Chapter 5

Making Blockchains Scale

I understand democracy as something that gives the weak the same chance as the strong.

Mohandas Gandhi

5.1 Introduction

In this chapter, we tackle the problem of *scalability* or the problem of preserving or improving a property as the system size increases. As blockchain found applications to track ownership of digital assets, it is crucial for companies to adopt more secure blockchains than the ones proven vulnerable to network attacks before moving them in production. In this chapter, we present *Red Belly Blockchain*, the first blockchain whose throughput scales to hundreds of geodistributed consensus participants without requiring synchrony or randomization. To achieve this result, Red Belly Blockchain drastically revisits BFT blockchains, getting rid of the predominant leader-based design to decide multiple proposals and parallelizing the verification of signatures.

Red Belly Blockchain builds upon the Democratic BFT consensus algorithm, that aims at balancing the load of proposing new blocks on multiple nodes rather than concentrating it on a single leader node. To this end, every node exploits a reliable broadcast primitive that propagates their block to other nodes, before invoking a series of binary consensus whose decisions indicate which reliably delivered blocks can be combined into a superblock.

5.2 Consensus without synchrony

In Chapter 3, we have seen several algorithms that solve consensus while assuming synchrony, or that every message in the network takes less time to be delivered than a known upper-bound. This synchrony assumption helps us solve the consensus problem since we know now that under asynchrony and failures, one would not be able to solve consensus. This synchrony assumption also makes the solution vulnerable to network attacks as we illustrated with the Balance Attack in Chapter 4. Partial synchrony (cf. Section 2.4.4) is another assumption not as strong as synchrony but that also offers the possibility to solve consensus. It allows to tolerate unpredictable network delays while always ensuring that no disagreement—and thus no forks—can occur as long as there are strictly less than $n/3$ failures. Recall that partial synchrony requires all messages to be delivered in a bounded but unknown amount of time.

5.2.1 The seminal Practical Byzantine Fault Tolerance

PBFT is a consensus protocol that works when assuming partial synchrony and when $n > 3f$. More precisely, PBFT is considered a state machine replication protocol because it can run a sequence of consecutive Byzantine consensus instances for nodes to agree upon the same sequence of commands to execute. Since its initial publication in 1999, it has been influential in the design and implementation of a large class of Byzantine consensus protocols (and their associated state machine replication protocol) for secure distributed systems outside the scope of blockchains. PBFT relies on a node that plays a special

role, called the *leader*, in order to solve the classic Byzantine consensus problem of Definition 3. In particular, PBFT falls in the class of *leader-based consensus algorithms*, the consensus algorithms in which a unique leader node sends its proposal to all other nodes in order to collect votes on this particular proposal. A *leaderless consensus algorithm* is one that is not leader-based.

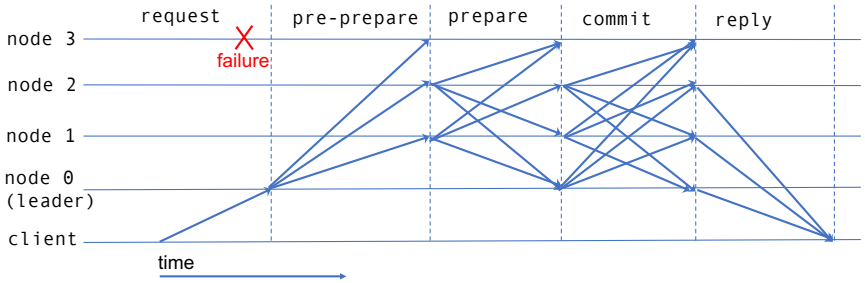

Figure 5.1: The leader-based communication pattern of a good execution of PBFT involves a leader that relays the client request to the $n-1$ followers so that a decision is taken by a correct node if $2f+1$ distinct replicas committed the request.

A simple, yet ideal, execution of PBFT is represented in Figure 5.1 as a distributed execution where time increases from left to right, and where a client sends a request to a leader that requires the participation of $2f+1$ servers, including itself, to decide upon this request. In this execution, one server, called node 3, is Byzantine and fails by crashing while the other nodes are correct. Note that the executions of the PBFT algorithm can be more elaborate: for example, the algorithm includes a view-change mechanism to change from one leader to another in case the leader is suspected to be faulty. Unfortunately, there is no way to detect whether a node is faulty, and a wrong suspicion can thus lead to a performance overhead induced by such a view-change.

5.2.2 Complexities

If the leader of PBFT is suspected by the other participants to be faulty, then a view-change procedure (not presented in Figure 5.1) starts to replace the leader in a round-robin fashion. To make sure that a new view will not prepare a different value from the one committed in a previous view, some information should be propagated across consecutive views. As the view-change rotates among at most $f+1$ nodes to elect a correct leader, there are at most $f+1$ view-change rounds before termination, hence leading to a time complexity of $O(f)$.

The message in each round is broadcast by n nodes leading to a message complexity of $O(fn^2)$.

The message in each view-change round contains the state received from the previous view-change rounds, which is at most $O(f)$ bits. As there are up to $O(fn^2)$ of these messages, the bit complexity is $O(f^2n^2)$ bits.

5.2.3 Changes required by the scale of the consensus network

PBFT was designed long before blockchain was invented, and was demonstrated on a network file system application running on four machines of a Local Area Network (LAN). This small scale deployment contrasts with the blockchain context that is typically a wide area network. The large variety of off-the-shelf PBFT-like protocols led blockchain developers to simply build their blockchain upon variants of PBFT, all inheriting its leader-based pattern that was shown instrumental for LANs. The question we answer below is whether this centralized design decision prevents these secure blockchains from scaling to wider networks.

5.3 Leveraging bandwidth

The inherent centralization of leader-based consensus algorithms makes these algorithms heavily dependent on the resources of the unique particular node that plays the role of the leader. The leader can act as a bottleneck when the distributed system enlarges or as the number of consensus participants increases. The view-change procedure that replaces a suspicious leader is typically difficult to implement and thus error-prone. The leader also makes the protocols vulnerable to denial-of-service attacks, where flooding the current leader is sufficient to stop the service. This is why coping with the problem induced by a leader has been an active area of research for decades. We illustrate below a reason why the leader can act as a bottleneck by comparing the time complexity of propagating a proposal in a leader-based design (as in the pre-prepare phase of PBFT depicted in Fig. 5.1) and a leaderless design that will be presented in detail in Section 5.5.

5.3.1 The time complexity of a leader-based propagation

Let us introduce a simple example to demonstrate the problem of the leader bottleneck and illustrate the advantage of a more distributed alternative. Consider a blockchain service that relies on PBFT for participating nodes to decide upon the next block to be appended to the blockchain. In a leader-based consensus protocol, the leader proposes a block to all the other nodes, as indicated by the one-to-all message exchange of Figure 5.1, and tries to gather sufficiently many votes in order to decide this block. We can already anticipate that the sending of this block to all nodes will have a cost imposed by the resources of the leader: for the consensus to be reached fast, the leader should send this value to all other nodes as fast as possible. Although this broadcast can be implemented efficiently in a LAN through the data link layer, it cannot be done

in a Wide Area Network (WAN) where there is no broadcast support. Instead, the leader will have to send a separate copy of this block to each individual node, a task whose execution time increases with the number of nodes.

More formally, consider a block of B bits and a blockchain system with n nodes with limited bandwidth resources. In particular, each node i is equipped with a download capacity of d_i and an upload capacity of u_i, both expressed in bits per unit of time. Without loss of generality, let the leader be process 1 with capacities d_1 and u_1. The time P it will take the leader to send the block to all nodes (we consider that the leader sends to itself for simplicity) is the maximum between these two:

- The time for the leader to upload nB bits, which is $\frac{nB}{u_1}$.
- The time to download B bits for the node that has the lowest download rate among all other nodes, which is $\frac{B}{min(d_i:1\leq i\leq n)}$.

Hence the time it takes for the leader to propagate the block to all nodes is $P = max\left(\frac{nB}{u_1}, \frac{B}{min(d_i:1\leq i\leq n)}\right)$ and we can conclude that this time is $\Omega(n)$.

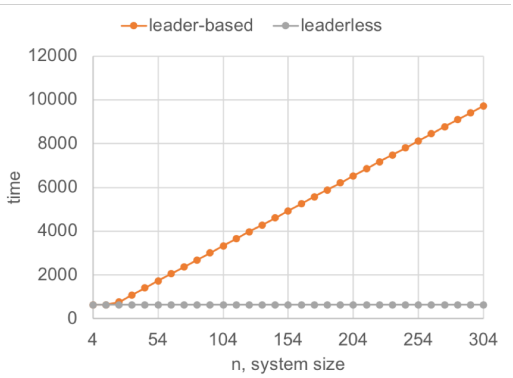

Figure 5.2: The time it takes for propagating a block in a leader-based consensus algorithm typically increases with the number of nodes whereas the time to propagate a block in a leaderless consensus algorithm is independent of n.

5.3.2 The time complexity of a leaderless propagation

To contrast the leader-based approach with a more 'decentralized' baseline, consider an hypothetical leaderless consensus protocol where all nodes propose values that can be decided. The example of such an algorithm is deferred to Section 5.5. Hence, let every node send B/n bits to each other for each of them to receive the same block of B bits as in the previous example. The time P' it takes for a leaderless protocol to exchange this block is the maximum of these two values:

- The time for the node with the lowest upload rate to upload B/n bits, which is $\frac{B/n}{min(u_i:1\leq i\leq n)}$.
- The time for the node with the lowest download rate to download B bits, which is $\frac{B}{min(d_i:1\leq i\leq n)}$.

Hence, the time it takes for the leaderless protocol to propagate the block to all nodes is $P' = max\left(\frac{B/n}{min(u_i:1\leq i\leq n)}, \frac{B}{min(d_i:1\leq i\leq n)}\right)$, which does not increase linearly with n. In other words, the performance of a leader-based algorithm degrades with the size of the network, but it is not necessarily the case for leaderless algorithms as explained below.

5.3.3 Bypassing the leader bottleneck with the superblock optimization

To illustrate the degradation of performance of the leader-based algorithm, Figure 5.2 depicts the time it would take to propagate a block of B bits in a system where the leader has a 10 times higher upload rate than any other node and where every node downloads twice faster than it can upload. When the system is small, the time it takes for a leader-based propagation of B bits is similar to the time it takes for a leaderless propagation of B bits, however, when the system size exceeds 21, then the leaderless protocol starts being faster and the difference in performance increases as the system grows. This illustrates an inherent lack of scalability in traditional leader-based designs induced by a bottleneck effect: the limited resources of the leader.

To conclude, a fundamental limitation of the leader-based consensus algorithm stems from the need for one node to propose its value of B bits to the rest of the nodes: at the end of the consensus algorithm execution, only this value can be decided. Of course, some tentatives may fail and other leaders might be elected one after another, but in the end the value decided is the value proposed by the chosen leader. By contrast, the leaderless consensus algorithm benefits from having multiple nodes proposing B/n bits and the possibility for the decided value to combine all these n proposed values into one large decided value of size B. In the context of blockchain, the values can be called blocks, hence we say that combining n proposed blocks, each of size B/n, results into a *superblock* of size B. This optimization of bypassing the leader bottleneck is thus called the *superblock optimization*.

5.4 The Set Byzantine Consensus problem

It is interesting to note that the classic definition of the consensus problem does not allow the superblock optimization. In particular, solving consensus generally requires to satisfy a validity property that states that the value decided must be one of the values that were proposed. As an example, recall the validity property of the previous consensus definition (Def. 3): this validity is not

satisfied by the superblock optimization as the decided value can be none of the proposed value; the decided value results from the combination of multiple proposed values. This is the reason why we need a new definition for the consensus problem, called the Set Byzantine Consensus (Def. 4) problem, that relaxes the classic validity requirement to allow for the superblock optimization.

Definition 4 (Set Byzantine Consensus) *Assuming that each correct node proposes a proposal, the* Set Byzantine Consensus *(SBC) problem is for each of them to decide on a set in such a way that the following properties are satisfied:*

- *SBC-Termination: every correct node eventually decides a set of transactions;*
- *SBC-Agreement: no two correct nodes decide on different sets of transactions;*
- *SBC-Validity: a decided set of transactions is a non-conflicting set of valid transactions taken from the union of the proposed sets; and if all nodes are correct and propose the same set of non-conflicting valid transactions, then this subset is the decided set.*

The SBC-Termination and SBC-Agreement properties are common to many Byzantine consensus definition variants, while SBC-Validity is different: it includes two predicates, the first states that transactions proposed by Byzantine proposers could be decided as long as they are correctly signed and non conflicting; the second one is necessary to prevent any trivial algorithm that decides a pre-determined value from solving the problem.

So why is the Set Byzantine Consensus not the classic definition of the consensus problem? There are several reasons. First, consensus is an old problem defined in the 80's at which time the key challenge was not to make a software run on a distributed system of thousands of machines. Second, the classic definition gave a uniform description of three properties whose simplicity had a pedagogical advantage. Third, blockchain did not exist at that time and researchers were trying to find a practical Byzantine fault tolerant consensus implementation that could run within a network file system on four nodes. Once discovered this protocol influenced a long series of Byzantine fault tolerant systems that all inherit the same leader-based design as listed in Section 5.8. Below, we explain how to solve the Set Byzantine Consensus with a leader-less consensus algorithm for large-scale blockchain systems.

5.5 Democratic Byzantine consensus

The Democratic Byzantine fault tolerance (DBFT) consists of avoiding the bottleneck effect of the leader by simply avoiding any particular process trying to impose its proposal. DBFT relies on a reduction from the binary Byzantine consensus 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 message delays in the good case, when all correct processes propose the same value. Note that the original DBFT algorithm [CGLR18] solves the classic Byzantine consensus we mentioned in Chapter 3. As we will see in Section 5.6, in order for DBFT to solve the Set Byzantine Consensus (Definition 4) we propose to merge the acceptable proposals into the final decision.

5.5.1 The binary Byzantine consensus problem

We first give the definition of the Binary Byzantine Consensus (BBC) problem.

Definition 5 (Binary Byzantine Consensus) *Assuming that each correct process proposes a binary value in* $\{0, 1\}$*, the Binary Byzantine consensus problem is for each of them to decide on a value in such a way that the following properties are satisfied:*

1. *Termination: every correct node eventually decides.*
2. *Agreement: no two correct nodes decide differently.*
3. *BBC-Validity. If all correct processes propose the same value, no other value can be decided.*

5.5.2 The binary Byzantine consensus algorithm of DBFT

For the sake of simplicity in the presentation, we present a safe version of the binary Byzantine consensus in Algorithm 8, a safe and live version is deferred to Section 5.6.

As depicted in Algorithm 8, process p_i proposes its initial binary value v_i by invoking bin_propose(v_i) at line 1. It then sets its estimate to its value (line 2) and initializes the round number r (line 3) before it can decide a value v by invoking decide(v) at line 14 within an asynchronous round (lines 4–15). Each of these asynchronous rounds comprises three phases depicted below:

Phase 1: Discard estimates proposed exclusively by Byzantine processes. Process p_i increments the round number at line 5. It then invokes an existing reliable broadcast protocol applied to binary values, called bv-broadcast (lines 16–22), that discards all the values proposed exclusively by Byzantine processes [MMR15b]. Note that this call is non-blocking, hence it executes in the background once the next line is reached. Symbol $\rightarrow$ indicates that the array $bin\text{-}values_i$ gets populated in the background by all concurrent bv-broadcasts (line 6) at index r for round r.

During the bv-broadcast (lines 16–22), every correct process counts the number of distinct processes from which it receives each binary value and re-broadcasts the value(s) received from at least $f + 1$ distinct processes knowing that they are necessarily coming from at least one correct process (line 19). Finally, each of them "delivers" each value received by $2f + 1$ distinct processes by adding it to the set $bin\text{-}values_i$ so that for all correct processes p_j,

Algorithm 8 The safe binary Byzantine consensus algorithm of DBFT

```
1: bin-propose(val):
2:   est ← val
3:   r ← 0
4:   repeat:
5:     r ← r + 1
6:     bv-broadcast(EST[r], val, i) → bin-values[r]
7:     wait until (bin-values[r] ≠ ∅)
8:     broadcast(AUX[r], bin-values[r], i) → favorites[r]
9:     wait until ∃values ⊆ favorites[r] where the following conditions hold:
10:      • if v received in AUX msgs of n − t processes then v ∈ values
11:      • ∀v ∈ values, v ∈ bin-values[r]
12:    if (values = {v}) then
13:      est ← v
14:      if v = (r mod 2) and no previous decision by p_i then decide(v)
15:    else est ← (r mod 2)

16: bv-broadcast(MSG, val, i):
17:   broadcast(BV, ⟨val, i⟩)
18:   repeat:
19:     if (BV, ⟨v, ∗⟩) received from (f + 1) processes but not yet re-broadcast then
20:       broadcast(BV, ⟨v, i⟩)
21:     if (BV, ⟨v, ∗⟩) received from (2f + 1) processes then
22:       bin-values ← bin-values ∪ {v}
```

$bin\text{-}values_j$ eventually contains all values broadcast by correct processes but no values broadcast only by Byzantine processes. Note that we do re-broadcast at line 20 the value received even if it was broadcast by the same process at line 17.

Phase 2: Identify values as sufficiently represented proposals. This second phase runs between lines 8 and 11. In this phase, p_i broadcasts normally (i.e., without using a reliable or binary value broadcast primitive) a message $\text{AUX}[r]$ whose content is $bin\text{-}values_i[r]$ (line 8) and delivers the broadcast messages it received from other processes by adding them to $favorites_i$.

Then, p_i waits until it has received a set of values $values_i$ satisfying the two following properties:

1. The values in $values_i$ come from the messages $\text{AUX}[r]$ of at least $(n - f)$ different processes.

2. The values in $values_i$ are included in the set $bin\text{-}values_i[r]$. Thanks to the bv-broadcast that filters out Byzantine value, even if Byzantine processes send fake $\text{AUX}[r]$ messages containing values proposed only by Byzantine processes, $values_i$ will contain only values broadcast by some correct process.

Hence, at any round r, after line 11, $values_i$ is included in $\{0, 1\}$ and contains only the initial binary values of correct processes.

Phase 3: Deciding the estimate upon convergence. The third phase runs between lines 12 and 15. This phase is a local computation phase, during which (if not done yet) p_i tries at line 14 to decide a value v that depends on the content of $values_i$ and the parity of the round.

1. If $values_i$ contains a single element v (line 12), then v becomes p_i's new estimate at line 13 and a candidate for the consensus decision. To ensure BBC-Agreement, v can be decided only if $v = r \bmod 2$. The decision is taken by the statement $\mathsf{decide}(v)$ at line 14.

2. If $values_i = \{0,1\}$, then p_i cannot decide. As both values have been proposed by correct processes, p_i selects the parity of the round $r \bmod 2$, which is the same at all correct processes, as its new estimate (line 15).

Let us observe that the invocation of $\mathsf{decide}(v)$ 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. For the terminating version of this algorithm, refer to Algorithm 12.

5.5.3 Safety proof of the binary Byzantine consensus

The proof is described from a point of view of a correct process p_i. Let $values_i^r$ denote the value of the set $values_i$ which satisfies both the predicate of lines 12 and 13 when invoked at line 11 during a round r. We now restate the properties of BV-broadcast [MMR15b] using the notations presented at lines 16–22 of Algorithm 8.

Definition 6 (BV-broadcast) *The* bv-broadcast *ensures the following properties:*

- *BV-Obligation: If at least $(f+1)$ correct processes bv-broadcast the same value v, then v is eventually delivered to the set $bin\text{-}values_i$ of each correct process p_i;*
- *BV-Justification: If a correct process p_i delivers v, then v has been bv-broadcast by a correct process;*
- *BV-Uniformity: If a correct process p_i delivers v, then v is eventually delivered at all correct processes;*
- *BV–Termination: Eventually all correct processes deliver some value.*

Lemma 7 *If at the beginning of a round r, all correct processes have the same estimate v, they never change their estimate value thereafter.*

Proof Let us assume that all correct processes (which are at least $n - f > f + 1$) have the same estimate v when they start round r. Hence, they all BV-broadcast the same message $\text{EST}[r](v)$ at line 6. It follows from the BV-Justification and BV-Obligation properties that each correct process p_i is such that $bin\text{-}values_i[r] = \{v\}$ at line 7, and consequently can broadcast only

AUX$[r](\{v\})$ at line 8. Considering any correct process p_i, it then follows from the predicate of line 11 ($values_i$ contains only v), the predicate of line 12 ($values_i$ is a singleton), and the assignment of line 14, that est_i keeps the value v. □

The next lemma states that if the set *values* in the same round of two correct processes are singletons then they are identical.

Lemma 8 *Let p_i and p_j be two correct processes.* $\big((values[r]_i = \{v\}) \wedge (values[r]_j = \{w\})\big) \Rightarrow (v = w)$.

Proof Let p_i be a correct process such that $values[r]_i = \{v\}$. It follows from line 11 that p_i received the same message AUX $[r](\{v\})$ from $(n - f)$ different processes, i.e., from at least $(n - 2f)$ different correct processes. As $n - 2f \geq f + 1$, this means that p_i received the message AUX $[r](\{v\})$ from a set Q_i including at least $(f + 1)$ different correct processes.

Let p_j be a correct process such that $values[r]_j = \{w\}$. Hence, p_j received AUX $[r](\{w\})$ from a set Q_j of at least $(n - f)$ different processes. As $(n - f) + (f + 1) > n$, 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 correct process. Hence, at line 8, p_k sent the same message AUX $[r](\{\})$ to p_i and p_j, and we consequently have $v = w$. □

Theorem 9 (BBC-Validity) *Let $f < n/3$. The value decided by a correct process was proposed by a correct process.*

Proof Let us consider the round $r = 1$. Due to the BV-Justification property of the BV-broadcast of line 6, it follows that the sets $bin\text{-}values_i[1]$ contains only values proposed by correct processes. Consequently, the correct processes broadcast at line 8 messages AUX$[1]()$ containing sets with values proposed only by correct processes. It then follows from the predicate of line 11 ($values[1]_i \subseteq bin\text{-}values_i[1]$), and the BV-Justification property of the BV-broadcast abstraction, that the set $values[1]_i$ of each correct process contains only values proposed by correct processes. Hence, the assignment of est_i (be it at line 13 or 15) provides it with a value proposed by a correct process. The same reasoning applies to rounds $r = 2$, $r = 3$, etc., which concludes the proof of the lemma. □

Theorem 10 (Agreement) *Let $f < n/3$. No two correct processes decide different values.*

Proof Let r be the first round during which a correct process decides, let p_i be a correct process that decides in round r (line 14), and let v be the value it decides. Hence, we have $values[r]_i = \{v\}$ where $v = (r \bmod 2)$.

If another correct process p_j decides during round r, we have $values[r]_j = \{w\}$, and, due to Lemma 13, we have $w = v$. Hence, all correct processes that decide in round r, decide v. Moreover, each correct process that decides in round r has previously assigned $v = (r \bmod 2)$ to its local estimate est_i.

Let p_j be a correct that does not decide in round r. As $values[r]_i = \{v\}$, and p_j does not decide in round r, it follows from Lemma 13 that we cannot have $values[r]_j = \{1 - v\}$, and consequently $values[r]_j = \{0, 1\}$. Hence, in round r, p_j executes line 13, where it assigns the value $(r \bmod 2) = v$ to its local estimate est_j.

It follows that all correct processes start round $(r + 1)$ with the same local estimate $v = r \bmod 2$. Due to Lemma 12, they keep this estimate value forever. Hence, no different value can be decided in a future round by a correct process that has not decided during round r, which concludes the proof of the lemma. □

5.6 Red Belly Blockchain

Recently, a new blockchain appeared particularly promising, the Red Belly Blockchain.[1] It relies on DBFT (Section 5.5) that solves the Set Byzantine Consensus problem (Definition 5.4) and scales to large networks by exploiting resources appropriately.

This section presents the two main design features of RBBC: its verification leverages the few computational resources when the system is small and its consensus leverages communication to commit more transactions when the system is large and bandwidth becomes limited. Although we call them 'sharding' techniques, all deciders decide the same set of transactions as opposed to traditional sharded blockchains [LNZ+16, ZMR18].

5.6.1 Reducing the computation at small scale

Verification is needed to guarantee that the *Unspent Transaction Output (UTXO)* transactions [Nak08] are correctly signed. As verifications are CPU-intensive and notably affect performance, we shard the verification by letting different verifiers verify distinct transactions without reducing security. The expected result is twofold. First, it improves performance as it reduces each verifier computational load. Second, it helps scaling by further reducing the per-verifier computational load as the number of verifiers increases. We confirmed empirically [CNG18] that this can reduce the verification load to a third of what it would be without verification sharding.

As f verifier nodes can be Byzantine, each transaction signature has to be checked by at least $f + 1$ verifier nodes. If all the $f + 1$ nodes are unanimous in that the signature check passes, then at least one correct node checked the signature successfully and it follows that the transaction is guaranteed to be correct. Given that f nodes may be Byzantine, a transaction may need to be verified by up to $2f + 1$ times before $f + 1$ equal responses are computed. This is why we map each transaction to $f + 1$ *primary verifiers* and f *secondary verifiers*, hence summing up to $2f + 1$ verifiers.

[1] http://poseidon.it.usyd.edu.au/~concurrentsystems/rbbc/.

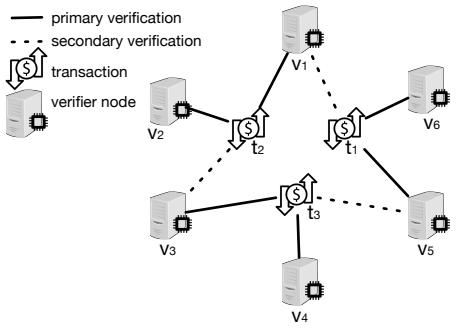

Figure 5.3: The sharded verification with $n = 6$ and $f = 1$ where each transaction is verified by $f + 1$ (2 in this example) primary verifier nodes linked to this transaction with a solid line, before being verified only if necessary by f (1 in this example) secondary verifier nodes linked to this transaction with a dashed line

The sharded verification consists of assigning proposals to two groups of nodes, $f + 1$ primary verifiers and f secondary verifiers as depicted in Figure 5.3. At the beginning of the consensus, every proposer node broadcasts its set of transactions to all permissioned nodes. Once a node receives one of the proposals for which it is a primary verifier, it immediately verifies the transactions of these proposals and broadcasts the outcome of the verification to all. As soon as a secondary verifier of a transaction detects that its $f + 1$ primary verifiers did not sent the same outcome, it verifies this proposal itself and broadcasts the outcome.

5.6.2 Leveraging bandwidth at larger scales

In a large scale environment, geodistributed proposers are likely to receive different sets of transactions coming from requesters located in their vicinity. Instead of selecting one of these sets as the next block and discarding the others, RBBC combines all the sets of transactions proposed by distinct servers into a unique superblock to improve the performance (as we quantify empirically in [CNG18]).

In particular, RBBC decides upon multiple proposed sets of transactions. To illustrate why this is key for scalability, consider that each of the n proposers proposes $O(1)$ transactions. As opposed to blockchains based on traditional Byzantine consensus that will decide $O(1)$ transactions, RBBC can decide $\Omega(n)$ transactions. As the typical communication complexity of Byzantine consensus is $O(n^4)$ bits [CL02, CGLR18], it results that $O(n^3)$ bits are needed per committed transaction in RBBC, instead of $O(n^4)$.

To illustrate how RBBC achieves this optimization, consider Figure 5.4 that depicts $n = 4$ permissioned nodes that propose different sets of transactions

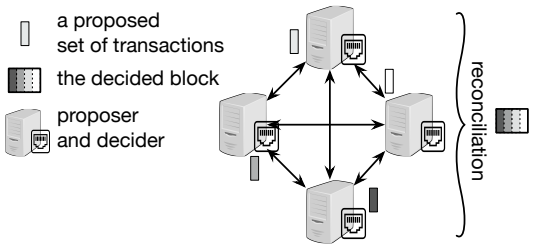

Figure 5.4: Blockchain consensus executed by $n = 4$ proposers among which $t = 1$ is Byzantine: they propose sets of transactions (depicted with gray rectangles) and the decided value (on the right-hand side) is actually a block containing the union of all verified and non-conflicting transactions from the sets that were proposed (including potentially the set proposed by the Byzantine node)

but that decide a value that is actually a superblock containing the union of all sets of transactions that were proposed. It results from this optimization that the number of transactions decided grows linearly in n as long as each proposer proposes disjoint sets of valid transactions.

When the consensus is about to output the sets of transactions to be decided, some of these transactions may not be executable. For example, if these transactions withdraw a cumulative amount from an address that exceeds its remaining balance. Thus before deciding on the superblock, a deterministic reconciliation is performed at every node in lexicographical order of the transactions in the decided sets as detailed in Section 5.6.4.

5.6.3 Assigning roles to nodes

We now explain how node roles are assigned for each transaction using a deterministic function. For each consensus instance, we have an ordered set P of permissioned node identifiers where $f + 1 \leq |P| < n$, indicating the nodes that play the role of primary or secondary proposers for all transactions.

For a requester to identify the proposer nodes responsible to propose a given transaction tx to the consensus, each node can execute a deterministic function $\mu(a)$ that takes as input the source account a of transaction tx and returns the identifier of a node $p_i \in P$, called the *primary proposer* of transaction tx. To guarantee that a transaction is proposed despite a faulty primary proposer, between f and $n - 1$ secondary proposers distinct from p_i are also selected deterministically. The number of proposers of each transaction tx is at least $f + 1$ to guarantee that tx will be proposed by at least one correct node. The number of proposers can be as large as n, however, fewer proposers lower latency whereas more proposers increase throughput, as we experimented in [CNG18].

As each transaction must be verified between $f + 1$ and $2f + 1$ times, each

proposer p_i is also mapped to a set of $f+1$ $primary\_verifiers_{p_i}$ and a set of f additional $secondary\_verifiers_{p_i}$. The $primary\_verifiers_{p_i}$ include p_i itself and verify upon reception the signatures of tx. If the verification returns the same $f+1$ results, then it becomes clear whether the signature of tx is correct. If not, f additional verifications are needed to identify the majority of $f+1$ identical responses indicating whether the signature of tx is correct. This is why, the $secondary\_verifiers_{p_i}$ set includes nodes of P that are distinct from the $primary\_verifiers_{p_i}$ and that verify tx in case one or more of the primary verifiers are faulty or slow. More verifiers can be selected but would waste CPU resources.

5.6.4 From DBFT to Red Belly Blockchain

We now present how to build upon DBFT (Section 5.5) to build the Red Belly Blockchain. For each proposal delivered at proposer p_i by the verified reliable broadcasts (Section 5.6.4), p_i participates in a binary consensus instance with value 1 (line 6). Proposer p_i proposes 0 to the remaining binary consensus instances (line 8) after a timer expires. This timer (line 3) increases with the age of the oldest transaction of the mempool to potentially decide it.

Algorithm 9 The DBFT Byzantine consensus algorithm with the superblock optimization

```
1:  propose(val):
2:    (verified-)reliable-broadcast(val) → props
3:    start-timer(age of oldest tx in mempool)
4:    while ∄k : bitmask[k] = 1 or timer did not expire do
5:      for all k such that props[k] has been delivered
6:        bitmask[k] ← bin-propose_k(1)
7:    for all k such that props[k] has not been delivered
8:      bitmask[k] ← bin-propose_k(0)
9:    wait until bitmask is full and ∀ℓ, bitmask[ℓ] = 1 : props[ℓ] ≠ ∅
10:   reconciliate(bitmask & props)
```

All binary consensus instances proceed in parallel (their invocation is non-blocking). The decisions of these binary consensus instances constitute a *bitmask* that is applied to the set of potentially decidable proposals (line 10). Although the array of verified proposals may differ across correct nodes, the bitmasks of all correct nodes are guaranteed to contain 1s and be identical due to the agreement properties of the binary consensus. Note that even though the proposal may not be known yet for some of these indices, it is guaranteed by the reliable broadcast to be eventually delivered at all correct proposers (Section 5.6.4). Each correct proposer waits until a decidable proposal has been delivered at each of these indices (line 9), then each correct proposer obtains the same set of proposals after applying the *bitmask*.

Modifications

We modify Ben-Or, Kelmer and Rabin's reduction (lines 1–10) of the multi-value consensus problem to the binary consensus problem [BOKR94] to solve the Set Byzantine Consensus (Section 5.4) by replacing the reliable broadcast by our verified reliable broadcast (Section 5.6.4) and invoking a reconciliation to decide a superblock of non-conflicting transactions.

Verified all-to-all reliable broadcast

In order for proposers to exchange verified proposals we add a `secp256k1` *Elliptic Curve Digital Signature Algorithm (ECDSA)* verification step to the reliable broadcast, which is originally a 3-step one-to-all communication abstraction where any message delivered to a correct node gets eventually delivered to all correct nodes [Bra87]. Because this verification only adds local computation and piggybacked information to the consensus, it does not impact its correctness [CGLR18].

Let us first recall that the reliable broadcast protocol [Bra87] consists of the following steps. First a proposer broadcasts an init message containing a proposal. Upon reception, the proposal is then broadcast to all nodes in an echo message. (Note that the init message of the proposer is considered as its echo message.) Upon reception of $\lceil \frac{n+f+1}{2} \rceil$ equal echo messages, a ready message containing the proposal is broadcast. Upon reception of $2f + 1$ equal ready messages the proposal is delivered. If the node receives $f + 1$ equal ready messages before it has sent a ready message, it broadcasts that ready message without waiting.

Our verified variant of the reliable broadcast adds a verification before the broadcast of the READY message. Namely, upon reception of $\lceil \frac{n+f+1}{2} \rceil$ equal ECHO messages at a verifier node, the verification of the proposal starts. Upon completion, a list of integers indicating the indices of invalid transactions in the proposal is appended to the READY message, which is then broadcast. Nodes that are not in the verifier sets will only broadcast a READY message upon the reception of $f + 1$ equal READY messages.

Reconciliation

Thanks to the verified reliable broadcast, each correct node ends up with an array of verified proposals and a bitmask, which is the array of n binary values corresponding to the binary consensus decisions. Although the array of verified proposals may differ across correct nodes, the bitmasks of all correct nodes are guaranteed to contains 1s and be identical due to the agreement properties of the binary consensus [CGLR18]. Correct nodes apply this bitmask to the array of proposals to obtain the indices of the proposals array at which the verified proposals are potentially decidable. Note that even though the proposal may not be known yet for some of these indices, it is guaranteed to be eventually known, thanks to the eventual delivery of the verifiable reliable broadcast

Algorithm 10 Verified reliable broadcast

```
1: verified-reliable-broadcast(v):
2:    broadcast(INIT, v)
3: upon receiving a message (INIT, v) from p_j:
4:    broadcast(ECHO, h(v), j)
5: upon receiving n − f (ECHO, h(v), j) msgs and not having sent READY:
6:    if p_i ∈ primary_verifiers(v) then verif ← verify(v)
7:    if p_i ∈ secondary_verifiers(v) then wait(Δ); verif ← verify(v)
8:    broadcast(READY, verif, h(v), j)
9: upon receiving f + 1 (READY, verif, h(v), j) and not having sent READY:
10:   stop-verify(v)
11:   broadcast(READY, verif, h(v), j)
12: upon receiving n − f (READY, verif, h(v), j) and not delivered from j:
13:   if is-verified(v, verif) then deliver(v, j)
```

that is identical at all correct nodes (cf. Section 5.6.4).

Each correct node waits until a potentially decidable proposal has been delivered at each of these indices, at which point they are guaranteed to have the exact same set of proposals. Then they extract the transactions that do not conflict from all these proposals. This extraction is done deterministically at all correct nodes by going through all proposals in increasing index number and through each of their transactions one-by-one, adding a transaction to the superblock if the UTXO it consumes exists, or simply discarding the transaction, if it conflicts (i.e., consumes the same UTXO) of an already selected transaction. For the sake of fairness (i.e., to not favor any particular proposer), correct nodes traverse the proposals from the index number $(k \bmod n)$ to index number $(k-1 \bmod n)$ where k is the index of the latest superblock in the blockchain. This prevents the proposer with the lowest index number from having its proposed transactions added to the superblock with a higher priority than the transactions of other proposers.

Algorithm 11 Reconciliation

```
1: reconciliate(props):
2:    for i = 0..(n − 1) do
3:       for tx ∈ props[(k + i) mod n] do
4:          for ctx ∈ superblock do
5:             if ¬conflict(tx, ctx) then superblock ← superblock ∪ {tx}
6:    decide(superblock)
```

5.6.5 Binary Byzantine consensus of RBBC

To solve the binary consensus deterministically, we chose the binary consensus of DBFT [CGLR18] because it is resilience optimal and time optimal. Each replica refines an estimate value, initially its input value to the consensus, across consecutive rounds until it decides (line 14). It invokes broadcast primitives that deliver some values into a dedicated variable pointed out by $\rightarrow$ at

lines 6, 7 and 10. The bv-broadcast (line 6) is a reliable broadcast for binary values [MMR15b]. (We optimize by piggybacking it for $r = 1$ with the verified-reliable-broadcast at line 2.) One replica per round acts as a coordinator by broadcasting its value c (line 7) that others prioritize (line 9) to help them converge to the same decision. Hence, RBBC is leaderless with multiple coordinators. The binary values are then forwarded in AUX messages (line 10) and each replica waits to receive a sufficiently represented set of these AUX values (lines 9–11). If only one value is sufficiently represented (line 12) and if it corresponds to the parity of the round, then it is decided (line 14). Otherwise, *val* is set to the parity of the round and another round starts.

Algorithm 12 The binary Byzantine consensus algorithm

```
1: bin-propose(val):
2:   repeat:
3:     (bv-broadcast(EST, r, val) → bin-values)
4:     start-timer(r)
5:     if i = r mod n then
6:       wait until (bin-values = {w})
7:       broadcast(COORD, r, w) → c
8:     wait until (bin-values ≠ ∅ ∧ timer expired)
9:     if c ∈ bin-values then e ← {c} else e ← bin-values
10:    broadcast(AUX, r, e) → bvals
11:    wait until ∃s ⊆ bvals where the two following conditions hold:
12:      • if v received in AUX msgs of n − f processes then v ∈ s
13:      • ∀v ∈ s, v ∈ e
14:    if s = {v} then
15:      val ← v
16:      if v = (r mod 2) and not decided yet then decide(v)
17:    else val ← (r mod 2)
18:    if decided in round r − 2 then exit()
19:    r ← r + 1
20:  until r is such that decide was invoked in round r − 2
```

5.6.6 Proof of Correctness

We show that our verified reliable broadcast ensures the properties of the reliable broadcast for valid values and discards invalid values, then we prove that our consensus protocol solves the Set Byzantine Consensus problem before showing that RBBC implements a replicated state machine (RSM).

Theorem 11 *The Verified Reliable Broadcast (lines 1–13) ensures the properties of the Reliable Broadcast for all valid values and does not deliver an invalid value at any correct proposer.*

Proof We proceed by showing the properties of reliable broadcast for a valid value: if a valid value is delivered then it was broadcast (validity), a correct proposer delivers at most one value from any given proposer (unicity), if a correct proposer broadcasts a valid value v then v is delivered at all correct

proposers (termination1) and if a correct proposer delivers a valid value v, then all correct proposers deliver v (termination2). It is easy to ensure validity and unicity, so let us focus on termination1 and termination2. Consider that a valid value v is broadcast by some proposer p_i. There are two cases to consider, either p_i is correct or Byzantine.

1. **Proposer p_i is correct.** Proposer p_i broadcasts INIT to all proposers, hence each proposer broadcasts ECHO to all proposers, and all correct proposers eventually receive ECHO messages with v from $n - f$ distinct correct proposers. These correct proposers, say Q, are thus ready to start verifying value v. As there are $f + 1$ primary verifiers and f secondary verifiers, there are up to $2f + 1$ verifiers, say Q', that will verify value v if not stopped at line 10. If $f + 1$ verifiers broadcast READY messages with the same verification outcome *verif*, then we know that all correct proposers will then retransmit READY with this *verif*, which will guarantee that all correct proposers will receive $n - f$ messages $\langle \text{READY}, verif, h(v), p_i \rangle$ and will thus deliver v (line 13). It thus remains to show that $f + 1$ verifiers will eventually broadcast READY messages with the same verification outcome *verif*. Note that $|Q \cap Q'| \geq f + 1$, which means that among the correct proposers Q, $f + 1$ of them verify v and obtain the same verification outcome *verif* that they broadcast.

2. **Proposer p_i is Byzantine.** First, if proposer p_i broadcasts INIT successfully to all $n - t$ correct proposers, in which case, they all broadcast ECHO with value v to all proposers and the case is identical to case (1), where all correct proposers deliver v (line 13). In the case where proposer p_i does not broadcasts INIT to $f + 1$ correct proposers because it broadcasts to less proposers, then not enough proposers will receive INIT for ECHO to be received by sufficiently many proposers at line 5 and v will not be delivered. Third, if proposer p_i broadcasts INIT to $f + 1 \leq \ell < n - f$ proposers, then it depends on the behaviors of the other Byzantine proposers, if sufficiently many of them send ECHO messages to $f + 1$ verifiers or if they help verifying correctly, then v will be delivered at all correct proposers, otherwise, it will not be delivered at any correct proposer.

To show that no invalid values can be delivered at any correct proposer, consider that v is invalid so there cannot be $f + 1$ distinct verifiers whose *verif* is identifying v as valid. As a result, if line 9 is enabled with $f + 1$ identical messages $\langle \text{READY}, verif, h(v), p_i \rangle$ from distinct proposers then we know that *verif* is necessarily identifying v as invalid and the precondition to deliver v at line 13 will not be satisfied. □

Lemma 12 *In the binary Byzantine consensus (lines 4–15), if at the beginning of a round r, all correct proposers have the same estimate val, they never change their estimate value thereafter.*

Proof Let us assume that all correct processes (which are at least $n - f > f + 1$) have the same estimate *val* when they start round r. Hence, they

all bv-broadcast the same message EST(val) either at line 2 or within the reliable broadcast at line 6. It follows from the properties of the reliable broadcast [Bra87] and bv-broadcast [MMR15b] that each correct process p_i is such that *bin-values*$_i = \{v\}$ at line 9, and consequently can broadcast only ECHO($\{v\}$) at line 10. Considering any correct process p_i, it then follows from the predicate of line 11 (s_i contains only v), the predicate of line 12 (s_i is a singleton), and the assignment of line 14, that val_i keeps the value v. □

The next lemma states that if the value s in the same round of two correct replicas are singletons then they are identical.

Lemma 13 *Let p_i and p_j be two correct proposers. In the Byzantine binary consensus (lines 4–15), if $s_i = \{v\}$ and $s_j = \{w\}$ in the same round, then $v = w$.*

Proof Let p_i be a correct proposer such that $s_i = \{v\}$. It follows from line 11 that p_i received the same message AUX ($\{v\}$) from $(n - t)$ different processes, i.e., from at least $(n - 2f)$ different correct processes. As $n - 2t \geq f + 1$, this means that p_i received the message AUX ($\{v\}$) from a set Q_i including at least $(f + 1)$ different correct proposers.

Let p_j be a correct proposer such that $s_j = \{w\}$. Hence, p_j received AUX ($\{w\}$) from a set Q_j of at least $(n - t)$ different proposers. As $(n - f) + (f + 1) > n$, 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 correct proposer. Hence, at line 10, p_k sent the same AUX message to p_i and p_j, and we consequently have $v = w$. □

Lemma 14 (Binary Consensus Validity) *In the Byzantine binary consensus (lines 4–15), the value decided by a correct proposer was proposed by a correct proposer.*

Proof Let us consider the round $r = 1$. Due to the property of the bv-broadcast executed at line 2 or piggybacked within the reliable broadcast at line 6, it follows that the sets *bin-values*$_i$ contains only values proposed by correct proposers. Consequently, the correct proposers broadcast, at line 10 AUX messages containing sets with values proposed only by correct proposers. It then follows from the predicate of line 11 ($s_i \subseteq$ *bin-values*$_i$), and the reliable and bv-broadcast properties, that the set s_i of each correct proposer contains only values proposed by correct proposers. Hence, the assignment of val_i (be it at line 13 or 15) provides it with a value proposed by a correct proposer. The same reasoning applies to rounds $r = 2$, $r = 3$, etc., which concludes the proof of the lemma. □

Lemma 15 (Binary Consensus Agreement) *In the Byzantine binary consensus (lines 4–15), no two correct replicas decide different values.*

Proof Let r be the first round during which a correct proposer decides, let p_i be a correct proposer that decides in round r (line 14), and let v be the value it decides. Hence, we have $s_i^r = \{v\}$ where $v = (r \bmod 2)$.

If another correct replica p_j decides during round r, we have $s_j^r = \{w\}$, and, due to Lemma 13, we have $w = v$. Hence, all correct proposers that decide in round r, decide v. Moreover, each correct proposer that decides in round r has previously assigned $v = (r \bmod 2)$ to its local estimate s_i.

Let p_j be a correct proposer that does not decide in round r. As $s_i^r = \{v\}$, and p_j does not decide in round r, it follows from Lemma 13 that we cannot have $s_j^r = \{1 - v\}$, and consequently $s_j^r = \{0, 1\}$. Hence, in round r, p_j executes line 13, where it assigns the value $(r \bmod 2) = v$ to its local estimate val_j.

It follows that all correct proposers start round $(r + 1)$ with the same local estimate $v = r \bmod 2$. Due to Lemma 12, they keep this estimate value forever. Hence, no different value can be decided in a future round by a correct proposer that has not decided during round r, which concludes the proof of the lemma. □

Lemma 16 *During the RBBC consensus (lines 1–10) execution, at least one binary consensus instance decides* 1*.*

Proof By Theorem 11, we know that all correct proposers eventually populate their proposal array with at least one common values. Due to the reduction, all correct proposers will thus have input 1 for the corresponding binary consensus instance. By the validity (Lemma 14) and termination [CGLR17] properties of the binary consensus, the decided value for this binary consensus instance has to be 1. □

Lemma 17 *If a Byzantine binary consensus instance at index i decides 1, then the verified reliable broadcast (lines 1–13) at index i reliably delivers a value at a correct proposer.*

Proof A correct proposer does not propose 1 to a binary consensus instance at index i without reliably delivering a proposal at index i. The result follows from Theorem 11 and the validity of the binary consensus (Lemma 14). □

Theorem 18 (Set Byzantine Consensus) *The RBBC consensus (lines 1–10) solves the Set Byzantine Consensus.*

Proof By Lemma 16, at one indice, a binary consensus instance terminates with 1. By Lemma 15, we know that all correct proposers have set 1 to the same indices of their *bitmask*. For each of these indices k there is a proposal $props[k]$ that will be delivered at every correct proposer by Lemma 17. As a result, all correct proposers invoke function reconciliate with the same argument at line 10. By examination of the code at lines 1–6, all correct proposers thus put in their *superblock* the same subset of valid and non-conflicting transactions hence guaranteeing agreement, termination and validity of the SBC problem (Section 5.4). □

Theorem 19 (Replicated State Machine) *The set of committed transactions of RBBC is totally ordered.*

Proof In RBBC, all permissioned nodes run the consensus algorithm either because they receive messages from proposers or because they propose themselves. Each node starts by running a single instance of this consensus algorithm for the block at index 1 (after the genesis block). A node can start a new consensus instance for a block at index $j > 1$ only after the consensus instance at index $j - 1$ has terminated. As Theorem 18 shows that consensus guarantees agreement there is a single block decided per index of the blockchain. It results that blocks are totally ordered through their index number: whenever a block is decided, it is ordered after all previously decided blocks. Given that in each block the transactions do not conflict and are ordered through the same deterministic strategy employed by all correct nodes (lines 1–6), transactions are totally ordered. □

Blockchain	deployment	network	throughput	latency	#nodes	#machines
Elastico [LNZ+16]	country-wide	emulated	20 KB/s	800 sec	1,600	800
Algorand [GHM+17]	country-wide	emulated	208 KB/s	12 sec	50,000	1000
Omniledger [KKJG+17b]	datacenter	emulated	1.7 MB/s	14 sec	1,800	60
RapidChain [ZMR18]	datacenter	emulated	3.6 MB/s	9 sec	4,000	32
RBBC	world-wide	real	11.7 MB/s	3 sec	9,400	1000

Table 5.1: Scalable blockchain experiments – the throughput of Elastico, Omniledger and RapidChain was previously observed to be 40 TPS, 3500 TPS and 7380 TPS for 512-byte transactions [ZMR18], the throughput of Algorand was reported to be 750 MB/h=208 KB/s [GHM+17], the throughput of RBBC is obtained from 30,684 TPS for 400-byte transactions [CNG18].

5.7 Conclusion

Classic solutions to the problem of consensus predate the blockchain era. Most of these solutions rely on a leader-based design that induces some overhead when propagating a proposal to a large network. In order to scale blockchain technologies, one has to rethink their consensus components rather than reusing off-the-shelf solutions. However, the consensus problem is quite challenging, which led researchers to try to shard the consensus: running multiple consensus instances among fewer nodes in parallel.

Red Belly Blockchain takes a different approach by offering a solution that scales to hundreds of consensus nodes by leveraging a leaderless consensus design and verification sharding. Its consensus, called Democratic BFT, has been designed for blockchains. Red Belly Blockchain adapted this algorithm

to combine multiple proposals from different nodes into a superblock without the need of a special leader node. The performance obtained is compared to alternative approaches, including sharding ones, in Table 5.1.

5.8 Bibliographic notes

DBFT was presented in [CGLR18] and Red Belly Blockchain has been evaluated in [CNG18].

Most partially synchronous Byzantine consensus solutions are leader-based [CL02, KAD+07, CMSK07, VCB+09, MJM09, AGK+15b, BSA14, YMR+19]. Some tentatives tried to solve the Byzantine consensus without a leader, however, these are either not detailed [Lam11] or experience exponential complexity [BS10]. To bypass the limitation of these approaches, the idea has been to run multiple instances in parallel, a technique called *sharding*. Elastico [LNZ+16], Omniledger [KKJG+18], RapidChain [ZMR18] and Monoxide [WW19] are examples of sharded blockchains. Sharding does not remedy the problem of having to order cross-shard transactions. In particular, Monoxide does not offer atomic cross-shard transactions.

The traditional consensus definition unnecessarily limits the scalability of the blockchain [Vuk16, Buc16, SBV18]: most blockchains decide at most one of the proposed blocks [Buc16, SBV18]. The scalability problem is even exacerbated with *batching*, the act of waiting to gather more than one transaction requests to send a block containing multiple transactions. The leader bottleneck effect has been observed empirically multiple times [HKJR10, GBFS16, VG19].

To decide a superblock that combines all proposed blocks, one may think of solving a variant of the consensus problem to instead combine proposals, sent across disjoint communication links, into a decision [BOKR94, NCV05, CGLR17]. For example, the related problems of Agreement on a Core Set or Asynchronous Common Subset (ACS) [BOKR94], Interactive Consistency (IC) [LSP82] and Vector Consensus (VC) [NCV05] all require at least $f + 1$ (either $n - f$ or $f + 1$ with $n > 3f$) proposed values to be decided. In blockchain, however, there may not even be $f + 1$ compatible proposed blocks. SBC-Validity is inspired by the external validity property [CKPS01] that requires a decision to be valid and the idea of deciding at least $f + 1$ proposed values [LSP82, BOKR94], however, SBC-Validity cannot result from any combination of these properties. The full liveness proof of DBFT can be found in [CGLR17].

Bibliography

[AGK+15] 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.

[BOKR94] 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.

[Bra87] Gabriel Bracha. Asynchronous byzantine agreement protocols. *Inf. Comput.*, 75(2):130–143, November 1987.

[BS10] 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.

[BSA14] Alyson Bessani, Joao 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.

[Buc16] Ethan Buchman. Tendermint: Byzantine fault tolerance in the age of blockchains, 2016. MS Thesis.

[CGLR17] Tyler Crain, Vincent Gramoli, Mikel Larrea, and Michel Raynal. (leader/randomization/signature)-free byzantine consensus for consortium blockchains. Technical report, arXiv, 2017.

[CGLR18] Tyler Crain, Vincent Gramoli, Mikel Larrea, and Michel Raynal. DBFT: Efficient leaderless byzantine consensus and its applications to blockchains. In *Proceedings of the 17th IEEE International Symposium on Network Computing and Applications (NCA'18)*, 2018.

[CKPS01] 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.

[CL02] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance and proactive recovery. *ACM Trans. Comput. Syst.*, 20(4):398–461, 2002.

[CMSK07] Byung-Gon Chun, Petros Maniatis, Scott Shenker, and John Kubiatowicz. Attested append-only memory: Making adversaries stick to their word. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, pages 189–204, New York, NY, USA, 2007. ACM.

[CNG18] Tyler Crain, Christopher Natoli, and Vincent Gramoli. Evaluating the red belly blockchain. Technical Report 1812.11747, arXiv, 2018.

[GBFS16] Vincent Gramoli, Len Bass, Alan Fekete, and Daniel Sun. Rollup: Non-disruptive rolling upgrade with fast consensus-based dynamic reconfigurations. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 27(9):2711–2724, Sep 2016.

[GHM+17] Yossi Gilad, Rotem Hemo, Silvio Micali, Georgios Vlachos, and Nickolai Zeldovich. Algorand: Scaling byzantine agreements for cryptocurrencies. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 51–68, 2017.

[HKJR10] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *ATC*, pages 11–11. USENIX, 2010.

[KAD+07] 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.

[KKJG+17] Eleftherios Kokoris-Kogias, Philipp Jovanovic, Linus Gasser, Nicolas Gailly, Ewa Syta, and Bryan Ford. Omniledger: A secure, scale-out, decentralized ledger via sharding. Cryptology ePrint Archive, Report 2017/406, 2017. https://eprint.iacr.org/2017/406.

[KKJG+18] Eleftherios Kokoris-Kogias, Philipp Jovanovic, Linus Gasser, Nicolas Gailly, Ewa Syta, and Bryan Ford. Omniledger: A secure, scale-out, decentralized ledger via sharding. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 583–598, 2018.

[Lam11] Leslie Lamport. Brief announcement: Leaderless byzantine paxos. In *Distributed Computing - 25th International Symposium, DISC 2011, Rome, Italy, September 20-22, 2011. Proceedings*, pages 141–142, 2011.

[LNZ+16] Loi Luu, Viswesh Narayanan, Chaodong Zheng, Kunal Baweja, Seth Gilbert, and Prateek Saxena. A secure sharding protocol for open blockchains. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 17–30, 2016.

[LSP82] Leslie Lamport, Robert Shostak, and Marshall Pease. The byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401, July 1982.

[MJM09] Yanhua Mao, Flavio P. Junqueira, and Keith Marzullo. Towards low latency state machine replication for uncivil wide-area networks. In *In Workshop on Hot Topics in System Dependability*, 2009.

[MMR15] Achour Mostéfaoui, Hamouma Moumen, and Michel Raynal. Signature-free asynchronous binary byzantine consensus with $T < N/3$, $O(N^2)$ messages, and $O(1)$ expected time. *J. ACM*, 62(4):31:1–31:21, September 2015.

[Nak08] Satoshi Nakamoto. Bitcoin: a peer-to-peer electronic cash system, 2008. http://www.bitcoin.org.

[NCV05] 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.

[SBV18] Joao 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.

[VCB+09] Giuliana Veronese, Miguel Correia, Alysson Bessani, Lau Cheuk Lung, and Paulo Verissimo. Minimal byzantine fault tolerance. Technical Report TR-2009-15, DI-FCUL, June 2009.

[VG19] Gauthier Voron and Vincent Gramoli. Dispel: Byzantine SMR with distributed pipelining. Technical Report 1912.10367, arXiv, 2019.

[Vuk16] Marco Vukolíc. The quest for scalable blockchain fabric: Proof-of-work vs. BFT replication. In *Proceedings of the IFIP WG 11.4 Workshop on Open Research Problems in Network Security (iNetSec 2015)*, LNCS, pages 112–125, 2016.

[WW19] Jiaping Wang and Hao Wang. Monoxide: Scale out blockchains with asynchronous consensus zones. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 95–112, Boston, MA, February 2019. USENIX Association.

[YMR+19] Maofan Yin, Dahlia Malkhi, Michael K. Reiter, Guy Golan-Gueta, and Ittai Abraham. HotStuff: BFT consensus with linearity and responsiveness. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, pages 347–356, 2019.

[ZMR18] Mahdi Zamani, Mahnush Movahedi, and Mariana Raykova. Rapidchain: Scaling blockchain via full sharding. Cryptology ePrint Archive, Report 2018/460, 2018. https://eprint.iacr.org/2018/460.