Journal of Parallel and Distributed Computing 176 (2023) 95–113

Contents lists available at ScienceDirect

Journal of Parallel and Distributed Computing

journal homepage: www.elsevier.com/locate/jpdc

Leaderless consensus

Karolos Antoniadis [a], Julien Benhaim [a], Antoine Desjardins [a], Poroma Elias [a], Vincent Gramoli [a,b,*], Rachid Guerraoui [a], Gauthier Voron [a], Igor Zablotchi [c]

[a] EPFL, Switzerland
[b] University of Sydney, Australia
[c] MIT, USA

ARTICLE INFO

Article history:
Received 17 January 2022
Received in revised form 18 November 2022
Accepted 26 January 2023
Available online 23 February 2023

Keywords:
Leaderless termination
Byzantine
Synchronous-*k*
Synchronizer
Fast-path

ABSTRACT

Classic synchronous consensus algorithms are leaderless in that processes exchange proposals, choose the maximum value and decide when they see the same choice across a couple of rounds. Indulgent consensus algorithms are typically *leader-based*. Although they tolerate unexpected delays and find practical applications in blockchains running over an open network like the Internet, their performance is highly dependent on the activity of a single participant.

This paper asks whether, under eventual synchrony, it is possible to deterministically solve consensus without a leader. The fact that the weakest failure detector to solve consensus is one that also eventually elects a leader seems to indicate that the answer to the question is negative. We prove in this paper that the answer is actually positive.

We first give a precise definition of the very notion of a leaderless algorithm. Then we present three indulgent leaderless consensus algorithms, each we believe interesting in its own right: (i) for shared memory, (ii) for message passing with omission failures and (iii) for message passing with Byzantine failures.

Finally, we implement a Byzantine fault tolerant (BFT) state machine replication (SMR), that is leaderless. Our empirical results demonstrate that it is faster and more robust than HotStuff, the recent BFT SMR algorithm used in the Facebook Libra blockchain when deployed in a wide area network.

1. Introduction

Consensus algorithms that are designed for an eventually synchronous system, coined *indulgent* algorithms, tolerate an adversary that can delay processes for an arbitrarily long period of time [36,30,7,39,29,28,2,13,19,46,45]. Recently, these algorithms gained traction to maintain blockchain safety despite communication delays [12,47,20]. A common characteristic of these algorithms is that they all rely on a *leader*. Essentially, the leader helps processes converge towards a decision and it usually does so in a *fast* manner when the system is initially synchronous and there is neither failure nor contention. The drawback arises in the other cases: as the leader slows down, so does its consensus execution.

Basically, the requirement for a leader in existing indulgent algorithms represents a weakness that the adversary can exploit to significantly delay any decision. Accurately detecting a faulty leader is impossible during asynchronous periods. Moreover, the choice of the timeout to suspect a faulty leader and replace it impacts performance drastically [30,41], sometimes by two orders of magnitude [28]. Besides, replacing the leader requires a view-change protocol that is so complex that research prototypes often omit it [21] or suffer from errors [2].

With the advent of blockchains aimed at running in open networks, various efforts have been recently devoted to minimize the role of the leader in an eventually synchronous system. One idea is to change the leader frequently even if it is not suspected to have failed [46,13]. Another is to bypass the leader bottleneck by having multiple proposers [19,45,17] before reverting to a weak coordinator to converge. A third one is to tolerate multiple leaders for different consensus instances [36,39,28], however, it only eliminates the leader from the state machine replication (SMR) algorithm, not from the underlying consensus algorithm for a single SMR slot. None of these approaches manages to eliminate the leader.

* Corresponding author.
E-mail addresses: karolos.antoniadis@alumni.epfl.ch (K. Antoniadis), julien.benhaim@epfl.ch (J. Benhaim), antoinedesjard@gmail.com (A. Desjardins), elias.poromawiri@alumni.epfl.ch (E. Poroma), vincent.gramoli@sydney.edu.au (V. Gramoli), rachid.guerraoui@epfl.ch (R. Guerraoui), gauthier.voron@epfl.ch (G. Voron), igorz@mit.edu (I. Zablotchi).

https://doi.org/10.1016/j.jpdc.2023.01.009
0743-7315/

This raises a fundamental question. Is it possible to eliminate the leader from a deterministic indulgent consensus algorithm? Two reasons might lead to believe that the answer is negative. First, the weakest failure detector to solve consensus has been shown to be an eventual leader [16]. Second, when seeking the weakest amount of synchrony needed to solve consensus, it was shown that one correct process must have as many eventually timely links as there can be failures (some sort of leader) [3,11].

The main contribution of this paper is to show that it is actually possible to devise and implement a leaderless indulgent consensus algorithm.

First, to address this question, we formally define the notion of "leaderless", that has been informally understood as the ability to cope with the delay caused by a malicious process [10,33,19,42]. We believe this definition to be of independent interest. Intuitively, a leaderless algorithm is one that should be robust to the repeated slow-downs of individual processes. We introduce the synchronous$-k$ (which reads "synchronous minus k") round-based model where executions are (eventually) synchronous and at most $k < n$ processes can be suspended per round. We define a *leaderless* algorithm as one that decides in an eventually synchronous-1 (denoted by $\diamond$synchronous-1) system. In a synchronous-1 system, the classical idea of exchanging values in rounds and adopting the maximum one would not work, because the adversary can suspend the process with the maximum value for as long as it wants.

Then we present three leaderless consensus algorithms, each for a specific setting. The first algorithm, called *Archipelago*,[1] works in shared memory and builds upon a new variant of the classical adopt-commit object [26] that returns maximum values to help different processes converge towards the same output. Interestingly, the algorithm requires $n \geq 3$ processes, which is not common for shared memory algorithms. The second algorithm is a generalization of Archipelago in a message passing system with omission failures. The third algorithm, called *BFT-Archipelago*, is a generalization of Archipelago for Byzantine failures. This algorithm shares the same asymptotic communication complexity as a classic Byzantine fault tolerant consensus algorithm [15] and can execute optimistically a fast path to terminate in two message exchanges under good conditions. Interestingly, all our algorithms are optimal both in terms of resilience and time complexity.

Finally, we propose a State Machine Replication (SMR) implementation of BFT-Archipelago in order to demonstrate the practicality of our approach. To this end, we deploy the BFT-Archipelago SMR in a geo-distributed setting and compare its performance to the HotStuff SMR [47] that recently inspired the development of the Libra blockchain initially proposed by Facebook. Since the HotStuff SMR features pipelining, which allows to start a new consensus instance before the preceding one is complete, we also implemented pipelining in the BFT-Archipelago SMR. Our results indicate that the BFT-Archipelago SMR outperforms HotStuff SMR when deployed across four distinct data centers. In addition, the performance of the BFT-Archipelago SMR is maintained even after isolated failures while the performance of HotStuff is negatively impacted by the same isolated failures, confirming the advantages of a leaderless SMR.

The rest of the paper is organized as follows. Section 2 gives some necessary background. Section 3 formalizes the notion of a leaderless consensus algorithm and explains why well-known leader-based consensus algorithms do not satisfy this definition. Section 4 presents a leaderless consensus algorithm for shared memory. Section 5 presents a leaderless consensus algorithm to tolerate omission failures in message passing. Section 6 presents a leaderless consensus algorithm to tolerate Byzantine failures. Section 7 discusses the complexities of our algorithms. Section 8 presents the experimental results of the state machine replication based on the Byzantine fault tolerant consensus algorithm. Section 9 discusses related work while Section 10 concludes.

[1] Unlike in Paxos, whose name refers to a unique island and where a unique leader plays the most decisive role, in Archipelago, whose name refers to a group of islands, all nodes play an equally decisive role.

2. Preliminaries

We first consider an *asynchronous* shared-memory model with n processes $\mathcal{P} = \{p_1, p_2, \ldots, p_n\}$. Processes have access to (an infinite) set $\mathcal{R}$ of atomic registers that can each store values from a set $\mathcal{V}$. Initially, all registers contain the initial value $\perp$. For simplicity of notation, we assume that $\mathcal{R}$ includes an infinite set of single-writer multi-reader (SWMR) arrays of n registers each. We denote these arrays as $\mathcal{R}_1, \mathcal{R}_2, \ldots$ where a process p_i can write locations $\mathcal{R}_1[i], \mathcal{R}_2[i], \ldots$. Processes *communicate* by reading from and writing to atomic registers. A *process* is a state machine that can change its state as a result of reading a register or writing to a register. An *algorithm* is the state machine of each process. A *configuration* corresponds to the state of all processes and the values in all registers in $\mathcal{R}$. An *initial configuration* is a configuration where all processes are in their initial state and all registers in $\mathcal{R}$ contain value $\perp$.

When a process p invokes a *read*or a *write*operation, we say that p performs a read or write *event* respectively. An *execution* corresponds to an alternating sequence of configurations and events, starting from an initial configuration. For example, in the execution $\alpha = C, \mathsf{read}(r, v)_p, C', \mathsf{write}(r', v')_{p'}, C''$ we have processes $p, p' \in \mathcal{P}$, registers $r, r' \in \mathcal{R}$, values $v, v' \in \mathcal{V}$, and configurations C, C', C'' where C is an initial configuration, and the system moves from configuration C to C' when p reads v from r and from C' to C'' when p' writes v' to r'. We assume that all executions are *well-formed*, hence for a process p to perform an event after configuration C in an execution, there must be a transition specified by p's state machine from p's state in C. In this work, we consider deterministic algorithms and hence the initial state of processes and the sequence of processes that take steps uniquely define a single well-formed execution. For the sake of simplicity, we represent an execution as a sequence of steps and omit the configurations.

An execution α' is called an *extension* of a finite execution α if α is a prefix of α'. Two executions α and β are *equal* if both executions contain the same configurations and events in the same order.

Synchronous$-k$ execution. We can now define what it means for an execution to be synchronous in shared-memory. Our definition is inspired by the notion of synchrony in a message passing model where there is a bound on the time needed for a message to propagate from one process to another and for the receiver to process this message. In a message passing model, we can divide time into rounds [23] such that, in each round, every process p: (i) sends a message to every other process in the system, and (ii) delivers any message that was sent to p and performs local computation.

To adapt synchrony to the shared memory model, we also assume that processes take steps in rounds. Specifically, in each round, every process p_i (i) performs a write in some $\mathcal{R}_j[i]$ and (ii) collects all the values written in array $\mathcal{R}_j$. The collect operation is useful to reason later in terms of message passing, where collecting from all cells of an array is similar to reading the messages received from all processes. In one round, different processes can read from different arrays.

More precisely, a *collect* by a process p_i on an array $\mathcal{R}_j$ is defined as a sequence of n read events: $\mathsf{collect}(\mathcal{R}_j)_{p_i} = \mathsf{read}(\mathcal{R}_j[1], \cdot)_{p_i}, \ldots, \mathsf{read}(\mathcal{R}_j[n], \cdot)_{p_i}$. Notation "$\cdot$" indicates any

	1	2	3	4	5	6	7	8	9	10	11
p_1	$step(\mathcal{R}_5)_{p_1}$	X	$step(\mathcal{R}_2)_{p_1}$	$step(\mathcal{R}_6)_{p_1}$	$step(\mathcal{R}_3)_{p_1}$	X	$step(\mathcal{R}_3)_{p_1}$	$step(\mathcal{R}_2)_{p_1}$	$step(\mathcal{R}_1)_{p_1}$	$step(\mathcal{R}_4)_{p_1}$	$step(\mathcal{R}_1)_{p_1}$
p_2	$step(\mathcal{R}_2)_{p_2}$	$step(\mathcal{R}_4)_{p_2}$	X	X	X	$step(\mathcal{R}_2)_{p_2}$	$step(\mathcal{R}_1)_{p_2}$	X	X	X	X

Fig. 1. Graphical depiction of a synchronous−1 execution.

value. We define a *step* of $\mathcal{R}_j$ by a process p_i as a write event and then a collect on $\mathcal{R}_j$. So, $\mathsf{step}(\mathcal{R}_j)_{p_i} = \mathsf{write}(\mathcal{R}_j[i], \cdot)_{p_i}, \mathsf{collect}(\mathcal{R}_j)_{p_i}$. A *round* consists of all the write events $\mathsf{write}(\mathcal{R}_{j_1}[1], \cdot)_{p_1}, \ldots, \mathsf{write}(\mathcal{R}_{j_n}[n], \cdot)_{p_n}$, followed by a sequence $\mathsf{collect}(\mathcal{R}_{j_1})_{p_1}, \ldots, \mathsf{collect}(\mathcal{R}_{j_n})_{p_n}$ of collects by the exact same processes that performed a write event. Note that indices j_a and j_b could be the same for $a \neq b$. For example, if we only consider two processes $\{p_1, p_2\}$, then a round r could be the following sequence of events $r = \mathsf{write}(\mathcal{R}_{j_1}[1], \cdot)_{p_1}, \mathsf{write}(\mathcal{R}_{j_2}[2], \cdot)_{p_2}, \mathsf{collect}(\mathcal{R}_{j_1})_{p_1}, \mathsf{collect}(\mathcal{R}_{j_2})_{p_2}$.

To capture that a process is *suspended* in a round r, we denote by $r|_{-\mathcal{P}_s}$ all the steps except the ones taken by processes in $\mathcal{P}_s$. For instance, for the above sequence r, we have $r|_{-\{p_1\}} = \mathsf{write}(\mathcal{R}_{j_2}[1], \cdot)_{p_2}, \mathsf{collect}(\mathcal{R}_{j_2})_{p_2}$.

We say that an execution is *synchronous*$-k$ (which reads "synchronous minus k") if α is the steps of a sequence of rounds $r_1|_{-\mathcal{P}_{s_1}}, r_2|_{-\mathcal{P}_{s_2}}, r_3|_{-\mathcal{P}_{s_3}}, \ldots$ and $|\mathcal{P}_{s_i}| \leq k$ for $i \geq 1$. In other words, at most k processes can be *suspended* in each round. A suspended process p in a round r does not perform all the events in r; i.e. it may perform some, most or none of the actions of the round, but not all of them. For this reason, we call such an execution "synchrony minus k", since all processes except k behave synchronously in each round. However, up to all of the n processes may, in turn, be suspended at some point during the execution. We say that an infinite execution α is *eventually synchronous*$-k$ (or $\diamond$synchronous$-k$) if an infinite suffix of α is equal to a synchronous$-k$ execution. Naturally, a synchronous$-k$ execution for $k = 0$ corresponds to a fully synchronous execution, while synchronous$-k$ with $k > 0$ allows for some asynchrony in an execution.

In a synchronous$-k$ or $\diamond$synchronous$-k$ execution α, we say that a round r' occurs after round r if the events of round r' appear after the events of round r in α.

We say that a process p is *correct* in an infinite execution α if p is not suspended forever in α. More precisely, a process p is *correct* in an infinite execution if, for every round r there exists a later round r' such that process p is not suspended in r'.

Example. Fig. 1 depicts a synchronous−1 execution for two processes p_1 and p_2 that take steps in a sequence starting from round 1 and ending in round 11. The X symbol in a round indicates that the process is suspended in this round. In Fig. 1, both processes perform steps in the first round, p_1 in array $\mathcal{R}_5$ and p_2 in $\mathcal{R}_2$. Then, in the next round, process p_1 is suspended, etc.

Omission and Byzantine fault models. A process is *faulty* in the omission model if it may at some point of the execution omit sending some message, or in the Byzantine model if it can behave arbitrarily, except impersonating another process.

Consensus. In consensus [14], each process proposes a value by invoking a propose(v) function and then all processes have to decide on a single value. Consensus is defined by the following three properties. *Validity* states that a value decided was previously proposed. *Agreement* states that no two processes decide different values, and *termination* states that every correct process eventually decides. We say that a consensus algorithm *decides* in an execution α if a propose(v) function call by some process p returns in α.

3. Defining a leaderless algorithm

We are now ready to define a *leaderless* consensus algorithm. We define it as a consensus algorithm that terminates despite an adversary suspending one process per round, defined as $\diamond$synchronous−1 in the previous section. To the best of our knowledge, this is the first formal definition of what "leaderless" means.

This definition stems from the intuition that a unique process—the leader—must perform some round for a "leader-based" consensus algorithm to decide. In other words, a leader-based consensus algorithm cannot terminate if an adversary can selectively suspend a process the moment it becomes the leader. We thus introduce termination despite such an adversary as a new liveness property:

Definition 1 *(Leaderless termination).* A consensus algorithm $\mathcal{A}$ satisfies leaderless termination if, in every $\diamond$synchronous−1 execution of $\mathcal{A}$, every correct process decides.

Intuitively, an algorithm that decides despite an adversary suspending one process per round has to be leaderless. This is why, we say that a consensus algorithm is leaderless if it is a consensus algorithm that satisfies safety (validity and agreement) and leaderless termination as follows.

Definition 2 *(Leaderless algorithm).* A consensus algorithm is *leaderless* if it satisfies validity and agreement as well as leaderless termination.

Leaderless termination implies termination as termination is simply leaderless termination in a $\diamond$synchronous−0 execution. Hence a leaderless algorithm also satisfies termination, but termination does not imply leaderless termination.

By contrast, a consensus algorithm that is not leaderless, is called *leader based*. We extend Definition 2 to the message-passing model in Section 5. An important aspect of Definition 2 is that it makes a leaderless consensus algorithm robust against the adaptive behavior of a dynamic adversary. In particular, an alternative definition of a leaderless consensus algorithm as an algorithm that decides in the exact same number of rounds irrespective of which process crashes (or gets suspended forever), would not share the same robustness.

Why leaderless termination is not sufficient. An important remark is now in order. Leaderless termination is not implied by the classical notion of termination. To illustrate this, we present Algorithm 1, a consensus algorithm that decides in every synchronous−1, but that violates safety (i.e., agreement) when executed in an $\diamond$synchronous−1 execution. Clearly, Algorithm 1 does not have a distinguished (leader) process that drives the decision, and the algorithm decides in two rounds if the system is synchronous−1. However, this algorithm is not leaderless according to Definition 1, because it does not tolerate asynchrony: in an $\diamond$synchronous−1 the algorithm can violate safety.

First, we prove that Algorithm 1 satisfies validity, agreement, and decides in finite time in every synchronous−1 execution:

- *Validity.* Each process writes the proposed value in $Reg[i]$ (line 6) and then collects (line 7) all the values written in

Algorithm 1 Consensus algorithm that correctly decides in every synchronous−1 execution.

```
1: Shared state:
2:   Reg[n] ← {⊥, ..., ⊥}  ▷ array of n single-writer multi-reader registers

3: ▷ process p_i proposes value v
4: Procedure propose(v):
5:   ▷ first round
6:   Reg[i] ← v
7:   vals ← collect(Reg) \ {⊥}
8:   if ∃⟨commit, cv⟩ ∈ vals then
9:     dv ← cv  ▷ p_i was suspended in the first round, hence adopt committed value
10:  else
11:    dv ← max({v : v ∈ vals ∨ ⟨·, v⟩ ∈ vals})
12:
13:  ▷ second round
14:  Reg[i] ← ⟨commit, dv⟩
15:  return dv
```

Reg. Hence, variable *vals* contains only proposed values. Then, if there is a $\langle \text{commit}, cv \rangle$ pair in *vals* the algorithm decides cv, stores $\langle \text{commit}, cv \rangle$ in $Reg[i]$ and returns (lines 8, 9, and 14). Otherwise, the algorithm retrieves the maximum value stored in *vals*, and hence retrieves a proposed value (line 11). The process then stores $\langle \text{commit}, cv \rangle$ in $Reg[i]$ and returns (line 14).

- *Agreement.* Algorithm 1 satisfies agreement in a model with $n \geq 3$ processes. In a model with $n \geq 3$ processes, at least one process p performs steps in both rounds one and two. Process p writes $\langle \text{commit}, v \rangle$ (line 14) in the second round and the algorithm decides v. If multiple processes were unsuspended in the first round, then all of the processes retrieve the same maximum value (line 11), and hence write the exact same $\langle \text{commit}, dv \rangle$ pair in the second round (line 14). Any process that was suspended in the first or second round, reads the committed value (line 9) and hence decides on the same value.

Then we show how Algorithm 1 could violate agreement with $n = 2$ processes, even in a synchronous−1 execution. For example, assume two processes p_1 and p_2 that propose v and v' respectively (with $v < v'$). Then, consider that process p_2 is suspended in the first round and process p_1 is suspended in the second round. Both processes p_1 and p_2 are unsuspended in the third round. In such an execution, p_1 writes v to $Reg[0]$ and then retrieves the maximum value in *Reg*, which is v. Then, in the second round, process p_2 writes v' to $Reg[1]$ and retrieves the maximum value in *Reg*, which now is v'. Hence in the third round, processes p_1 and p_2 decide v and v' respectively.

The challenge is, instead, to devise a leaderless consensus algorithm that decides in finite time in every ◇synchronous−1 execution and never violates safety. In the next sections, we present three leaderless consensus algorithms that tolerate omissions in shared memory, omissions in message passing and Byzantine failures.

The pros and cons of being leaderless. With the property of being leaderless comes various advantages for practical systems: avoiding leader bottlenecks [19,9] and reducing the impact of a single point of failure on performance [7,45] are well-known advantages that add to the aforementioned robustness. But are there drawbacks of being leaderless? For example, are there fault models for which leaderless algorithms do not exist? Actually, we present several leaderless consensus algorithms that tolerate classic types of faults in the partially synchronous model. One might also ask whether leaderless algorithms induce a higher complexity than leader-based ones. It turns out that our algorithms are both time optimal and resilience optimal. In addition, our Byzantine fault tolerant leaderless algorithm, BFT-Archipelago, shares the same communication complexity as PBFT [15] and DBFT [19], namely $O(n^4)$ bits. Note, however, that a recent leader-based consensus protocol [18] achieved $O(n^2)$ communication complexity, indicating that PBFT is not optimal, however, we are not aware of the optimal communication complexity of a leaderless consensus protocol. Finally, since BFT-Archipelago can be written as an Abstract [8] (see Section 7), it is compatible with leader-based consensus instances and inherits an optimal fast path in good executions.

Algorithm 2 Leader-based consensus algorithm.

```
1: Shared state:
2:   R[n] ← {⟨⊥, 0⟩, ..., ⟨⊥, 0⟩}  ▷ 1 SWMR reg. per proc.

3: Local state:
4:   ts ← i  ▷ for process p_i

5: Procedure propose(v):  ▷ process p_i proposes value v
6:   while true do
7:     R[i].ts ← ts
8:     val ← getHighestTspValue(R)
9:     if val = ⊥ then
10:      val ← v
11:    R[i] ← ⟨val, ts⟩
12:    if ts = getHighestTsp(R) then
13:      return val
14:    ts ← ts + n
```

Paxos: a counter example. Consider Algorithm 2, a leader-based algorithm that, when combined with a leader election, corresponds to Paxos [31] in shared memory (or more specifically to Disk Paxos [27] with a single non-faulty disk).

All processes share an array R of n single-writer multi-reader (SWMR) registers (line 2), each storing a pair $\langle a, b \rangle$ associating value a to timestamp b. Each process also maintains a ballot number as a local *ts* value (line 4). When a process p_i invokes propose(v), it executes a prepare phase and a propose phase [32]. During the prepare phase, p_i stores its current timestamp value to $R[i]$ (line 7) and either retrieves the value *val* of R associated with the highest timestamp (line 8), or (if no such value exists) sets *val* to its own value v. During the propose phase, p_i stores the pair $\langle val, ts \rangle$ to array $R[i]$ (line 11) and examines whether the highest timestamp in R is the one that p_i wrote (line 12). If this is the case, the algorithm decides (line 13), otherwise p_i increases *ts* and repeats the loop (line 14).

According to Definition 2, Algorithm 2 is leader based. In fact, Algorithm 2 does not terminate if an adversary suspends a process p when it is about to check whether its timestamp *ts* is the highest timestamp (line 12) and until some other process p' stores a timestamp $ts' > ts$ in array R (line 7).

4. Archipelago: a leaderless consensus algorithm

In the following sections, we present a series of leaderless consensus algorithms. For pedagogical reason we start, in this section, by presenting a simple shared memory leaderless consensus algorithm, called Archipelago, before its message-passing variant. Archipelago satisfies Definition 1 when $n \geq 3$ and never violates safety. It builds upon a new variant of an adopt-commit object [26], called *adopt-commit-max*, whose invocations by different processes help them converge towards the same output value without a leader.

Adopt-commit-max implementation. The *adopt-commit object* [26] has the following specification. Every process p proposes an input value to such an object and obtains an output, which consists of a pair $\langle d, v \rangle$; d can be either commit or adopt. The following properties are satisfied:

- **CA-Validity**: If a process p obtains output $\langle \text{commit}, v \rangle$ or $\langle \text{adopt}, v \rangle$, then v was proposed by some process.

Algorithm 3 The adopt-commit-max algorithm.

```
1:  Shared state:
2:    A and B, two arrays of n single-writer multi-reader
3:        registers, all initially ⊥

4:  Procedure propose(v): ▷ taken by a process p_i
5:    A[i] ← v   ▷ step A starts
6:    S_A ← collect(A)   ▷ step A ends
7:    if (S_A \ {⊥} = {v'}) then   ▷ step B starts
8:      B[i] ← ⟨commit, v'⟩
9:    else B[i] ← ⟨adopt, max(S_A)⟩ ▷ or step B starts
10:   S_B ← collect(B)   ▷ step B ends
11:   if S_B \ {⊥} = {⟨commit, v'⟩} then
12:     return ⟨commit, v'⟩
13:   else if ⟨commit, v'⟩ ∈ S_B then return ⟨adopt, v'⟩
14:   else return ⟨adopt, max(S_B)⟩
```

- **CA-Agreement**: If a process p outputs $\langle \text{commit}, v\rangle$ and a process q outputs $\langle \text{commit}, v'\rangle$ or $\langle \text{adopt}, v'\rangle$, then $v = v'$.
- **CA-Commitment**: If every process proposes the same value, then no process may output $\langle \text{adopt}, \cdot\rangle$.
- **CA-Termination**: Every correct process eventually obtains an output.

Algorithm 3 depicts a new implementation of an adopt-commit object. It differs from the classic implementation [26] in that if the collect of A by process p that proposes v returns different values, then p stores $\langle \text{adopt}, mv\rangle$ to array B (line 9) instead of storing $\langle \text{adopt}, v\rangle$, where mv is the maximum of the values collected from A ($max(S_A)$). Additionally, if all pairs collected from B are of the form $\langle \text{adopt}, \cdot\rangle$, then process p returns $\langle \text{adopt}, mv\rangle$, where mv is $max(S_A)$ (line 14). Note that Algorithm 3 is just a different implementation of the classic implementation [26] and that the main properties of an adopt-commit object remain the same. These modifications are crucial for the leaderless termination of Archipelago.

Correctness of the adopt-commit-max object. We now present the proof of correctness of Algorithm 3, which is similar to that of an adopt-commit object [26]. Algorithm 3 satisfies CA-Validity (the max function preserves validity) and CA-Termination (Algorithm 3 does not use waiting or loops). To prove CA-Agreement and CA-Commitment, we first prove the following lemma.

Lemma 4.1. *If B contains two entries* (commit, v_1) *and* (commit, v_2), *then* $v_1 = v_2$.

Proof. Assume not. Since every process writes in A and B at most once, it must be that some process p_1 wrote (commit, v_1) and some other process p_2 wrote (commit, v_2). Thus, it must be that p_1 wrote v_1 in A, took a collect of A and only saw v_1 in that collect. Similarly, it must be that p_2 wrote v_2 in A, took a collect of A and only saw v_2 in that collect. This is impossible: since the processes update A before collecting, it must be that either p_1 saw p_2's value, or vice-versa. We have reached a contradiction. □

- CA-Agreement. In order for a process p to commit v, p must write v to A, collect A and see only entries equal to v; p must then write $\langle \text{commit}, v\rangle$ to B, collect B and see only entries equal to $\langle \text{commit}, v\rangle$ and finally return $\langle \text{commit}, v\rangle$.

  Assume by contradiction that process p commits v and some process q commits or adopts $v' \neq v$. q's collect of B cannot include the $\langle \text{commit}, v\rangle$ entry written by p, otherwise q would adopt v (remember that by Lemma 4.1, q cannot see any entry $\langle \text{commit}, v'\rangle$ with $v' \neq v$ in B since p writes $\langle \text{commit}, v\rangle$ to b). Therefore, q's collect of B must happen before p's write to B. Furthermore, q's collect of B must include some entry $e = \langle \cdot, v'\rangle$ with $v' \neq v$ (written either by q or some other process). But then p's collect of B (which is after p's write to B and therefore after q's collect of B) will also include e, and thus p cannot commit v. We have reached a contradiction.
- CA-Commitment. Assume all proposed values are equal. Then no process can write $\langle \text{adopt}, \cdot\rangle$ in B; B contains only entries of the form $\langle \text{commit}, \cdot\rangle$. By Lemma 4.1, all such entries have equal values, so all processes that return must commit.

Algorithm 4 Archipelago leaderless consensus.

```
15: Shared state:
16:   C[0, ..., +∞], an infinite array of adopt-commit-max
17:       objects in their initial state
18:   m, a max register object that initially contains ⟨0, ⊥⟩.
19:   Note that ⟨x, y⟩ > ⟨x', y'⟩ if x > x' or
20:       (x = x' and y > y')

21: Local state:
22:   c ▷ index of adopt-commit-max object, initially 0

23: Procedure propose(v):
24:   while true do
25:     m.write(⟨c, v⟩)   ▷ step R starts
26:     ⟨c', v'⟩ ← m.readmax()   ▷ step R ends
27:     ⟨control, v''⟩ ← C[c'].propose(v')
28:     c ← c' + 1
29:     if control = adopt then v ← v''
30:     else return v''
```

The Archipelago algorithm. Algorithm 4 depicts Archipelago where all processes share an infinite sequence of adopt-commit-max objects (C) to ensure safety and a max register m (lines 17 to 20) to help with convergence. A *max register* r is a wait-free register that provides a write operation, as well as a readmax operation that retrieves back the largest value that was previously written to r [5]. Its write can be implemented by letting each process write to a single-writer multi-reader register whereas its readmax can be implemented by collecting all values written by all processes and taking the maximum. In a synchronous−1 execution, the processes converge towards one value and there is an adopt-commit-max object where all processes propose this exact single value. Then, due to CA-commitment property of the adopt-commit-max object, the adopt-commit-max outputs $\langle \text{commit}, \cdot\rangle$ and Archipelago decides in finite time.

More precisely, Algorithm 4 performs repeatedly three steps (by writing and collecting as defined in Section 2) called R-step, A-step and B-step. In the R-step (lines 25-26), each process p first writes $\langle c, v\rangle$ to register m (line 25) and then retrieves the maximum tuple $\langle c', v'\rangle$ stored in m (line 26). Note that values c and v are not necessarily equal to c' and v'. In the A-step (lines 5-6), process p proposes value v' to adopt-commit-max object $C[c']$ by invoking function $C[c']$.propose(v') (line 27) described in Algorithm 3 and sets c to the next adopt-commit-max object to be used (line 28). A process starts a B-step either at line 7 or 9 of Algorithm 3 and the subsequent collect takes place in line 10. If process p receives a commit response from some adopt-commit-max object (line 30), then process p decides and returns. Otherwise, when process p receives an $\langle \text{adopt}, v''\rangle$ response, it stores this result in the m register (line 29) and restarts.

Difference with eventual leader election, Ω. The cautious reader might think that by solving consensus in an ⋄synchronous−1 execution with Archipelago, we could implement the Ω failure detector [16]. Intuitively, Ω ensures that eventually all correct processes elect the same process as their leader. More precisely, Ω satisfies two properties: (1) *eventual accuracy*: there is a time after which every correct process trusts some correct process, and (2) *eventual agreement*: there is a time after which no two correct processes trust different correct processes.

We could then augment Algorithm 2 with Ω so that Algorithm 2 decides in every $\diamond$synchronous-1 execution. There are ways to implement Ω in crash-recovery settings, but only when a crashed process can recover a finite number of times [24,37,14]. This is in contrast with our model, where a process can be suspended an infinite number of times on an infinite number of rounds. In other words, in our model every process is *unstable* [37], hence the existence of Ω in our model is impossible.

4.1. Archipelago: proof of correctness

Archipelago is a leaderless consensus algorithm. First we show that it satisfies the consensus properties (validity, agreement, and termination under $\diamond$synchrony) and then we prove that it provides leaderless termination, which is more interesting and significantly more challenging. Note that Archipelago solves multi-valued consensus. Naturally, we could have presented and proved correct a modified version of Archipelago for binary consensus. However, we do not believe that such an approach would simplify either the presentation or the proof of Archipelago as we explain later on.

Validity, agreement, termination. Algorithm Archipelago satisfies *validity*. We prove that if an adopt-commit-max object $C[c]$ returns a $\langle \cdot, v \rangle$ tuple, then v was proposed by some process. We can easily show this using induction. For $c = 0$, this is clearly the case, since all the values that were proposed to $C[0]$ are written in m and were initially proposed. Let $c \geq 0$. Assume that for every adopt-commit-max object $C[c']$ with $c' \leq c$, $C[c']$ returns a value that was initially proposed by some process. Then, for a value v to be proposed to $C[c+1]$, this means that a process read $\langle c+1, v \rangle$ from m (line 26). This implies that at some point, some process p writes $\langle c+1, v \rangle$ to m (line 25). But for this to happen, p retrieved $\langle \text{adopt}, v \rangle$ from an adopt-commit-max object $C[c']$ with $c' < c+1$ and by induction, this means that v is a proposed value. Since all the values returned by adopt-commit-max objects are proposed, and Archipelago decides (line 30) upon a value that Archipelago retrieves from some adopt-commit-max object, Archipelago satisfies validity.

Algorithm Archipelago satisfies *agreement*. To see this, assume by way of contradiction that two processes p and p' decide on different values v and v' respectively. This means that process p returned v after receiving a $\langle \text{commit}, v \rangle$ response for an adopt-commit-max object $C[c]$ and process p' received a $\langle \text{commit}, v' \rangle$ response for an adopt-commit-max object $C[c']$. Because the adopt-commit-max object satisfies CA-agreement, it has to be the case that $c \neq c'$, otherwise $v = v'$. Without loss of generality, assume that $c < c'$. All the processes (including p') that received a response from $C[c]$ either received $\langle \text{commit}, v \rangle$ or $\langle \text{adopt}, v \rangle$ due to the agreement property of the adopt-commit-max object. Hence, all processes that write to m (line 25), write $\langle c+1, v \rangle$, since they retrieved v from $C[c]$. Therefore, all possible values that are proposed to the $C[c+1]$ adopt-commit-max object, propose v, and hence $C[c+1]$ returns $\langle \text{commit}, v \rangle$. Similarly, all upcoming adopt-commit-max-objects return $\langle \text{commit}, v \rangle$ contradicting the fact that $C[c']$ $(c < c')$ responds with $\langle \text{commit}, v' \rangle$ with $v' \neq v$.

4.2. Archipelago: proof of leaderless termination

It is far from obvious that Archipelago satisfies leaderless termination. As a matter of fact, Archipelago does not provide leaderless termination for $n = 2$ processes. However, Archipelago satisfies leaderless termination for $n \geq 3$ processes. Before we describe the proof, we introduce some auxiliary notation.

Notation. For an execution α we say that a process p takes a step $A_i(v)$ when p performs an A step that belongs to adopt-commit-max object $C[i]$ (lines 5 and 6). We denote with $A_i^0(v)$ the fact that p is the first process that performed the A step for adopt-commit-max object $C[i]$ in execution α. Note that a single round might contain multiple $A_i^0(v)$ steps taken by different processes. We denote with $A_i^+(v)$ the fact that this step is not the first A step on $C[i]$. We denote with $B_i(\mathbf{1}, v)$ the B step of a process on adopt-commit-max object $C[i]$ that writes $\langle \text{commit}, v \rangle$ (lines 7 and 10). With $B_i(\mathbf{0}, v)$, we denote the B step of a process on adopt-commit-max object $C[i]$ that writes $\langle \text{adopt}, v \rangle$ (lines 9 and 10). Similarly to the notation of an A step, we use the notation $B_i^0(\mathbf{1}, v)$, and $B_i^+(\mathbf{1}, v)$. We say that in an execution α values $v_1, v_2, \ldots, v_k$ are proposed to $C[i]$ if there are steps $A_i(v_j)$ $\forall 1 \leq j \leq k$ in α. We denote with $R\langle c, v \rangle$ the R step of a process and the fact that the process read $\langle c, v \rangle$ as the maximum value in m (lines 25 and 26). As with steps A and B, we use the $R^0\langle c, v \rangle$ and $R^+\langle c, v \rangle$ notation. Specifically, with $R^0\langle i, \cdot \rangle$ we denote the first R step that reads $\langle i, \cdot \rangle$. Note that in this notation when we have $A_i(v)$ and $B_i(\cdot, v)$, this v is the value that is written, while in $R\langle c, v \rangle$ the value v is read from m. Furthermore, note that R is not part of an adopt-commit-max operation like the A and B steps and hence has no subscript.

$n = 2$ processes. For $n = 2$ processes, we can devise a synchronous -1 execution in which the Archipelago algorithm never decides. This execution is depicted in Fig. 2. Fig. 2 has a pattern that repeats every 5 rounds (light-green boxes). In Fig. 2, processes p_1 and p_2 propose values v' and v respectively with $v' > v$. In the first round, process p_1 is suspended, so process p_2 performs an R step, writes $\langle 0, v \rangle$, and retrieves $\langle 0, v \rangle$ from m. Then, in the second round both processes p_1 and p_2 take steps. Process p_1 writes $\langle 0, v' \rangle$ and retrieves $\langle 0, v' \rangle$ since $\langle 0, v' \rangle > \langle 0, v \rangle$. In the same round, p_2 writes v to $C[0].A[2]$. Then, in the third round, when process p_1 takes an A step it writes value v' in $C[0].A[1]$ and when p_1 collects the values written in array A (line 6), p_1 sees that there are two different values (v and v') in $C[0].A$. Therefore, in the fourth round, when process p_1 performs a B step, it retrieves back $\langle \text{adopt}, v' \rangle$. Process p_2 takes a B step in the fifth round after being suspended in the third and fourth rounds, p_2 writes $\langle \text{commit}, v \rangle$ in $C[0].B[2]$, and then during the collect of B, p_2 sees that $\langle \text{adopt}, v' \rangle$ is written in $C[0].B[1]$ and p_2 returns $\langle \text{commit}, v \rangle$ (line 13). Afterwards, starting from the sixth round the processes behave in the exact same way: processes p_1 and p_2 propose v' and v to the next adopt-commit-max object respectively. This can happen ad infinitum and Archipelago never decides.

$n \geq 3$ processes. We consider synchronous-1 executions that start from an arbitrary, albeit valid (i.e., state corresponds to a configuration in a well-formed execution), initial state. We prove that in every synchronous-1 execution, irrespectively of the initial state, Archipelago terminates in finite time. Therefore, in every $\diamond$synchronous-1 execution, eventually the execution becomes synchronous-1 and hence Archipelago decides in finite time.

Theorem 4.2. *Archipelago satisfies leaderless termination for $n \geq 3$.*

To prove Theorem 4.2 we first need to prove some auxiliary lemmas.

Lemma 4.3. *If an execution α contains step $R^0\langle i, v \rangle$, then for any step $R\langle j, v' \rangle$ with $j > i$ that is in α, it is the case that $v' \geq v$.*

Proof. Consider an execution α that contains a step $R^0\langle i, v \rangle$ in a round r taken by process p. Then, when process p continues, p proposes value v to adopt-commit-max object $C[i]$. Similarly and since each process retrieves the maximum value when reading array R (line 26), any later process that performs an R step in round r or after r reads at least $\langle i, v \rangle$, and hence retrieves a value at least as great as v. Note that a process that performs an R step in round

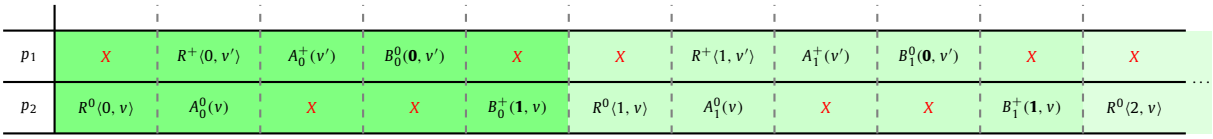


Fig. 2. With 2 processes, Archipelago might never decide in a synchronous−1 execution ($v' > v$).

r cannot read $\langle j, v'\rangle$ with $j > i$ and $v' < v$, since process p takes step $R^0\langle i, v\rangle$. Hence, all values that are proposed to adopt-commit-max object $C[j]$ ($j \geq i$) are $\geq v$ and therefore for any step $R\langle j, v\rangle$ with $j > i$, it holds that $v' \geq v$. □

Lemma 4.4. *If an execution α contains step $B_i^0(\mathbf{1}, v)$, then Archipelago decides v in α.*

Proof. Assume an execution α contains step $B_i^0(\mathbf{1}, v)$ in round r. If a process p takes a step $B_i(\cdot, \cdot)$, then p definitely takes the step in a round k with $k \geq r$. Therefore, process p sees $\langle \text{commit}, v\rangle$ when collecting B (line 10) and either returns $\langle \text{commit}, v\rangle$ (line 12 and then line 30) and decides, or returns $\langle \text{adopt}, v\rangle$ (line 13). Due to CA-agreement, p cannot return $\langle \text{commit}, v'\rangle$ $\langle \text{adopt}, v'\rangle$ with $v' \neq v$. Thus, process p proposes v in adopt-commit-max object $C[i+1]$. However, when all processes propose the same value v to adopt-commit-max object $C[i+1]$, then Archipelago decides v. □

Lemma 4.5. *If an execution α contains at least two steps $A_i^0(v)$ from processes p and p' ($p \neq p'$), and there is no process performing step $A_i^0(v')$ with $v' \neq v$ in α, then either p, or p', or both perform step $B_i^0(\mathbf{1}, v)$ in α.*

Proof. Suppose that a round r contains two $A_i^0(v)$ events by processes p and p' respectively. Since in a round, there can be at most one suspended process, this means that at least one of the processes p and p' take a step in round $r+1$. Since both processes p and p' write value v in array $C[i].A$, and no process wrote another value in $C[i].A$ during that round, v is the only value that p and p' read when collecting A, and hence in the upcoming step in round $r+1$, at least one of the two processes writes $B_i^0(\mathbf{1}, v)$. □

Roughly speaking, the following lemma states that if an execution contains a step $A_i^0(v')$ where $v' > min(\{v : \exists A_i(v) \in \alpha\})$, then any value proposed to a later adopt-commit-max object (i.e., written in A) is greater than $min(\{v : \exists A_i(v) \in \alpha\})$, namely is greater than the minimum value proposed in adopt-commit-max object $C[i]$.

Lemma 4.6. *In an execution α, consider $\mathcal{V}_f = \{v : \exists A_i(v) \in \alpha\}$ and let v_m be $min(\mathcal{V}_f)$. If there is a step $A_i^0(v) \in \alpha$ with $v > v_m$, then for any step $A_j(v') \in \alpha$ with $j > i$, it is the case that $v' > v_m$.*

Proof. Because execution α contains step $A_i^0(v)$ with $v > min(\mathcal{V}_f)$, any step A_j with $j > i$ on adopt-commit-max object $C[j]$ sees value v written in array A (line 9) and hence adopts a value v' with $v' \geq v > v_m$. □

To prove Theorem 4.2 we show that as Archipelago traverses adopt-commit-max objects, the current minimal value, among those values still being proposed to adopt-commit-max objects, eventually gets eliminated (i.e., processes only propose larger values in later adopt-commit-max objects). Specifically, we show that in at most three consecutive adopt-commit-max objects, the minimal value gets eliminated. Since we have n processes, we can have at most n distinct proposed values. Therefore, using at most $3n$ adopt-commit-max objects, Archipelago decides in finite time. From the moment of synchrony, Archipelago needs $\mathcal{O}(n)$ rounds to decide.

Towards this goal, the following lemma is useful. Lemma 4.7 captures the idea that if in an execution α, the minimum value proposed to an adopt-commit-max object $C[i]$ appears in a later adopt-commit-max object $C[j]$ with $j > i$, then α contains a specific execution pattern. By execution pattern we mean, that some process has to take a step, then be suspended, then another process has to take some step, etc.

Fig. 3 captures the fact that there is some process p_a that takes an $A_i^0(v_m)$ step and before p_a performs $B_i(\mathbf{1}, v_m)$ some other process p_b performs $A_i^+(v)$ and $B_i^0(\mathbf{0}, v)$, etc.

Lemma 4.7. *In an execution α, consider $\mathcal{V}_f = \{v : \exists A_i(v) \in \alpha\}$ and let v_m be $min(\mathcal{V}_f)$. If Archipelago does not decide in α and there is a step $A_j(v_m) \in \alpha$ with $j > i$, then $\exists x \geq 2$ and $\exists p_a, p_b \in \mathcal{P}$ and round r such that p_a, p_b perform steps as depicted in Fig. 3 and there is no $R\langle i+1, \cdot\rangle$ step taken before round $r + x + 2$.*

Proof. Suppose that α has no step $A_i^0(v_m)$ and hence α contains a step $A_i^0(v)$ with $v > v_m$. Then, due to Lemma 4.6, we know that for every $A_j(v')$ with $j > i$ it is the case that $v' > v_m$. But this implies that there is no $A_j(v_m)$ with $j > i$ in α and this is not the case we consider. Therefore, for an $A_j(v_m)$ to exist in α, execution α must contain $A_i^0(v_m)$.

Assume that process p_a takes step $A_i^0(v_m)$ in some round r. Lemmas 4.4 and 4.5 imply that if there is another $A_i^0(v_m)$ step in α taken by some process $p \neq p_a$, then the algorithm decides. Since in the lemma we assume that Archipelago does not decide, we can exclude this case and consider that there is at most one $A_i^0(v_m)$ in round r.

Suppose that process p_a takes a step in round $r+1$. Then, process p_a takes a $B_i^0(\mathbf{1}, v_m)$ step since p_a was the process that first performed an A step on adopt-commit-max object $C[i]$. However, if process p_a takes a $B_i^0(\mathbf{1}, v_m)$, due to Lemma 4.4, the algorithm decides. Again, we do not consider this case. Similarly, if process p_a takes a B step in round $r+2$, then process p_a takes a $B_i^0(\mathbf{1}, v_m)$ step and due to Lemma 4.4, the algorithm decides. Therefore, we need to consider the case where process p_a is suspended in both rounds $r+1$ and $r+2$. Process p_a can potentially be suspended for more rounds, up to round $r+x$ where $x \geq 2$. Therefore, for v_m to appear in a later adopt-commit-max object $C[j]$ with $j > i$ with an $A_j(v_m)$ step, execution α has to be similar to the execution depicted in Fig. 4.

We now show that there cannot be an $R\langle i+1, \cdot\rangle$ step before round $r+x+2$. Assume by way of contradiction that there exists an $R\langle i+1, \cdot\rangle$ step before round $r+x+2$ in α. If multiple such steps exist in α, consider the one that takes place in the earliest round. Suppose that this $R^0\langle i+1, v\rangle$ has $v > v_m$. This means that a later process reads value $v > v_m$ and hence when later processes perform an R in some later round, they see a value (line 26) greater than v_m and hence propose only values greater than v_m to upcoming adopt-commit-max objects (Lemma 4.3). This contradicts the fact that there is a $j > i$ with $A_j(v_m)$.

This means that if an $R^0\langle i+1, v'\rangle$ step appears before round $r+x+2$ in α, then it has to be that $v' = v_m$. Suppose that this $R^0\langle i+1, v_m\rangle$ is taken by some process p in round $r+y$. Before round $r+y$ process p has to take steps A_i and B_i since p performs the

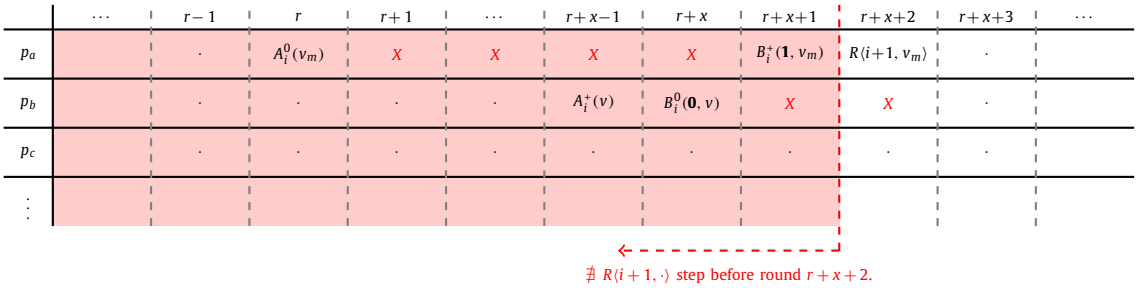


Fig. 3. Execution pattern that appears when the minimum value propagates to the next adopt-commit-max object ($x \geq 2$).

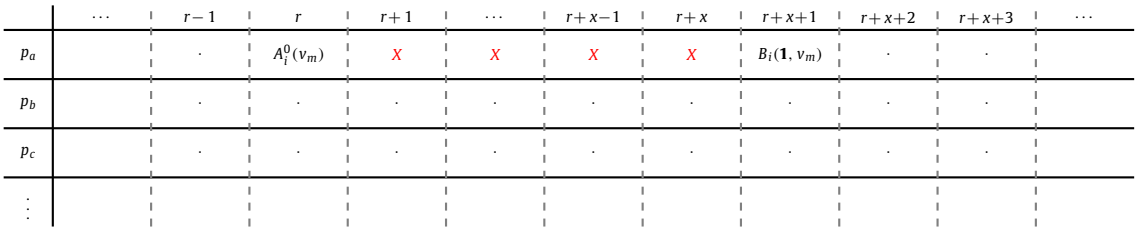


Fig. 4. Long suspension of process p_a with value v_m.

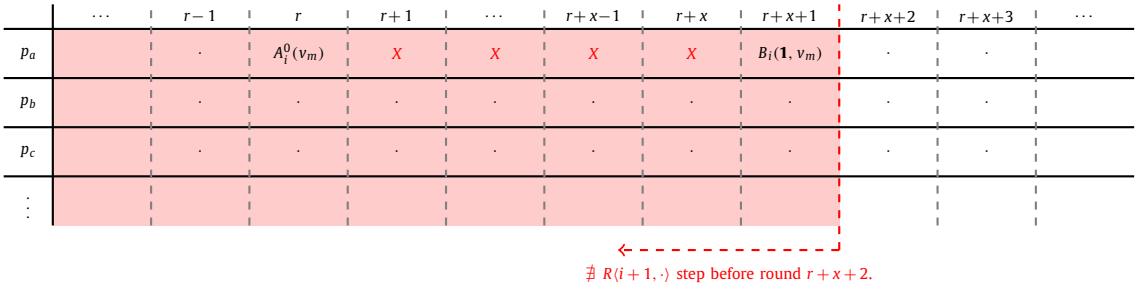


Fig. 5. Impossibility of an $R\langle i+1, \cdot\rangle$ step before round $r+x+2$.

first $R^0\langle i+1, v_m\rangle$ step. This means that value y has to be greater than 2, since otherwise it implies that step A_i taken by p occurs in a round smaller or equal than r. However, process p_a is the only process that takes an $A_i^0(v_m)$ in round r.

Since $R^0\langle i+1, v_m\rangle$ occurs in round $r+y$, where $2 < y < x+2$, then p must perform an $A_i(v)$ step in round $r+y-2$ and a $B_i^0(\cdot,\cdot)$ step in round $r+y-1$ (p cannot be suspended between $r+y-2$ and $r+y$ because p_a is already suspended). If $v = v_m$, then p's B_i step will be $B_i^0(\mathbf{1}, v_m)$ and so, due to Lemma 4.4, the algorithm decides (line 12 and line 30), which we assume does not happen in α. If $v > v_m$, then p's B_i step will be $B_i^0(\mathbf{0}, v)$, which contradicts the fact that p does $R^0\langle i+1, v_m\rangle$ immediately afterwards.

Therefore, there cannot be an $R\langle i+1, \cdot\rangle$ step before round $r+x+2$. This is depicted in the Fig. 5 where all rounds less than $r+x+2$ highlighted in light-red cannot contain an $R\langle i+1, \cdot\rangle$ step.

If between rounds r and $r+x+1$ no other process performs a $B_i^0(\cdot,\cdot)$ step, then process p_a is the first to take a B-Step in adopt-commit-max object $C[i]$ and thus its B-Step is $B_i^0(\mathbf{1}, v_m)$. Hence Archipelago decides due to Lemma 4.4, which contradicts our initial assumption. Therefore, there is at least one process p_b that performs $B_i^0(\cdot,\cdot)$ between rounds $r+1$ and $r+x+1$. If process p_b takes step $B_i^0(\cdot,\cdot)$ in a round smaller than $r+x$, then it performs $R\langle i+1, \cdot\rangle$ before round $r+x+2$ since process p_b has to take continuous steps because p_a is suspended from round $r+1$ to round $r+x+1$, a contradiction. Therefore, process p_b performs a step $A_i(v)$ with $v > v_m$ in round $r+x-1$ and $B_i^0(\mathbf{0}, v)$ in round $r+x$. The current execution is depicted in Fig. 6.

Due to Lemma 4.3, process p_b must be suspended in round $r+x+1$, as well as in round $r+x+2$. Since otherwise, if process p_b is not suspended in rounds $r+x+1$ and $r+x+2$, this implies that process p_b takes an $R^0\langle i+1, v\rangle$ step, where $v > v_m$. Due to Lemma 4.3, this implies that no process proposes v_m to all upcoming adopt-commit-max objects, because all $R\langle i+1, \cdot\rangle$ appear after round $r+x+1$, which contradicts the if-statement of our lemma. Since process p_b is suspended in round $r+x+2$ and at most one process can be suspended in each round, process p_a takes an $R^0\langle i+1, v_m\rangle$ step in round $r+x+2$.

We are therefore in the setting of Fig. 7 that is exactly the same execution pattern as the one in Fig. 3.

To conclude, given an adopt-commit-max object $C[i]$ where the minimum value proposed is v_m, for value v_m to be proposed in the next adopt-commit-max object $C[i+1]$, it has to be that the execution is as shown in Fig. 3. In other words, there is some process p_a that takes an $A_i^0(v_m)$ step alone and, before p_a performs $B_i(\mathbf{1}, v_m)$, some other process p_b performs $A_i^+(v)$ and $B_i^0(\mathbf{0}, v)$, etc. □

Lemma 4.8. *In an execution α, consider $\mathcal{V}_f = \{v : \exists A_i(v) \in \alpha\}$, then for any $A_j(v)$ step with $j \geq i+3$ in α, it is the case that $v > min(\mathcal{V}_f)$ or the algorithm decides.*

Proof. The proof is by contradiction and the idea is to apply Lemma 4.7 on three consecutive adopt-commit-max objects ($C[i]$, $C[i+1]$, and $C[i+2]$) and show that either the algorithm decides or that v_m ($= min(\mathcal{V}_f)$) does not propagate beyond these three adopt-commit-max objects. Due to Lemma 4.7 we know that all processes, except p_a, p_b execute continuously for at least four rounds. We also know that operating on an adopt-commit-max object in Archipelago has only three round-steps (R, A, and B). Be-

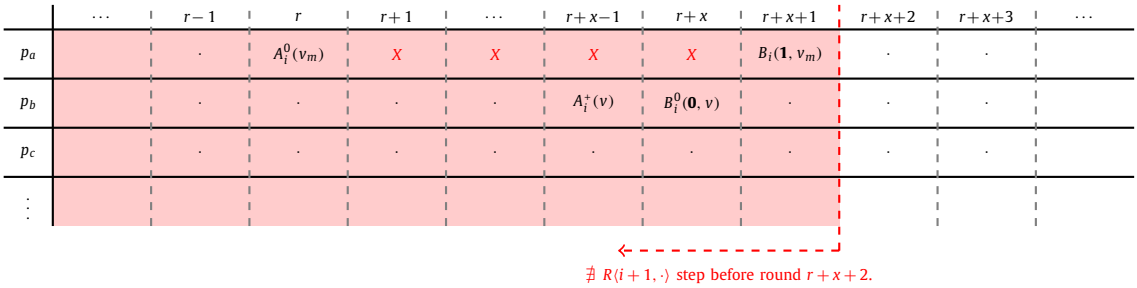


Fig. 6. Process p_b performs a step $A_i(v)$ with $v > v_m$ in round $r+x-1$ and $B_i^0(\mathbf{0}, v)$ in round $r+x$.

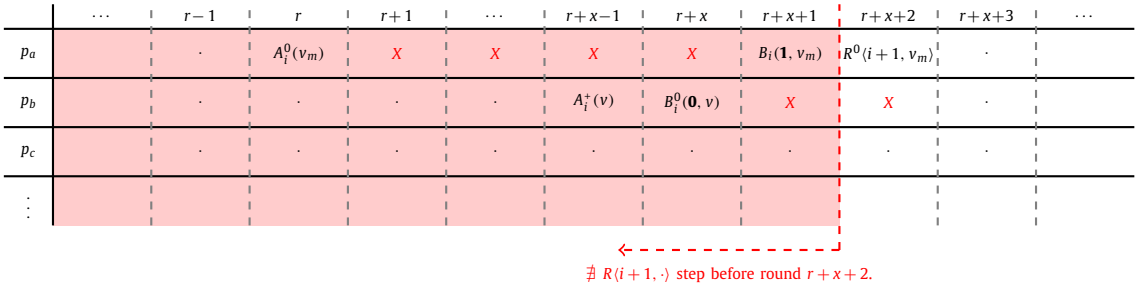


Fig. 7. Execution pattern that appears when the minimum value propagates to the next adopt-commit-max object ($x \geq 2$).

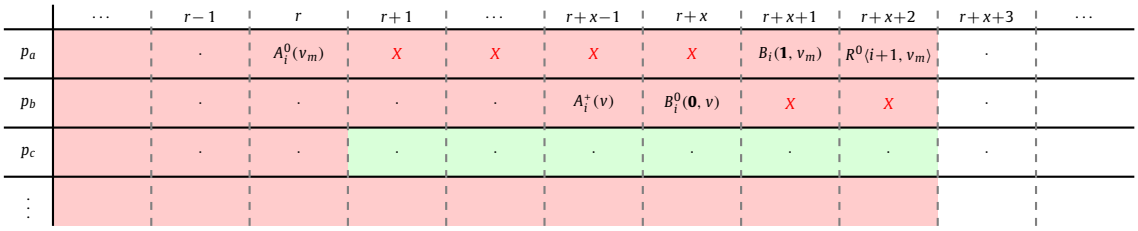


Fig. 8. Lemma 4.8 (1).

cause of this, after three adopt-commit-max objects, we can show that for adopt-commit-max-object $C[i+2]$, there are r'' and x'' such that a process takes a step $R\langle i+3, \cdot\rangle$ before some $r''+x''+2$, which contradicts Lemma 4.7.

To prove this lemma, assume by way of contradiction that there is an execution α such that (1) the algorithm does not decide in α, (2) α contains an $A_i(v_m)$ step and (3) α contains an $A_j(v_m)$ step, where $j \geq i+3$.

Due to Lemma 4.7, we know that if there is a $j \geq i+3$ with $A_j(v_m)$, then the execution looks like Fig. 8. Because $x \geq 2$, we have at least 4 continuous suspensions from round $r+1$ to round $r+x+2$.

Note that, in any execution, a process takes a sequence of steps: $R\langle i_1, \cdot\rangle, A_{i_1}, B_{i_1}, R\langle i_2, \cdot\rangle, A_{i_2}, B_{i_2}, \ldots$ where $i_1 < i_2 < \ldots$. We show that all processes must perform certain steps in this sequence prior to certain rounds. One of the three steps that p_c's takes in rounds $r+1$, $r+2$ or $r+3$ is an R step that returns a value that is at least $\langle i, \cdot\rangle$, since process p_a performed an A_i^0 step in round r. Thus, by round $r+x+2$, p_c must perform an A_j step with $j \geq i$. Processes p_a and p_b have also performed a step A_i by round $r+x+2$. So, every process in the system has performed an A_j step with $j \geq i$ by round $r+x+2$.

By assumption, value v_m does not get eliminated, and hence when the algorithm operates on adopt-commit-max object $C[i+1]$ we have the exact same execution as in Fig. 3 but for adopt-commit-max object $C[i+1]$. See Fig. 9. Again, let $p_{a'}$ and $p_{b'}$ be the processes described in Lemma 4.7 with respect to A_{i+1} and let $p_{c'}$ be any other process. Note that in process $p_{a'}$ is not necessarily the same as process p_a, etc., since it could be that a different process is the one that performs the $A_{i+1}^0(v_m)$ now. For example, it could be that $p_{a'} = p_c$ and $p_{b'} = p_a$. Also, note that round numbers are now based upon $r' \neq r$. By Lemma 4.7, no $R\langle i+1, \cdot\rangle$ occurs before round $r+x+2$ and since $p_{a'}$ does an $R\langle i+1, \cdot\rangle$ step before round r', we have $r' > r+x+2$. Thus, $p_{c'}$ must perform a step A_j with $j \geq i$ before round r'. Then, $p_{c'}$ takes at least four more steps by round $r'+x'+2$. So, $p_{c'}$ must perform a step B_k with $k \geq i+1$ by round $r'+x'+2$. Processes $p_{a'}$ and $p_{b'}$ have performed step B_{i+1} by round $r'+x'+2$. So, every process performs a step B_k with $k \geq i+1$ by round $r'+x'+2$.

Again, because of Lemma 4.7, this pattern of execution should appear for adopt-commit-max object $C[i+2]$. Consider Fig. 10. Again, let $p_{a''}$ and $p_{b''}$ be the processes described in Lemma 4.7 with respect to A_{i+2} and let $p_{c''}$ be any other process. By Lemma 4.7, no $R\langle i+2, \cdot\rangle$ step occurs before round $r'+x'+2$ and since process $p_{a''}$ does such a step before round r'', we have $r'' > r'+x'+2$. Thus, $p_{c''}$ must perform a step B_k with $k \geq i+1$ before round r''. Then, $p_{c''}$ takes at least four more steps by round $r''+x''+2$. Hence, by round $r''+x''+2$, $p_{c''}$ must perform a step $R\langle \ell, \cdot\rangle$ with $\ell \geq i+3$. This contradicts the fact that no $R\langle i+3, \cdot\rangle$ step occurs before step $r''+x''+2$ dictated by Lemma 4.7. □

Lemma 4.8 implies Theorem 4.2, because either the algorithm decides or the minimum value proposed to an adopt-commit-max object $C[i]$ does not propagate in any later adopt-commit-max object $C[j]$ with $j \geq i+3$. Hence, due to the continual elimination of the current minimal value, eventually only one value gets proposed to an adopt-commit-max object and hence the algorithm decides. Finally, note that if we had devised Archipelago for binary consen-

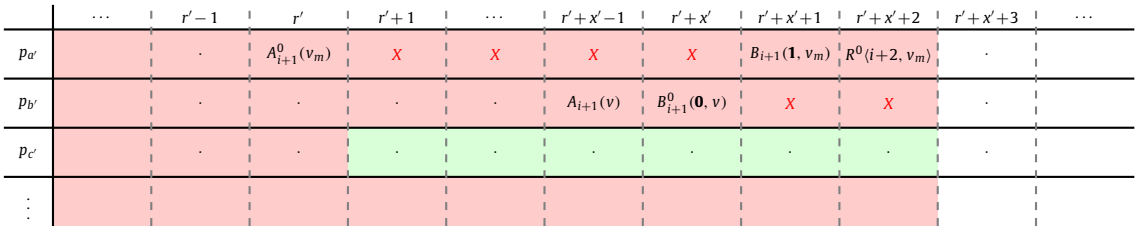


Fig. 9. Lemma 4.8 (2).

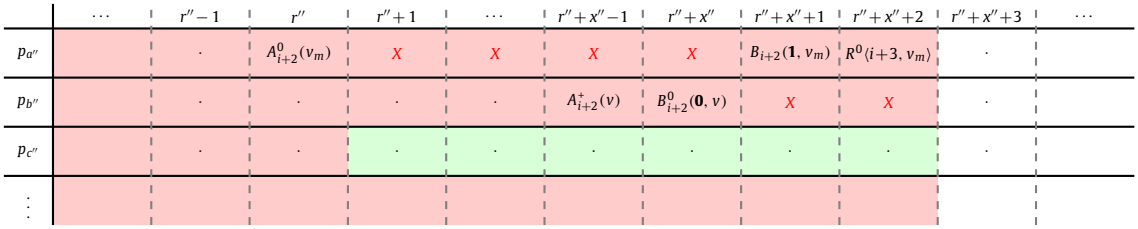


Fig. 10. Lemma 4.8 (3).

sus, this would not substantially simplify the proof. We would still need to prove that the minimum value, in this case 0, does not propagate in later adopt-commit objects.

Archipelago in the common case. In this section we show that Archipelago terminates in any ⋄synchronous execution with up to $f = n-1$ faulty processors. Consider such an execution and let r be a round such that (1) the system has reached synchrony by round r and (2) each process p is either correct or p has stopped omitting by round r. In such an ⋄synchronous execution, Archipelago needs at most 5 rounds starting from round r in order to decide.

As in the proof of leaderless termination for Archipelago, we assume a model with $n \geq 3$ processes. In this scenario, since processes take steps without omissions starting from round r, every correct process p takes steps R, A, and B without suspensions somewhere between round r and $r+5$. Each process p performs an R step at least by round $r+2$, because p can perform step A in round r and then B in round $r+1$. Consider a process p that performs an $R^0\langle i, v\rangle$ step with the greatest $\langle i, v\rangle$ value. This means, that p immediately afterwards performs $A^0_i(v)$ and then $B^0_i(\mathbf{1}, v)$ and due to Lemma 4.4 Archipelago decides. If multiple such processes perform $R^0\langle i, v\rangle$, then all the processes retrieve the same maximum value $\langle i, v\rangle$ from m (line 26) and hence propose the same value to adopt-commit-max object $C[i]$ and perform steps $A^0_i(v)$ and $B^0_i(\mathbf{1}, v)$ and hence the algorithm decides (see Lemma 4.4).

The above discussion implies that Archipelago satisfies termination, thus meaning that in an ⋄synchronous execution, Archipelago decides. Furthermore, note that the Archipelago can withstand up to $f = n-1$ faulty processors and decides in an ⋄synchronous execution. Naturally, the message passing variant of Archipelago (Section 5) can only withstand up to $f = (n-1)/2$ faulty processors.

5. Leaderless consensus in message passing

We now adapt Archipelago for the message passing model where f processes among $n = 2f+1$ can fail: $f-1$ processes can fail by crashing (fail-stop) or fail to send or receive messages when they should (omission faults) and at most 1 additional process can be suspended per round. For the sake of clarity, we refer to this new algorithm as *omission fault tolerant Archipelago or OFT-Archipelago* for short. We consider a message passing model with a point-to-point reliable channel between any pair of processes.

⋄synchronous$-k$ in message passing. To preserve the definition of ⋄synchronous$-k$ in message passing, we first need to define the notion of round and suspension in message passing: In each round r, every (correct, non-suspended) process p_i (i) broadcasts a message (called a *request*), (ii) delivers all requests that were sent to p_i in r, (iii) sends a message (called a *response*) for every request it has delivered in (ii), and (iv) delivers all replies sent to it in r. Note that this notion of round involves 2 message delays, so it corresponds to two rounds in the "traditional" sense [23]. We say that a process p is *suspended* [4] in a round r, if p does not send any messages in r and does not receive any messages sent by other processes in round r.

Adapting Archipelago to message passing. One might be tempted to apply the Algorithm 4 to the ABD emulation [6], which implements a shared-memory abstraction from a set of processes that communicate by message-passing. However, this would require at least two message-passing rounds for each of the R-step, A-step and B-step (one round for the write and one round for the parallel n reads of the collect) and it is unclear whether it would remain leaderless since Archipelago's proof hinges on each step taking exactly one round. This is why, Algorithm 5 combines the write and collect in a single round: the broadcasts in lines 14, 20 and 26 act as both the write and read invocations whereas the responses in lines 36, 39 and 42 confirm the write, and return all values written so far.

We defer the proof of correctness of OFT-Archipelago to Appendix A.

6. Byzantine leaderless consensus

We finally present BFT-Archipelago, the Byzantine fault tolerant (BFT) variant of Archipelago. As BFT consensus cannot be solved without synchrony with $n \leq 3f$ [34], we assume the ⋄synchronous-1 model where f processes among $n = 3f+1$ can fail: at most one is suspended and $f-1$ can behave arbitrarily or be Byzantine. For simplicity of presentation, we also assume authentication and we produce the proof that the result generalizes to the ⋄synchronous$-k$ model, where $k \leq f$ and $f-k$ processes can be Byzantine.

The R-, A-, and B-Steps. BFT-Archipelago is depicted in Algorithm 6 and follows the same 3-step pattern as Archipelago, with the R-, A- and B-Steps executed in consecutive loop iterations, called *ranks*.

Algorithm 5 OFT-Archipelago: Archipelago in message passing.

```
1: Local state:
2:   i, the current adopt-commit-max object, initially 0
3:   R, a set of tuples, initially empty
4:   A[0, 1, ...], a sequence of sets, all initially empty
5:   B[0, 1, ...], a sequence of sets, all initially empty

6: Procedure propose(v):
7:   while true do
8:     ⟨i, v'⟩ ← R-Step(v)
9:     ⟨flag, v''⟩ ← A-Step(v')
10:    ⟨control, val⟩ ← B-Step(flag, v'')
11:    if control = commit then return val
12:    else i ← i + 1

13: Procedure R-Step(v):
14:   broadcast(R, i, v)
15:   wait until receive (R-response, i, R) from f + 1 proc.
16:   R ← R ∪ { union of all Rs received in previous line}
17:   ⟨i', v'⟩ ← max(R)
18:   return ⟨i', v'⟩

19: Procedure A-Step(v):
20:   broadcast(A, i, v)
21:   wait until receive (A-response, i, A[i]) from f + 1 proc.
22:   S ← union of all A[i]s received
23:   if S contains only one value val then return ⟨true, val⟩
24:   else return ⟨false, max(S)⟩

25: Procedure B-Step(flag, v):
26:   broadcast(B, i, flag, v)
27:   wait until receive (B-response, i, B[i]) from f + 1 proc.
28:   S ← union of all B[i]s received
29:   if S contains only ⟨true, val⟩ for some val then
30:     return ⟨commit, val⟩
31:   else if S contains some entry ⟨true, val⟩ then
32:     return ⟨adopt, val⟩
33:   else return ⟨adopt, max(S)⟩

34: Upon reception of (R, j, v) from p:
35:   Add ⟨j, v⟩ to R
36:   send(R-response, j, R) to p

37: Upon reception of (A, j, v) from p:
38:   Add v to A[j]
39:   send(A-response, j, A[j]) to p

40: Upon reception of (B, j, flag, v) from p:
41:   Add ⟨flag, v⟩ to B[j]
42:   send(B-response, j, B[j]) to p
```

- R-Step: process p gathers the *rank* and *value* of other processes with the aim to settle on a common (*rank*, *value*) at lines 17–24. Processes answer the R-broadcast (if they find it valid as we explain below) by sending their highest (*rank*, *value*).
- A-Step: processes broadcast their values and assess whether other processes have conflicting values with theirs. Lines 33–40 describe how a process answers to an A-broadcast, by sending its highest value and another value if it has received one.
- B-Step: a process may broadcast its value with the label true to force other processes to adopt or commit it (lines 52–58). A process responds to a B-broadcast by checking the validity of the broadcast and then responding with its own B-value (lines 64–71).

Except for the messages containing the value proposed in step 1 of rank 0, each message must be accompanied with a valid partial certificate (or it is ignored) as we explain below.

Certificates. Lines 73–91 describe how to build and check certificates. A *partial certificate for a response* message from p_i to p_j contains the queries that justify this response. Below we distinguish a broadcast (i.e., query) from its response even though the response is itself sent to all. A broadcast from p_i justifies a response from p_j for an R-Step if it contains the highest value encountered that appears in the response from p_j. A broadcast from p_i justifies a response from p_j for an A-Step, if it contains the highest value v and, if possible, any value from the response different from v. For a broadcast from p_i to justify a response from p_j for a B-Step, it must ensure the following: if the response contains only true, then the broadcast should contain true; if the response contains at least one true and false pair, then the broadcast should contain the true pair, and any of the false pairs; if the response contains only false pairs, then the broadcast should contain the pair among them with the highest value.

A *partial certificate for a broadcast* contains the union of the $2f+1$ responses received during the previous step with the partial certificates for these responses. A *complementing certificate* at p_i to a partial certificate for a broadcast (resp. response) comprises $f+1$ (resp. $2f+1$) responses received by p_j to each of the queries comprised in the partial certificate.

6.1. BFT-Archipelago: proof of correctness

Theorem 6.1 *(Validity). With no faulty processes, if some process decides v, then v is the input of some process.*

Proof. If all processes are correct, given that all values have to be proposed by some process at some point, then the decided value was necessarily proposed by a correct process. Indeed, at each rank i, processes can only adopt a value that was proposed at some point. □

Before we can prove Agreement, we need two lemmas to show some Byzantine behaviors are impossible under our certificate system.

Lemma 6.2. *If a correct uninterrupted process B-broadcasts $\langle true, v_1\rangle$ at rank i, then no process, even Byzantine, can R-broadcast a value different from v_1 with a valid certificate at rank $i+1$ or more.*

Proof. Assume the B-broadcast of $\langle true, v_1\rangle$ happened first. When a process R-broadcasts at a rank strictly above i, he must add a certificate of all messages and their signatures. In order to be considered as correct by correct processes, this process must, at the very least, provide the B-answers from $2f+1$ processes that led him to R-broadcasting this value. Since it is impossible to forge a signature from another process, this process will have to show unaltered answers from at least $f+1$ correct processes, which will all show the $\langle true, v_1\rangle$ couple, proving that the process should necessarily either commit or adopt v_1.

Now consider by way of contradiction the case where a B-broadcast of $\langle true, v_1\rangle$ by a correct process was to happen after an R-broadcast of a value v_2 different from v_1 at a rank $i+1$ or higher. That is not possible, because during its A-step i, the correct process would see the other value (which has necessarily been A-broadcast at step i in order to obtain a valid certificate) and return a $\langle false, .\rangle$. □

Lemma 6.3. *Let (i, v) be the tuple that is R-broadcast with the highest rank i and a valid certificate. Then no valid certificate can be constructed by a Byzantine process for any R-response (i', v') with $i' > i$.*

Proof. When sending a R-response, the process has to send with it a certificate for each value that it sends. In particular, this process would need to provide a certificate showing that at least one process (possibly himself) rightfully R-broadcast such a (*rank*, *value*), which is impossible according to Lemma 6.2. □

Algorithm 6 BFT-Archipelago in message passing with $n = 3f + 1$.

```
 1: Local state:
 2:   i, the current rank, initially 0
 3:   R, a set of tuples, initially empty
 4:   A[0, 1, ...] and B[0, 1, ...], two
 5:     sequences of sets, all initially empty
 6:   C a sequence of broadcasts ID with the
 7:     number of answers they have received

 8: Procedure propose(v):
 9:   while true do
10:     ⟨i, v'⟩ ← R-Step(v)
11:     ⟨flag, v''⟩ ← A-Step(i, v')
12:     ⟨contr, val⟩ ← B-Step(flag, i, v'')
13:     if contr = commit then return val
14:     else i ← i + 1, v ← val

15: Procedure R-Step(v):
16:   compile certificate C (empty at rank 0)
17:   broadcast(R, i, v, C)
18:   wait until (receive valid (Rresp, i, R, C)
19:     from 2f + 1 processes)
20:   R ← R ∪ {union of all valid Rs received
21:     in previous line}
22:   ⟨i', v'⟩ ← max(R)
23:   R ← max(R)
24:   return ⟨i', v'⟩

25: Upon delivering (R, j, v, C) from p:
26:   if reliability check(R, j, v, C) then
27:     R ← max(⟨j, v⟩, R)
28:     b ← bcast responsible for R[j]'s value
29:     send(Rresp, j, R, sig, b) to all
30:   else ignore message from p

31: Procedure A-Step(i, v):
32:   compile certificate C
33:   broadcast(A, i, v, C)
34:   wait until receive valid (Aresp, i, A[i])
35:     from 2f + 1 processes
36:   S ← union of all A[i]s received
37:   if (S contains at least 2f+1 A-answers
38:     containing only val) then
39:     return ⟨true, val⟩
40:   else return ⟨false, max(S)⟩

41: Upon delivering (A, j, v, C) from p:
42:   if reliability check(A, j, v, C) then
43:     if v ∉ A[j] and |A[j]| < 2 then
44:       add v to A[j]
45:     else if v > max(A[j]) then
46:       min(A[j]) ← v
47:     b ← bcast responsible for A[j]'s value
48:     send(Aresp, j, A[j], sig, b) to all
49:   else ignore message from p

50: Procedure B-Step(∫, i, v):
51:   compile certificate C
52:   broadcast(B, i, ∫, v, C)
53:   wait until receive valid (Bresp, i, B[i])
54:     from 2f + 1 proc.
55:   S ← array with all B[i]s received
56:   if |{⟨true, val⟩ ∈ S}| ≥ 2f + 1 then
57:     return ⟨commit, val⟩
58:   else if |{⟨true, val⟩ ∈ S}| ≥ 1 then
59:     return ⟨adopt, val⟩
60:   else return ⟨adopt, max(S)⟩

61: Upon delivery (B, j, ∫, v, C) from p:
62:   if reliability check(B, j, v, C) then
63:     m ← max(B[j][0].v, B[j][1].v)
64:     if |B[j]| < 2 then add ⟨∫, v⟩ to B[j]
65:     else if (∫ ∧ ⟨∫, v⟩ ∉ B[j]∨
66:     ¬∫ ∧ v > m) then
67:       B[j][0] ← ⟨∫, v⟩
68:       b ← bcast resp. for B[j]'s ⟨∫, vals⟩
69:       send(Bresp, j, B[j], sig, b)
70:     b ← resp. for B[j]'s ⟨∫, vals⟩
71:     send(Bresp, j, B[j], sig, b) to all
72:   else ignore message from p

73: Reliability check broadcast(X, i, v):
74:   if |{bcast-answers ∈ C}| > f then
75:     return true
76:   check that |C| ≥ 2f + 1 messages
77:   check signatures of those messages
78:   check if |{bcast-answers}| > f
79:   if X = R then
80:     check (i, v) is correct according to
81:     signed B-answers received and step B
82:   else if X = A then
83:     check (i, v) is correct according to
84:     signed R-answers received and step R
85:   else if X = B then
86:     check (i, ∫, v) is correct according to
87:     signed A-answers received and step A
88:   return true if all checks pass,
89:     false otherwise

90: To compile a broadcast certificate, list all 2f + 1 answers to the previous step broadcast received during the previous step.
91: To reliably check response (check if a response is valid), check if, for the broadcast(s) originating its value we have received 2f + 1 responses to that broadcast.
```

Theorem 6.4 *(Agreement). Let p_1 and p_2 be two correct processes. If p_1 and p_2 return $\langle commit, v_1\rangle$ and $\langle commit, v_2\rangle$ then $v_1 = v_2$.*

Proof. Consider that both p_1 and p_2 are correct. Assume by contradiction that $v_1 \neq v_2$.

First, assume they both commit using the same rank i in A and B. Then this means both p_1 and p_2 saw, during their B-step line 56, at least $2f + 1$ $\langle true, v_1\rangle$ and $\langle true, v_2\rangle$ respectively. Since processes can only ever send one B-answer to each process, it means that p_1 and p_2 both received B-answers from at least $f + 1$ correct processes. If we consider f processes to be possibly Byzantine, this leaves only $2f + 1$ correct processes. Hence, there is at least one of these correct processes which will answer to both p_1 and p_2. One of them will be answered second and will see the value proposed by the other, and therefore cannot commit its own value. Hence, it is impossible for two correct processes to commit different values.

For different ranks i and j, assume now without loss of generality that one of those two processes, say p_1, commits v_1 using B_i and p_2 commits v_2 using B_j with $j > i$. Then this means p_1 saw, during its B-step line 56, at least $2f + 1$ sets containing only $\langle true, v_1\rangle$, meaning that no other process had yet B-broadcast another value or that any process B-broadcasting in the same round will have to either adopt or commit v_1 (indeed, another process would see at least one B-answer from a correct process containing $\langle true, v_1\rangle$ and would hence at least adopt, maybe commit v_1).

Now there are two possibilities: either no other process has yet run an R-step at a rank strictly higher than i. Then the max function prevents it from jumping directly ahead of rank i. In this case,

before advancing to rank $i+1$, p_2 has to go through rank i. Notice that no Byzantine process can pretend to have advanced past rank i without actually providing the signed messages that led to it, i.e. actually advancing through steps while acting like a normal process (cf. Lemma 6.3). Thus it is certain that p_2 will see at least one $\langle true, v_1 \rangle$ in his B-answers from rank i. It will thus either commit or adopt it. Therefore, all correct processes who reach rank $i+1$ by incrementing their rank (line 14) will propose value v_1. Other processes who run an R-step after that will be able to jump straight to the highest R-visited rank and will R-return value v_1, because there is no value different from v_1 past rank i. Hence no two correct processes can decide on different values. □

Lemma 6.5. *The R-Step satisfies the following properties:*

- ***Validity*** *For a fixed i, if some correct process returns v, then v was the input of some process.*
- ***Monotonicity*** *If a **correct** process p returns (i, v_i) in an R-Step and p returns (j, v_j) in a later R-Step, then $j \geq i$ and $v_j \geq v_i$.*

Proof.
- **Validity** At line 24 (i', v') (the value returned by the R-Step) is computed as the maximum of all tuples ever received, which must in turn have been broadcast at line 17 by some process (we can be sure that there are at least $f+1$ correct processes that proposed a value because there at most f faulty processes and we wait for a quorum of $2f+1$ answers). Hence all values that appear have been proposed by some process.
- **Monotonicity** Assume by contradiction that some **correct** process p returns (i, v_i) in R-Step r_1 and later returns (j, v_j) in R-Step r_2 such that $(j, v_j) < (i, v_i)$. Because R always keeps the maximum element, it is impossible to later R-return a smaller element, thanks to the max function. □

6.2. BFT-Archipelago: proof of leaderless termination

In this section we prove that BFT-Archipelago satisfies leaderless termination.

The key idea of the proof is that in order to prevent termination, processes have to release some higher value during the A-step to prevent processes from seeing only "true" messages. But this means the value will be seen by $O(n)$ processes and hence the smaller value will be discarded. As it consumes a value to delay the algorithm by $O(1)$ rounds, and there are at most n different values, after $O(n)$ rounds there will be only one value left, which will be committed. Before we prove that BFT-Archipelago (Theorem 6.10) satisfied leaderless termination, we need to prove the following lemmas.

Lemma 6.6 *(Commitment). If no process R-broadcast anything other than the same (i, v), then all correct processes must output $\langle commit, v \rangle$.*

Proof. Since all the ranks and values coming in R-answers are identical, all correct processes will R-return (i, v) and Byzantine processes cannot present a valid A-broadcast with any value other than v.

Hence all correct processes will A-broadcast v. All valid A-answers will contain only v and hence all correct processes will A-return $\langle true, v \rangle$. Therefore, no Byzantine process can present a valid B-broadcast with anything other than $\langle true, v \rangle$.

Hence all correct processes B-broadcast $\langle true, v \rangle$ and can only receive valid B-responses containing only $\langle true, v \rangle$ or invalid B-responses which will be ignored. Therefore, all correct processes will B-return $\langle commit, v \rangle$. □

Lemma 6.7. *All correct processes eventually receive $2f+1$ replies to their R, A or B-broadcasts.*

Proof. Once GST is reached, all messages eventually arrive. The certificate of a correct process p will therefore eventually get accepted by any correct process as (i) all calculations made are correct and (ii) all broadcasts referenced in the certificate can be checked as valid by all correct processes as soon as they receive $f+1$ responses to each of those broadcasts. As p checked before accepting responses in the previous step that all broadcasts referenced in the next certificate had received $2f+1$ responses received by p, amongst which $f+1$ were made by correct processes and hence were sent to all processes (and eventually received). □

Lemma 6.8. *With the hypothesis that processes only get interrupted for whole rounds, it is not possible for a Byzantine process to make correct processes R-return different values after GST and round synchronization.*

Proof. Let us recall that all messages are signed, therefore Byzantine processes cannot make up fake messages that are not coming from themselves.

If a Byzantine process sends its proposed $(rank, value)$ to all correct processes, either the certificate is invalid and it is ignored, either it is valid and all correct processes will see the $(rank, value)$ in at least $f+1$ R-answers and all R-return the same value.

If the Byzantine process decides to R-broadcast only to some correct processes, there are 2 cases. If the Byzantine process R-broadcasts to f or less correct awake processes, then some processes may not see this value at all, and those who see it will see at least $f+1$ R-answers not containing that value, and can therefore deduce it was sent fraudulently and ignore it.

If the Byzantine process R-broadcasts to $f+1$ or more awake processes, then all correct processes will receive at least one R-answer containing that value. Hence if the value is big enough to be the max of the values R-broadcast, it will be R-returned by all correct awake processes. □

Lemma 6.9. *There cannot be a $\langle true, v_1 \rangle$ and a $\langle true, v_2 \rangle$ B-broadcasts with valid certificates and $v_1 \neq v_2$.*

Proof. In order to have a valid certificate, a process would need to show proof of $2f+1$ different processes providing A-answers containing only v_1 (respectively v_2), which amounts to $4f+2$ different answers. Since there are only f Byzantine processes, it means that at least one correct process answered to both and will therefore show at least one A-answer containing (v_1, v_2). Hence, no valid certificate for two different values with the true flag can be produced. □

Theorem 6.10. *In every $\diamond$synchronous-1 execution of BFT-Archipelago, every correct process decides.*

Proof. Assume we have reached GST. We will study what happens during the B-step and the following R-step. Remember that because of Lemma 6.9, no two different values can be B-broadcast with the label "*true*" and a valid certificate. Hence only two cases are available: either all values B-broadcast at rank i are flagged as "*false*", or only one of them is flagged as true.

Assume all processes only B-broadcast values flagged as false. Either all those values are the same, in which case we already have only one value that can be R-broadcast with a valid certificate. Either there are some different values. Let us call v_{min} the smallest of those values. The fact that all values are flagged as false indicates that all correct processes have encountered at least two different values during their previous A-step, and thus have discarded the minimum one(s). As processes can only ever R-broadcast greater or equal values due to the max function at every step, it means

that all correct processes have discarded at least one value during the A-step. As the number of values and processes are finite, there will eventually be only one value left. Assume now all values B-broadcasted are flagged as false but one (if all values are flagged as true, all correct processes commit). Let us call that value v_{true}. The number of processes with flag false at rank i is either $O(n)$, in which case we only need to mention that those processes have each encountered different values at step A (which is why they have a "*false*" flag) and hence have all discarded at least one value. Now let us assume by way of contradiction that there are only $O(1)$ of those processes. We will show that this is impossible. Without loss of generality, we are considering the group of processes which are in the highest rank i. The fact that those $O(1)$ processes delivered some answers to receive the flag "*false*" means that there were $2f+1$ correct uninterrupted processes to deliver those answers. Those processes (which total amounts to $O(n)$) can be either in steps R, A or B. We will now explore what happens if a $O(n)$ of those processes are in those three cases. As there are at least 2 different values delivered by each $2f+1$ different processes, then there are at least $f+1$ processes that delivered both values. Let us consider those processes. Consider the $O(n)$ processes in step R. those processes will take step A afterwards and will therefore see the (at least two) values they have delivered. Hence they will also A-return a false, and hence there were $O(n)$ processes with flag "*false*", which is a contradiction. Consider the $O(n)$ processes in step A. Then those processes have delivered different values in their A-responses, hence they will also A-return a false, and hence there were $O(n)$ processes with flag "*false*", which is a contradiction. Consider the $O(n)$ processes in step B. At the same round where they were uninterrupted and they delivered the A-responses that led to the "*false*", they must have B-broadcast the value with flag "*true*". When uninterrupted, the $2f+1$ processes will process the reliable-B-broadcast of the "*true*" at the same pace as the reliable-B-broadcast of the values in "*false*" but with some overhead. Hence the value with flag "*true*" will be delivered before the ones with "*false*", and all the processes with "*false*" will have to adopt that value and at the next R-step only the value flagged "*true*" can be R-broadcast with valid certificate.

Hence at each suite of 3 steps R, A and B taken by all processes there are $O(n)$ processes which discard at least one value each. As there are only $O(n)$ different values at most, there will be at most $O(n)$ rounds before there is only one value left to be R-broadcast (with a valid certificate).

Due to Lemma 6.6, when that happens all correct processes will commit within 5 rounds. □

7. Discussion and complexity analysis

Termination. In addition to leaderless termination (Theorem 4.2), Archipelago satisfies termination for $n \geq 3$, meaning that in an eventually synchronous [14] execution, every correct process eventually decides. In such an execution, Archipelago needs at most 5 rounds, after the global stabilization time [23] and round synchronization (i.e., all processes start and end a round at the same time).

Fast path of BFT-Archipelago. The common-case performance of BFT-Archipelago can be improved by executing an optimistic fast path under favorable conditions (e.g., synchrony, no failures, no contention), and falling back to a robust path when these conditions are not met. This can be achieved with the Abstract scheme [8] as it allows chaining multiple BFT protocols, called Abstract instances, that can abort and fall back to the next instance. In particular, the *Backup* wrapper allows any full BFT protocol to become an Abstract instance. Since BFT-Archipelago is a full BFT protocol, it is amenable to a Backup instance, and thus can be accelerated with Quorum fast path that can decide in two message delays.

Complexity of BFT-Archipelago. BFT-Archipelago terminates deterministically by exchanging and storing at most $O(n^4)$ messages and bits (each message is of length $O(1)$ bits), and terminates within $O(n)$ rounds and $O(n^4)$ calculations and signature checks. BFT-Archipelago is resilient optimal [23] and time optimal [25,22]. BFT-Archipelago also has the same communication complexity as PBFT [15] and DBFT [19].

In particular, the message length is $O(1)$ bits because the algorithm broadcasts (i) a local register (R, A or B) of size $O(1)$ messages, each containing a certificate, (ii) i and v of fixed length $O(1)$, (iii) a *flag* of length $1 = O(1)$ and a (iv) signature of length $O(1)$. A response contains $O(1)$ messages (one or two to be precise). Each of these messages is certified as having been rightfully broadcast, but only by the $(2f+1)$ answers that the processes have received. Hence the length of an answer is $O(1)$. The full proof of complexity can be found in the proof of Theorem 6.10, which depicts the number and sizes of messages sent. Below, we give a quick additional sketch of the complexity for pedagogical purposes:

For all correct processes to complete the calculations necessary to truthfully go through a whole rank, it takes (at worst) each of the $O(n)$ processes $O(n)$ broadcasts, which leads in total to $O(n^2)$ broadcasts, each of length $O(1)$ bits. For each broadcast there are $O(n^2)$ responses exchanged of length $O(1)$ bits each, for a total amount of $O(n) * (O(n^2) * O(1) + O(n^2) * O(n) * O(1)) = O(n^4)$ bits exchanged in the worst case in order to have all of the processes proceed through a rank. After GST, we need at most $O(n)$ rounds taken by all processes to decide. However, as can be read in the proof of Theorem 6.10, each process will not have to perform the computations executed above more than $O(1)$ time; most of the rounds are only performed by one process that has been delayed either by byzantine processes or simply because it was interrupted, and hence most of the processes, although alive, do not broadcast and only respond. This is why we have a total complexity of $O(n^4)$.

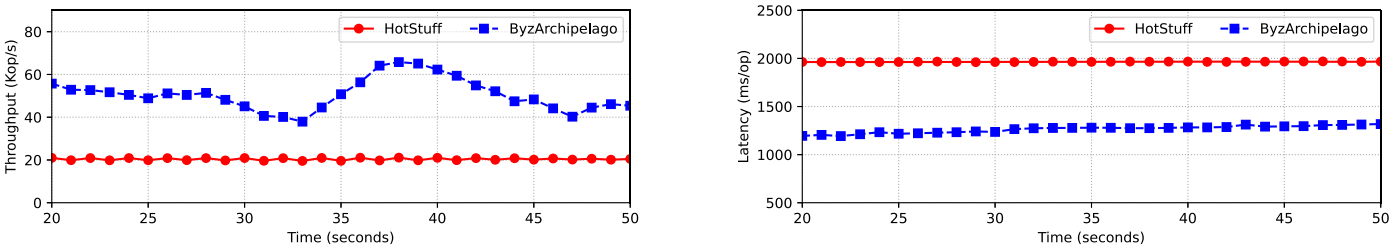
8. Evaluation

In this section, we implement BFT-Archipelago as a state machine replication (SMR) and compare it against the HotStuff BFT SMR [47] also used by Libra, the blockchain designed by Facebook [1], because it is the most communication-efficient SMR we know of. Specifically, our experiments aim at answering two questions: (a) is the performance BFT-Archipelago suitable for a real world setup? and (b) can we confirm empirically that BFT-Archipelago is more robust than a leader-based approach? We answer the first question in Section 8.2 by showing that BFT-Archipelago exhibits better performance than HotStuff in a WAN setup. We answer the second question in Section 8.3 by showing that faults impact negatively the throughput of HotStuff but not BFT-Archipelago's.

8.1. Experimental setup

We implement BFT-Archipelago in Java using the BouncyCastle[2] library for cryptographic operations, using *ecdsa* with the curve `secp256k1` and `sha256` cryptographic primitives and the standard *java.nio* library for network primitives. We optimized our BFT-Archipelago SMR implementation using batching and pipelining. As BFT-Archipelago is leaderless, we benefited from the *distributed* pipelining [45] to have distributed nodes spawning $P = 3$ BFT-Archipelago consensus instances in parallel. By contrast, HotStuff centrally pipelines up to two consensus instances due to its

[2] https://www.bouncycastle.org/.



(a) Throughput (b) Latency

Fig. 11. Throughput and Latency of BFT-Archipelago and HotStuff in Europe.

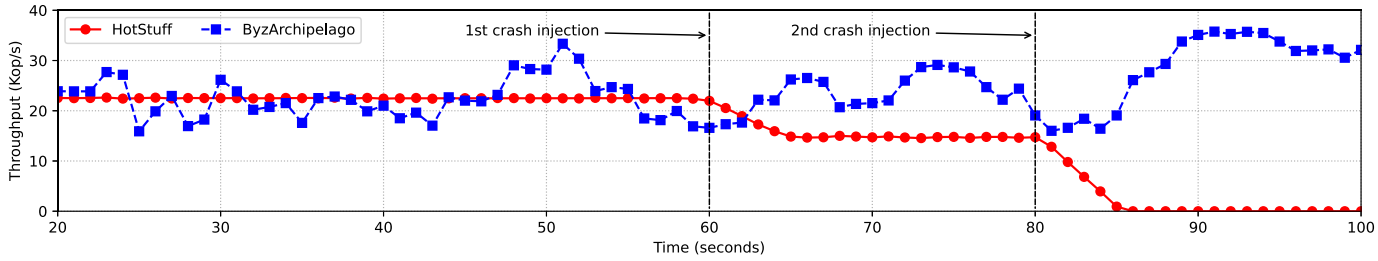


Fig. 12. Throughput of BFT-Archipelago and HotStuff in Europe with two crash faults.

leader-based nature. Instead of proposing a single value, each BFT-Archipelago replica accumulates up to $B = 20000$ transactions in a batch and then proposes this batch as a single value. When a replica transfers the same batch to another replica more than once, it sends the hash of this batch instead of the batch itself to save bandwidth. As HotStuff requires instead dedicated clients to request all servers, it can only exploit hashes to encode each individual transaction.

We deploy BFT-Archipelago and HotStuff on Amazon EC2 and evaluate their performance utilizing 4 or 8 nodes distributed evenly in four datacenters in Frankfurt, London, Ireland and Paris. For every experiment we use *c5.xlarge* instances. Each instance has 4 vCPU and 8 GB of memory. In each datacenter, we also deploy an additional *c5.xlarge* instance to act as a set of clients. The RTT between two datacenters is consistently between 10 ms and 30 ms. The RTT between two machines in the same datacenter is negligible.

We use the official version of HotStuff written in C++. It uses the Salticidae library for network communication and its cryptographic primitives, which are the same as for BFT-Archipelago. HotStuff also uses batching and pipelining as an optimization. Each HotStuff replica proposes a batch of B transaction hashes as a single value in its consensus instances. Additionally, HotStuff replicas always execute P consensus instances in parallel. In all experiments, we use $B = 400$ and $P = 3$ for HotStuff which are the default values and the ones described in the original publication [47].

8.2. Performance in WAN

In a first experiment, we compare the throughput and latency of BFT-Archipelago against HotStuff when deployed on 4 server nodes located in different data centers. The Fig. 11a shows the throughput of BFT-Archipelago and HotStuff during a 50 seconds experiment. The throughput is averaged over time in a 5 seconds sliding window. We observe that the throughput of BFT-Archipelago is between 2× and 3× higher than the throughput of HotStuff. We explain this difference by two factors: the absence of a single point of contention in BFT-Archipelago and the batching optimization of BFT-Archipelago which consumes less bandwidth than in HotStuff. Moreover, we observe that the throughput BFT-Archipelago oscillates with an amplitude of 20 Kops/s. This is because the BFT-Archipelago implementation uses very large batches whose transmission time tends to vary a lot in WAN setups. Additionally, Fig. 11b shows the average transaction latency of BFT-Archipelago and HotStuff. We observe that the latency of BFT-Archipelago is consistently lower than 0.7× the latency of HotStuff. This shows that for a small network, BFT-Archipelago provides a good throughput, even compared to a state of the art BFT SMR. Note that we do not evaluate the algorithms with a large number of nodes, this is because HotStuff is known to not scale: its performance decreases with the system size [47] and similarly the size of certificates in BFT-Archipelago also prevents us from scaling to very large number of nodes.

8.3. Fault tolerance

In a second experiment, we compare the throughput of BFT-Archipelago against HotStuff under crash failures. To this end we deploy both SMR on 8 server nodes, 2 in each datacenters. At $t = 60$ seconds, we simulate the crash of one server with id `0`, in the Frankfurt datacenter, by killing the server process with `SIGKILL`. At $t = 80$ seconds, we also crash the server with id `1`, still in the Frankfurt datacenter. The Fig. 12 shows the evolution of throughput over time for BFT-Archipelago and HotStuff. The throughput is averaged over time in a 5 seconds sliding window. We observe that after the crash of the first server, which is the leader in HotStuff, the throughput of HotStuff falls from 22.5 Kops/s to 14.5 Kops/s while the throughput of BFT-Archipelago oscillates between 15 Kops/s and 33 Kops/s before and after the first crash. We explain the performance drop in HotStuff by the fact it is a leader-based system and the crash of its leader negatively impacts the whole system throughput. In contrast, BFT-Archipelago which is leaderless is left unaffected by the crash of one server. Additionally, we observe that after the crash of the second server, the throughput of HotStuff collapses to 0 and never recovers while the throughput of BFT-Archipelago increases to an interval between 30 Kops/s and 35.5 Kops/s. The complete crash of HotStuff is unexpected since the HotStuff algorithm tolerates $t = 2$ faults when $n = 8$. We conjecture that the increased throughput of BFT-

Archipelago after the second crash is caused by the batch broadcast made faster by the reduced number of nodes.

9. Related work

Given the notorious impact of a leader on consensus performance [36,10,30,7,39,29,28,2,13,19,46,45,9], it is surprising that the leaderless concept has never been precised.

The leader has become a limitation to scale consensus to large blockchain networks. Crain et al. [19] consider the Democratic BFT (DBFT) consensus algorithm as leaderless. DBFT is a multivalue consensus algorithm at the heart of the Red Belly Blockchain [20] whose n proposers bypass the leader bottleneck. It spawns n concurrent binary consensus instances, each relying on a weak coordinator to help converge when many correct processes propose distinct values. Although DBFT could use n different weak coordinators, its binary consensus is not leaderless according to our definition.

Dispel [45] is a pipelined SMR invoking the Democratic BFT consensus algorithm. An empirical comparison of Dispel with HotStuff also confirms our observation: isolated failures affect the performance of HotStuff significantly.

In a brief announcement [35], Lamport proposed a high level transformation of a class of leader-based consensus algorithms into a class of leaderless algorithms using repeatedly a synchronous virtual leader election algorithm where all processes try to agree on a set of proposals. In a corresponding patent document [33], Lamport explains that during a period of asynchrony, if the virtual leader election fails, then the consensus algorithm may not progress [35]. Our adopt-commit-max object of Archipelago allows processes to converge towards a unique value, hence sharing similarities with the proposal of some virtual leader. Yet, neither a leaderless definition nor a virtual leader specification were given by Lamport.

Borran and Schiper proposed a so-called "leader-free" consensus algorithm [10] without presenting however any precise leader-freedom definition. The algorithm has an exponential complexity, which limits its applicability.

Interestingly, SMR algorithms that rely on multiple leaders (e.g., Mencius [36], RBFT [7]) do not necessarily rely on a leaderless consensus algorithm.

Moraru et al. [39] used multiple "command leaders" in EPaxos. Each leader tries to commit one command. When commands have dependencies only one of the leaders can get its command committed at a time, as if there were successive leader-based consensus instances. If a leader fails after receiving a positive acknowledgment from a fast quorum of $n-1$ processes, it rejoins with a new identifier and a greater ballot without being able to acknowledge the previous commit message.

Recently, some errors [44,43] were found in both randomized [40,38] and multi-leader consensus algorithms [39], indicating that getting rid of the leader is error prone.

10. Conclusion

In this paper, we demonstrated the existence of leaderless indulgent consensus algorithms. Our definition of leaderless is general. It relies on the ability to terminate despite a specific kind of fault, *interruption*, which complements the classical crash, omission or Byzantine faults. An interruption can be seen as a form of weak synchrony. Our evaluation of a pipelined state machine replication built on top of the Byzantine fault tolerant consensus algorithm demonstrates the applicability of such algorithms to the permissioned blockchain context.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

Data will be made available on request.

Acknowledgments

This work is supported in part by the Australian Research Council Future Fellowship funding scheme (180100496).

Appendix A. OFT-Archipelago: proof of correctness in message passing

We now prove the correctness of OFT-Archipelago or Archipelago in its message-passing version. All results and line numbers in this sub-section refer to Algorithm 5.

Lemma Appendix A.1. *The R-Step satisfies the following properties:*

- ***Validity*** *For a fixed i, if some process returns v, then v was the input of some process.*
- ***Monotonicity*** *If process p returns (i, v_i) in an R-Step and p returns (j, v_j) in a later R-Step, then $j \geq i$ and $v_j \geq v_i$.*

Proof.
- **Validity** At line 17 (i, v') (the value returned by the R-Step) is computed as the maximum of all tuples ever received, which must in turn have been broadcast at line 14 by some process.
- **Monotonicity** Assume by contradiction that some process p returns (i, v_i) in R-Step r_1 and later returns (j, v_j) in R-Step r_2 such that $(j, v_j) < (i, v_i)$. During r_1, p selected and returned (i, v_i) as the maximum element of its local R set. Since elements can only be appended to a process's R set, (i, v_i) will still be in R during r_2. Thus, p cannot select and return a tuple smaller than (i, v_i) during r_2. We have reached a contradiction. □

Lemma Appendix A.2. *For a fixed i, an A-Step followed by a B-Step corresponds to an adopt-commit object.*

Proof. Validity holds because at lines 23, 24, 30, 32, and 33, processes only return values that were sent at lines 39 or 42. In turn, these values must be input values of some process who broadcast them at lines 20 or 26.

Termination holds because the only waiting is done at lines 21 and 27; processes always wait for $f+1$ responses; since $f+1 = n-f$, processes eventually receive these responses.

Commitment holds because if all processes enter A-Step with the same value v, then the check at line 23 will succeed and all processes will enter B-Step with (true, v); thus the check at line 29 will succeed and all processes will return (commit, v) in the B-Step.

Agreement. Assume by contradiction that process p outputs (commit, v) and process p' outputs $(\cdot, v')$ with $v \neq v'$. Then p must have received B-responses containing only $(true, v)$ from a set R_p of $f+1$ distinct processes; p' must have also received B-responses from a set $R_{p'}$ of $f+1$ distinct processes. Since $f+1 > n/2$, R_p and $R_{p'}$ must intersect in at least one process q.

Let $\mathcal{S}$ be the union of all $B[i]$s received by p' in B-responses. We distinguish three cases, based on the number of distinct values val for which the $\mathcal{S}$ contains $(true, val)$.

- $\mathcal{S}$ does not contain any $(true, val)$ tuples. In this case, q's B-response to p' must contain a $(false, val)$ tuple. If q responded to p before p', then by Lemma Appendix A.3 q's B-response to p' must include a $(true, v)$ tuple — a contradiction. If q responded to p' before p, then by Lemma Appendix A.3 q's B-response to p must include $(false, val)$ — a contradiction.
- $\mathcal{S}$ contains $(true, val)$ tuples for a single value val. Then $val \neq v$, otherwise p' would either commit v or adopt v. Assume wlog the q responds to p before it responds to p'. Then q's response to p' must contain both $(true, v)$ and $(true, val)$, contradicting Lemma Appendix A.4.
- $\mathcal{S}$ contains more than one value v. This is impossible by Lemma Appendix A.4. □

Lemma Appendix A.3. *For a fixed i, if a process p sends a B-response* $(\textit{B-response}, i, B[i])$ *to some process q at time t and p sends a B-response* $(\textit{B-response}, i, B[i]')$ *to some process q′ at time* $t' > t$*, then* $B[i] \subseteq B[i]'$.

Proof. This is because items can only be added to $B[i]$ (line 41). □

Lemma Appendix A.4. *For a fixed i, if two processes p and q broadcast* $(true, v)$ *and* $(true, v')$ *at line 26, then* $v = v'$.

Proof. Assume not, then p must have received A-responses containing only v from a set R_p of $f+1$ processes and q must have received A-responses containing only v' from a set $R_{p'}$ of $f+1$ processes. Since $f+1 > n/2$, R_p and $R_{p'}$ must intersect in at least one process r. Assume without loss of generality r responded to p first and then to q: then the response to q must also include v by Lemma Appendix A.3. We have reached a contradiction. □

Theorem Appendix A.5 *(Validity). With no faulty processes, if some process decides v, then v is the input of some process.*

Proof. The theorem follows by induction from the validity properties of the R-Step (Lemma Appendix A.1) and of the A- and B-Steps (Lemma Appendix A.2). □

Theorem Appendix A.6 *(Agreement). Let* p_1 *and* p_2 *be two correct processes. If* p_1 *and* p_2 *return* $\langle commit, v_1\rangle$ *and* $\langle commit, v_2\rangle$ *then* $v_1 = v_2$.

Proof. Consider that both p_1 and p_2 are correct, the proof is by contradiction. Assume that $v_1 \neq v_2$.

First, assume they both commit using the same rank i in A and B. By Lemma Appendix A.2, an A-Step followed by a B-Step correspond to (enforce the same properties as) an adopt-commit object. Thus, the theorem follows from the agreement property of adopt-commit.

For different ranks i and j, assume now without loss of generality one of those two processes, say p_1, commits v_1 using B_i and p_2 commits v_2 using B_j with $j > i$. Then this means p_1 saw, during its B-step line 29, at least $f+1$ sets containing only $\langle true, v_1\rangle$, meaning that no other process had yet B-broadcast another value or that any process B-broadcasting in the same round will have to either adopt or commit v_1 (indeed, another process would see at least one B-answer from a correct process containing $\langle true, v_1\rangle$ and would hence at least adopt, maybe commit v_1).

Now there are two possibilities: either no other process has yet run an R-step at a rank strictly higher than i. Then the max function prevents it from jumping directly ahead of rank i. In this case, before advancing to rank $i+1$, p_2 has to go through rank i. Thus it is certain that p_2 will see at least 1 $\langle true, v_1\rangle$ in his B-answers from rank i. It will thus either commit it or adopt it. Therefore, all correct processes who reach rank i+1 by incrementing their rank (line 12) will propose value v_1. Other processes who run an R-step after that will be able to jump straight to the highest R-visited rank and will R-return value v_1, because there is no value different from v_1 past rank i. Hence no two correct processes can decide on different values. □

A.1. OFT-Archipelago: leaderless termination in message passing

We now prove that OFT-Archipelago satisfies leaderness termination.

Lemma Appendix A.7 *(Commitment). If no process R-broadcast anything other than the same* (i, v)*, then all correct processes output* $\langle commit, v\rangle$.

Proof. Since all the ranks and values coming in R-answers are identical, all correct processes will R-return (i, v). Hence all correct processes will A-broadcast v. All A-answers will contain only v and thus, all correct processes will A-return $\langle true, v\rangle$. As a result, all correct processes B-broadcast $\langle true, v\rangle$ and can only receive valid B-responses containing only $\langle true, v\rangle$. Therefore, all correct processes B-return $\langle commit, v\rangle$. □

Lemma Appendix A.8 *(Iterative elimination of values). Eventually only one value can be R-broadcast or all correct processes commit.*

Proof. Assume we have reached GST. We study what happens during the B-step and the following R-step. Remember that no two different values can be B-broadcast at the same rank with the label *true* (that would mean that two different processes had each seen during the A-step $f+1$ answers containing only one value, which is impossible as there are only $2f+1$ processes in all). Hence only two cases are available: either all values B-broadcast at rank i are flagged as $false$, or only one of them is flagged as *true*.

Assume all processes only B-broadcast values flagged as *false*. Either all those values are the same, in which case we already have only one value that can be R-broadcast with a valid certificate. Either there are some different values. The fact that all values are flagged as $false$ indicates that all correct processes have encountered at least two different values during their previous A-step, and thus have discarded the minimum one(s). As processes can only ever R-broadcast greater or equal values due to the max function at every step, it means that all correct processes have discarded at least one value during the A-step. As the number of values and processes are finite, there will eventually be only one value left. Assume now all values B-broadcast are flagged as false but one (if all values are flagged as true, all correct processes commit; no two different values can be flagged as true). Let us call that value v_{true}. The number of processes with flag false at rank i is either $O(n)$, in which case we only need to mention that those processes have each encountered different values at step A (which is why they have a "*false*" flag) and hence have all discarded at least one value. Now let us assume by way of contradiction that there are only $O(1)$ of those processes. We will show that this is impossible. Without loss of generality, we are considering the group of processes which are in the highest rank i. The fact that those $O(1)$ processes delivered some answers to receive the flag "*false*" means that there were $f+1$ correct uninterrupted processes to deliver those answers. Those processes (which total amounts to $O(n)$) can be either in steps R, A or B at the time of sending the message. We will now explore what happens if a $O(n)$ of those processes are in any of those three cases. If there are at least 2 different values each delivered by $f+1$ different processes, then there is at least 1 process that delivered both values. Let us consider those processes.

Consider the $O(n)$ processes in step R. those processes will take step A afterwards and will therefore see the (at least two) values they have delivered. Hence they will also A-return a false, and hence there were $O(n)$ processes with flag "*false*", which is a contradiction.

Consider the $O(n)$ processes in step A. Then those processes have delivered different values in their A-responses, hence they will also A-return a $false$, and hence there were $O(n)$ processes with flag "*false*", which is a contradiction. Consider the $O(n)$ processes in step B. At the same round where they were uninterrupted and they delivered the A-responses that led to the "*false*", they must have B-broadcast the message with flag "$true$". When uninterrupted, the $f+1$ processes will process the B-broadcast of the "$true$" at the same pace as the B-broadcast of the values in "*false*" but with some overhead. Hence the value with flag "$true$" will be delivered before the ones with "*false*", and all the processes with "*false*" will have to adopt that value and at the next R-step only the value flagged "$true$" can be R-broadcast.

Hence at each series of 3 steps R, A and B taken by all processes there are $O(n)$ processes which discard at least one value each. As there are only $O(n)$ different values at most, there will be at most $O(n)$ rounds before there is only one value left to be R-broadcast. □

Theorem Appendix A.9 *(Leaderless termination). In every $\diamond$synchronous-1 execution of OFT-Archipelago, every correct process decides.*

Proof. Assume by the time we reach GST for every correct, uninterrupted process, and no process has yet committed (otherwise all processes are R-broadcasting the same value and Lemma Appendix A.7 ensures termination within 3 steps).

The only way for processes not to commit is for some process to A-return a *false* flag. One way for that to happen is for two different processes (at least) to return different values from an R-step. This may happen if a higher value is received after the $f+1$ first ones by some processes which will ignore it while some other will receive it as part of the $f+1$ first ones and take it in consideration. If that happens, however, that higher value will be disclosed to some new process. Either the value is A-broadcast to all processes, in which case all processes will adopt it and the lowest value is discarded (in which case within $O(n)$ rounds all values will be discarded and termination will happen due to Lemma Appendix A.7). Either some process does not receive that value (or receives it too late), and B-broadcasts another value with true. In this case, all processes will adopt that value and commit at the next B-step due to Lemma Appendix A.7. In both cases, termination happens within $O(n)$ rounds. □

References

[1] Z. A., et al., The Libra blockchain, Tech. Rep., Calibra, revised version of September 25, 2019.

[2] I. Abraham, G. Gueta, D. Malkhi, L. Alvisi, R. Kotla, J.-P. Martin, Revisiting fast practical Byzantine fault tolerance, Tech. Rep., 2017, arXiv:1712.01367.

[3] M.K. Aguilera, C. Delporte-Gallet, H. Fauconnier, S. Toueg, Communication-efficient leader election and consensus with limited link synchrony, in: PODC, 2004.

[4] K. Antoniadis, R. Guerraoui, D. Malkhi, D.-A. Seredinschi, State machine replication is more expensive than consensus, in: DISC, 2018.

[5] J. Aspnes, H. Attiya, K. Censor, Max registers, counters, and monotone circuits, in: PODC, 2009.

[6] H. Attiya, A. Bar-Noy, D. Dolev, Sharing memory robustly in message-passing systems, J. ACM 42 (1) (1995) 124–142.

[7] P. Aublin, S.B. Mokhtar, V. Quéma, RBFT: redundant Byzantine fault tolerance, in: ICDCS, 2013, pp. 297–306.

[8] P.-L. Aublin, R. Guerraoui, N. Knežević, V. Quéma, M. Vukolić, The next 700 BFT protocols, TOCS 32 (4) (2015) 12:1–12:45.

[9] L. Bonniot, C. Neumann, F. Taïani, PnyxDB: a lightweight leaderless democratic Byzantine fault tolerant replicated datastore, in: SRDS, 2020.

[10] F. Borran, A. Schiper, A leader-free Byzantine consensus algorithm, in: ICDCN, 2010.

[11] Z. Bouzid, A. Mostfaoui, M. Raynal, Minimal synchrony for Byzantine consensus, in: PODC, 2015.

[12] E. Buchman, Tendermint: Byzantine fault tolerance in the age of blockchains, MSc Thesis, 2016.

[13] E. Buchman, J. Kwon, Z. Milosevic, The latest gossip on BFT consensus, Tech. Rep., 2018, arXiv:1807.04938.

[14] C. Cachin, R. Guerraoui, L. Rodrigues, Introduction to Reliable and Secure Distributed Programming, Springer, 2011.

[15] M. Castro, B. Liskov, Practical Byzantine fault tolerance and proactive recovery, ACM Trans. Comput. Syst. 20 (4) (2002) 398–461.

[16] T.D. Chandra, S. Toueg, Unreliable failure detectors for reliable distributed systems, J. ACM 43 (2) (1996) 225–267.

[17] P. Civit, S. Gilbert, V. Gramoli, Brief announcement: polygraph: accountable Byzantine agreement, in: DISC, 2020.

[18] P. Civit, M.A. Dzulfikar, S. Gilbert, V. Gramoli, R. Guerraoui, J. Komatovic, M. Vidigueira, Byzantine Consensus Is Theta(n^2): The Dolev-Reischuk Bound Is Tight Even in Partial Synchrony! 2022.

[19] T. Crain, V. Gramoli, M. Larrea, M. Raynal, DBFT: efficient leaderless Byzantine consensus and its application to blockchains, in: NCA, 2018.

[20] T. Crain, C. Natoli, V. Gramoli, Red Belly: a secure, fair and scalable open blockchain, in: S&P, 2021.

[21] K. Croman, C. Decker, I. Eyal, A.E. Gencer, A. Juels, A.E. Kosba, A. Miller, P. Saxena, E. Shi, E.G. Sirer, D. Song, R. Wattenhofer, On scaling decentralized blockchains, in: Financial Cryptography, 2016, pp. 106–125.

[22] D. Dolev, H.R. Strong, Authenticated algorithms for Byzantine agreement, SIAM J. Comput. 12 (4) (1983) 656–666.

[23] C. Dwork, N. Lynch, L. Stockmeyer, Consensus in the presence of partial synchrony, J. ACM 35 (2) (1988) 288–323.

[24] C. Fernández-Campusano, M. Larrea, R. Cortiñas, M. Raynal, Eventual leader election despite crash-recovery and omission failures, in: PRDC, 2015.

[25] M.J. Fischer, N.A. Lynch, A lower bound for the time to assure interactive consistency, Inf. Process. Lett. 14 (4) (1982) 183–186.

[26] E. Gafni, Round-by-round fault detectors: unifying synchrony and asynchrony, in: PODC, 1998.

[27] E. Gafni, L. Lamport, Disk Paxos, Distrib. Comput. 16 (1) (2003) 1–20.

[28] V. Gramoli, L. Bass, A. Fekete, D. Sun, Rollup: non-disruptive rolling upgrade with fast consensus-based dynamic reconfigurations, TPDS 27 (9) (2016) 2711–2724.

[29] D. Gupta, L. Perronne, S. Bouchenak, BFT-Bench: Towards a practical evaluation of robustness and effectiveness of BFT protocols, in: DAIS, 2016.

[30] P. Hunt, M. Konar, F.P. Junqueira, B. Reed, Zookeeper: wait-free coordination for Internet-scale systems, in: ATC, 2010.

[31] L. Lamport, The part-time parliament, TOCS 16 (2) (1998) 133–169.

[32] L. Lamport, Paxos made simple, SIGACT News 32 (4) (2001) 18–25.

[33] L. Lamport, Brief announcement: leaderless Byzantine paxos, in: DISC, 2011.

[34] L. Lamport, R. Shostak, M. Pease, The Byzantine generals problem, ACM Trans. Program. Lang. Syst. 4 (3) (1982) 382–401.

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

[36] Y. Mao, F.P. Junqueira, K. Marzullo, Mencius: building efficient replicated state machines for WANs, in: OSDI, 2008.

[37] C. Martín, M. Larrea, E. Jiménez, Implementing the omega failure detector in the crash-recovery failure model, J. Comput. Syst. Sci. 75 (3) (2009) 178–189.

[38] A. Miller, Y. Xia, K. Croman, E. Shi, D. Song, The honey badger of BFT protocols, in: ACM CCS, 2016, pp. 31–42.

[39] I. Moraru, D.G. Andersen, M. Kaminsky, There is more consensus in egalitarian parliaments, in: SOSP, 2013.

[40] A. Mostefaoui, H. Moumen, M. Raynal, Signature-free asynchronous byzantine consensus with $t < n/3$ and $o(n^2)$ messages, in: PODC, 2014.

[41] D. Ongaro, J. Ousterhout, In search of an understandable consensus algorithm, in: ATC, 2014.

[42] T.F. Rezende, P. Sutra, Leaderless state-machine replication: Specification, properties, limits (extended version), Tech. Rep., 2020, arXiv:2008.02512.

[43] P. Sutra, On the correctness of egalitarian paxos, Inf. Process. Lett. 156 (2020) 105901.

[44] P. Tholoniat, V. Gramoli, Formally verifying blockchain Byzantine fault tolerance, in: FRIDA, 2019, available at https://arxiv.org/pdf/1909.07453.pdf.

[45] G. Voron, V. Gramoli, Dispel: Byzantine SMR with distributed pipelining, Tech. Rep., 2019, arXiv:1912.10367.

[46] M. Yin, D. Malkhi, M.K. Reiter, G. Golan-Gueta, I. Abraham, HotStuff: BFT consensus with linearity and responsiveness, in: PODC, 2019.

[47] M. Yin, D. Malkhi, M.K. Reiter, G.G. Gueta, I. Abraham, Hotstuff: Bft consensus with linearity and responsiveness, in: Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing, 2019, pp. 347–356.

Karolos Antoniadis obtained the PhD degree from EPFL.

Julien Benhaim was a Master's student at EPFL.

Elias Poroma was a Master's student at EPFL.

Antoine Desjardins was a Master's student at EPFL.

Vincent Gramoli is a visting professor at EPFL. He received a Future Fellowship from the Australian Research Council and leads the Concurrent Systems Research Group at the University of Sydney. He is the Founder and CTO of Redbelly Network. His expertise is in distributed computing and security. In the past, Gramoli has been affiliated with INRIA, Cornell and CSIRO. He received his PhD from Université de Rennes and his Habilitation from UPMC Sorbonne University.

Rachid Guerraoui has been affiliated with Ecole des Mines of Paris, the Commissariat à l'Energie Atomique of Saclay, Hewlett Packard Laboratories and the Massachusetts Institute of Technology. He has worked in a variety of aspects of distributed computing, including distributed algorithms and distributed programming languages. He is most well known for his work on (e-)Transactions, epidemic information dissemination and indulgent algorithms. He co-authored a book on Transactional Systems (Hermes) and a book on reliable distributed programming (Springer). He was appointed program chair of ECOOP 1999, ACM Middleware 2001, IEEE SRDS 2002, DISC 2004 and ACM PODC 2010.

Gauthier Voron is a postdoctoral fellow at EPFL.

Igor Zablotchi obtained the PhD degree from EPFL. He is now a postdoctoral fellow at the Massachusetts Institute of Technology, MA, USA.