing*, 2(2):80–94, Jun 1987.

[44] Guillaume Vizier and Vincent Gramoli. ComChain: Bridging the gap between public and consortium blockchains. In *Proceedings of the IEEE International Conference on Blockchain (Blockchain'18)*, pages 1469–1474, Jul 2018.