Leaderless Consensus

Karolos Antoniadis
DCL, EPFL
karolos.antoniadis@epfl.ch

Antoine Desjardins
DCL, EPFL
antoinedesjard@gmail.com

Vincent Gramoli
University of Sydney and EPFL
vincent.gramoli@sydney.edu.au

Rachid Guerraoui
DCL, EPFL
rachid.guerraoui@epfl.ch

Igor Zablotchi
DCL, EPFL
igor.zablotchi@epfl.ch

***Abstract*—Classical synchronous consensus algorithms are *leaderless*: processes exchange their proposals, pick the max and decide when they see the same choice across a couple of rounds. Indulgent consensus algorithms are more robust in that they only require eventual synchrony, but are however typically *leader-based*. Intuitively, this is a weakness for a slow leader can delay any decision.**

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 (with and without authentication).

***Index Terms*—Leaderless termination, Byzantine, synchronous-k, synchronizer, fast-path**

I. 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 [1], [7], [11], [16], [24], [25], [27], [33], [35], [38], [39]. 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. The choice of the timeout to suspect a faulty leader and replace it impacts drastically performance [27], [36], sometimes by two orders of magnitude [24]. Besides, replacing the leader requires a view-change protocol that is so complex that research prototypes often omit it [17] or implement it with notorious errors [1].

Various efforts have been recently devoted to minimize the role of the leader. One idea is to change the leader frequently even if it is not suspected to have failed [11], [39]. Another is to bypass the leader bottleneck by having multiple proposers [15], [16], [38] before reverting to a weak coordinator to converge. A third one is to tolerate multiple leaders for different consensus instances [24], [33], [35], 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.

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 [14]. 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) [2], [10].

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

First, to address this question, we formally define the notion of "leaderless". 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 (or $\diamond$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 [22] 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 algorithms [13] 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 and time complexity.

The rest of the paper is organized as follows. Section II gives some necessary background. Section III formalizes the notion of a leaderless consensus algorithm and explains why well-known leader-based consensus algorithms do not satisfy this definition. Section IV presents three leaderless consensus algorithms, one for shared memory, another to tolerate omission failures in message passing and a third one to tolerate Byzantine failures. Section V discusses the complexities of our algorithms. Section VI discusses related work.

A series of appendices are left to the discretion of the reader. Appendix A presents an example of an algorithm that fails to ensure safety in the $\diamond$synchronous-1 model. Appendix B shows that the adopt-commit-max algorithm is correct. Appendix C proves the shared memory algorithm solves the consensus problem. Appendix D proves that the shared memory algorithm satisfies leaderless termination. Appendix E proves that Archipelago terminates in any synchronous execution with up to $f = n - 1$ faulty processors. Appendix F proves that the message passing variant of Archipelago is correct despite crash failures. Appendix G proves Archipelago correct despite omission failures. Appendix H proves BFT-Archipelago correct and Appendix I shows the complexity of BFT-Archipelago. Appendix J presents BFTU-Archipelago as a variant of BFT-Archipelago without authentication. Appendix K presents the complexity of BFTU-Archipelago.

II. 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 notational simplicity, 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*

[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.

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.

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 exact same configurations and events in the same order.

Synchronous$-k$ execution. We can now precise what it means for an execution to be synchronous in a shared-memory system. Our definition is inspired by the definition 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 [19] such that, in each round, every process p does the following: (i) sends a message to every other process in the system, and (ii) delivers any message that was sent to p and performs some local computation.

To adapt synchrony to 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 $\mathcal{R}_j$. 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 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 equal to 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 events in r. For this reason, we call such an execution "synchrony minus k," since all processes except k behave synchronously in each round. 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. Figure 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 Figure 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.

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 [12], each process proposes a value by invoking a $\mathsf{propose}(v)$ 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 $\mathsf{propose}(v)$ function call by some process p returns in α.

III. 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 leaderless termination as follows.

Definition 2 (Leaderless Algorithm)**.** *A consensus algorithm is* leaderless *if it satisfies validity, agreement and termination, as well as 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 IV-B. 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. Appendix C offers a detailed argument. Essentially, one can design a consensus algorithm that decides in finite time in all synchronous-1 executions, but could however violate safety in an $\diamond$synchronous-1 execution (see Appendix A for such an algorithm). The challenge is, instead, to devise a leaderless consensus algorithm that decides in finite time in every $\diamond$synchronous-1 execution and never violates safety. Section IV-A presents three leaderless consensus algorithms that tolerate

	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.

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 [8], [16] and reducing the impact of a single point of failure on performance [7], [38] 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, both our authenticated Byzantine fault tolerant leaderless algorithm, BFT-Archipelago, and its version without signatures, BFTU-Archipelago (Algorithm 7), share the same communication complexity as PBFT [13] and DBFT [16], namely $O(n^4)$ bits. Finally, since BFT-Archipelago can be written as an Abstract [6] (see Section V), it is compatible with leader-based consensus instances and inherits an optimal fast paths in good executions.

Paxos: a counter example. Consider Algorithm 1, a leader-based algorithm that, when combined with a leader election, corresponds to Paxos [28] in shared memory (or more specifically to Disk Paxos [23] 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 [29]. During the prepare phase, p_i

Algorithm 1 Leader-based consensus algorithm

```
1:  Shared state:
2:    $R[n] \leftarrow \{\langle\bot,0\rangle,\ldots,\langle\bot,0\rangle\}$   ▷ 1 SWMR reg. per proc.

3:  Local state:
4:    $ts \leftarrow i$   ▷ for process $p_i$

5:  procedure propose($v$):   ▷ process $p_i$ proposes value $v$
6:    while true do
7:      $R[i].ts \leftarrow ts$
8:      $val \leftarrow$ getHighestTspValue($R$)
9:      if $val = \bot$ then
10:       $val \leftarrow v$
11:     $R[i] \leftarrow \langle val, ts\rangle$
12:     if $ts =$ getHighestTsp($R$) then
13:       return $val$
14:     $ts \leftarrow ts + n$
```

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 1 is leader based. In fact, Algorithm 1 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).

IV. Leaderless Consensus Algorithms

In this section, we present a series of leaderless consensus algorithms, called Archipelago. For pedagogical reasons, we introduce a simple shared memory version before its message-passing variant, called Archipelago, and finally a Byzantine fault tolerant variant, called BFT-Archipelago.

A. *Archipelago: A Leaderless Consensus Algorithm*

Archipelago satisfies Definition 1 when $n \geq 3$ and never violates safety. It builds upon a new variant of an adopt-commit object [22], 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* [22] 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 $\langle \mathsf{commit}, v \rangle$ or $\langle \mathsf{adopt}, v \rangle$, then v was proposed by some process.
- **CA-Agreement**: If a process p outputs $\langle \mathsf{commit}, v \rangle$ and a process q outputs $\langle \mathsf{commit}, v' \rangle$ or $\langle \mathsf{adopt}, v' \rangle$, then $v = v'$.
- **CA-Commitment**: If every process proposes the same value, then no process may output $\langle \mathsf{adopt}, \cdot \rangle$.
- **CA-Termination**: Every correct process eventually obtains an output.

Algorithm 2 depicts a new implementation of an adopt-commit object. It differs from the classic implementation [22] in that if the collect of A by process p that proposes v returns different values, then p stores $\langle \mathsf{adopt}, mv \rangle$ to array B (line 9) instead of storing $\langle \mathsf{adopt}, v \rangle$, 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 $\langle \mathsf{adopt}, \cdot \rangle$, then process p returns $\langle \mathsf{adopt}, mv \rangle$, where mv is $max(S_A)$ (line 14). Note that Algorithm 2 is just a different implementation of the classic implementation [22] and that the main properties of an adopt-commit object remain the same. These modifications are crucial for the leaderless termination of Archipelago.

We defer the correctness proof of Algorithm 2, which is similar to that of an adopt-commit object [22], to the companion technical report.

The Archipelago Algorithm. Algorithm 3 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 [4]. 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 $\langle \mathsf{commit}, \cdot \rangle$ and Archipelago decides in finite time.

Algorithm 2 The adopt-commit-max algorithm

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

4:  procedure propose($v$): ▷ taken by a process $p_i$
5:     $A[i] \leftarrow v$  ▷ step $A$ starts
6:     $S_A \leftarrow$ collect($A$)  ▷ step $A$ ends
7:     if ($S_A \setminus \{\perp\} = \{v'\}$) then  ▷ step $B$ starts
8:        $B[i] \leftarrow \langle$commit$, v'\rangle$
9:     else $B[i] \leftarrow \langle$adopt$, \max(S_A)\rangle$ ▷ or step $B$ starts
10:    $S_B \leftarrow$ collect($B$)  ▷ step $B$ ends
11:    if $S_B \setminus \{\perp\} = \{\langle$commit$, v'\rangle\}$ then
12:       return $\langle$commit$, v'\rangle$
13:    else if $\langle$commit$, v'\rangle \in S_B$ then return $\langle$adopt$, v'\rangle$
14:    else return $\langle$adopt$, max(S_B)\rangle$
```

Algorithm 3 Archipelago leaderless consensus

```
15: Shared state:
16:    $C[0, \ldots, +\infty]$, an infinite array of adopt-commit-max
17:       objects in their initial state
18:    $m$, a max register object that initially contains $\langle 0, \perp \rangle$.
19:    Note that $\langle x, y \rangle > \langle x', y' \rangle$ 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($\langle c, v \rangle$)  ▷ step $R$ starts
26:       $\langle c', v' \rangle \leftarrow m$.readmax()  ▷ step $R$ ends
27:       $\langle control, v'' \rangle \leftarrow C[c']$.propose($v'$)
28:       $c \leftarrow c' + 1$
29:       if $control =$ adopt then $v \leftarrow v''$
30:       else return $v''$
```

More precisely, Algorithm 3 performs repeatedly three steps (by writing and collecting as defined in Section II) 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 2 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 2 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 $\diamond$synchronous-1 execution with Archipelago, we could implement the Ω failure detector [14]. We could then augment Algorithm 1 with Ω so that Algorithm 1 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 [12], [20], [34]. 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* [34], hence the existence of Ω in our model is impossible.

Theorem IV.1. *Archipelago satisfies leaderless termination for $n \geq 3$.*

To prove Theorem IV.1, 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). Therefore, eventually only one value gets proposed to some adopt-commit-max object, and every correct process decides. For brevity, we defer the proof of Theorem IV.1 to Appendix D.

B. *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.

$\diamond$synchronous$-k$ in message passing. To preserve the definition of $\diamond$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) a 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 [19]. We say that a process p is *suspended* [3] 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 ABD emulation [5] to Algorithm 3. 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 4 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.

This way of combining writes and reads can break atomicity, but is sufficient to guarantee safety (of consensus) during asynchronous periods. More precisely, the R-Step behaves like a "regular" max-register, one that returns valid, non-decreasing values to each invoker (see Lemma A.9 in Ap-

Algorithm 4 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:     $A[0, 1, \ldots]$, a sequence of sets, all initially empty
5:     $B[0, 1, \ldots]$, a sequence of sets, all initially empty

6:  **procedure** propose($v$):
7:     **while** true **do**
8:        $\langle i, v' \rangle \leftarrow$ R-Step($v$)
9:        $\langle flag, v'' \rangle \leftarrow$ A-Step($v'$)
10:       $\langle control, val \rangle \leftarrow$ B-Step($flag, v''$)
11:       **if** $control =$ commit **then return** val
12:       **else** $i \leftarrow i + 1$

13: **procedure** R-Step($v$):
14:    broadcast($R, i, v$)
15:    **wait until** receive (R-response, $i, R$) from $f + 1$ proc.
16:    $R \leftarrow R \cup \{$ union of all $R$s received in previous line$\}$
17:    $\langle i', v' \rangle \leftarrow \max(R)$
18:    **return** $\langle i', v' \rangle$

19: **procedure** A-Step($v$):
20:    broadcast($A, i, v$)
21:    **wait until** receive (A-response, $i, A[i]$) from $f+1$ proc.
22:    $\mathcal{S} \leftarrow$ union of all $A[i]$s received
23:    **if** $\mathcal{S}$ contains only one value $val$ **then return** $\langle$true$, val\rangle$
24:    **else return** $\langle$false$, \max(\mathcal{S})\rangle$

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:    $\mathcal{S} \leftarrow$ union of all $B[i]$s received
29:    **if** $\mathcal{S}$ contains only $\langle$true$, val\rangle$ for some $val$ **then**
30:       **return** $\langle$commit$, val\rangle$
31:    **else if** $\mathcal{S}$ contains some entry $\langle$true$, val\rangle$ **then**
32:       **return** $\langle$adopt$, val\rangle$
33:    **else return** $\langle$adopt$, max(\mathcal{S})\rangle$

34: **upon reception of** $(R, j, v)$ **from** $p$:
35:    Add $\langle j, v \rangle$ 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 $\langle flag, v \rangle$ to $B[j]$
42:    send(B-response$, j, B[j]$) to $p$
```

pendix F), and the A- and B-Steps together behave like an adopt-commit object (see Lemma A.10 in Appendix F). As such, our proof of safety in Section C applies to Algorithm 4 as well.

The non-atomic behavior exhibited during asynchronous periods is due to the overlap in time of the request and response parts of each round. However, during synchronous-1 periods, we can assume that requests are delivered by all processes before any response is sent out. Thus, once the system becomes permanently synchronous-1, the R-Step satisfies the (atomic) max-register properties and the A- and B-Steps together behave like an adopt-commit-max object. Therefore, our proof of leaderless termination in Section C remains valid for Algorithm 4 as well. Due to space constraints, we defer the proof of Algorithm 4 to Appendix F.

C. *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 $n \leq 3f$ [32], we assume the $\diamond$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 in the presentation, we also assume authentication. The alternative unauthenticated variant, BFTU-Archipelago, and the proof that the result generalizes to the $\diamond$synchronous$-k$ model, where $k \leq f$ and $f - k$ processes can be Byzantine, is deferred to the Appendix.

The R-**,** A-**, and** B-**Steps.** BFT-Archipelago is depicted in Algorithm 5 and follows the same 3-step pattern as Archipelago, with the R-, A- and B-Steps, executed in consecutive loop iterations, called *ranks*.

- R-Step: process p gathers the $rank$ and $value$ of other processes with the aim to settle on a common $(rank, value)$ (lines 17–24). Processes answer the R-broadcast (if they find it valid) 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

Algorithm 5 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:   $A[0, 1, \ldots]$ and $B[0, 1, \ldots]$, 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:    $\langle i, v' \rangle \leftarrow$ R-Step($v$)
11:    $\langle flag, v'' \rangle \leftarrow$ A-Step($i, v'$)
12:    $\langle contr, val \rangle \leftarrow$ B-Step($flag, i, v''$)
13:    if $contr$ = commit then return val
14:    else $i \leftarrow i + 1$, $v \leftarrow val$

15: procedure R-Step($v$):
16:   compile certificate $\mathcal{C}$ (empty at rank 0)
17:   broadcast($R, i, v, \mathcal{C}$)
18:   wait until (receive valid (Rresp, $i$, $R, \mathcal{C}$)
19:     from $2f + 1$ processes)
20:   $R \leftarrow R \cup \{$union of all valid $R$s received
21:     in previous line$\}$
22:   $\langle i', v' \rangle \leftarrow$ max($R$)
23:   $R \leftarrow$ max($R$)
24:   return $\langle i', v' \rangle$

25: upon delivering $(R, j, v, \mathcal{C})$ from $p$:
26:   if reliability check($R, j, v, \mathcal{C}$) then
27:     $R \leftarrow max(\langle j, v \rangle, R)$
28:     $b \leftarrow$ 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 $\mathcal{C}$
33:   broadcast($A, i, v, \mathcal{C}$)
34:   wait until receive valid (Aresp, $i$, $A[i]$)
35:     from $2f + 1$ processes
36:   $\mathcal{S} \leftarrow$ union of all $A[i]$s received
37:   if ($\mathcal{S}$ contains at least 2f+1 A-answers
38:     containing only $val$) then
39:     return $\langle$true, $val\rangle$
40:   else return $\langle$false, max($\mathcal{S}$)$\rangle$

41: upon delivering $(A, j, v, \mathcal{C})$ from $p$:
42:   if reliability check($A, j, v, \mathcal{C}$) then
43:     if $v \notin A[j]$ and $|A[j]| < 2$ then
44:       add $v$ to $A[j]$
45:     else if $v > \max(A[j])$ then
46:       $\min(A[j]) \leftarrow v$
47:     $b \leftarrow$ 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($\int, i, v$):
51:   compile certificate $\mathcal{C}$
52:   broadcast($B, i, \int, v, \mathcal{C}$)
53:   wait until receive valid (Bresp, $i$, $B[i]$)
54:     from $2f + 1$ proc.
55:   $\mathcal{S} \leftarrow$ array with all $B[i]$s received
56:   if $|\{\langle$true, $val\rangle \in S\}| \geq 2f + 1$ then
57:     return $\langle$commit, $val\rangle$
58:   else if $|\{\langle$true, $val\rangle \in \mathcal{S}\}| \geq 1$ then
59:     return $\langle$adopt, $val\rangle$
60:   else return $\langle$adopt, $\max(\mathcal{S})\rangle$

61: upon delivery $(B, j, \int, v, \mathcal{C})$ from $p$:
62:   if reliability check($B, j, v, \mathcal{C}$) then
63:     $m \leftarrow max(B[j][0].v, B[j][1].v)$
64:     if $|B[j]| < 2$ then add $\langle \int, v \rangle$ to $B[j]$
65:     else if $(\int \wedge \langle \int, v \rangle \notin B[j] \vee$
66:     $\neg \int \wedge v > m)$ then
67:       $B[j][0] \leftarrow \langle \int, v \rangle$
68:       $b \leftarrow$ bcast resp. for $B[j]$'s $\langle \int, vals \rangle$
69:       send(Bresp, $j$, $B[j]$, $sig, b$)
70:     $b \leftarrow$ resp. for $B[j]$'s $\langle \int, vals \rangle$
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 $|\{bcast\text{-}answers \in C\}| > f$ then
75:     return true
76:   check that $|\mathcal{C}| \geq 2f + 1$ messages
77:   check signatures of those messages
78:   check if $|\{bcast\text{-}answers\}| > f$
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 $(i, \int, v)$ 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 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.
```

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 explained 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 p_j response. 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 ensures 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. The reason why this complementing certificate contains more responses is to avoid waiting for $f+1$ responses that are never received (e.g., responses responding to a broadcast sent by a malicious broadcaster). Waiting for $2f+1$ responses before considering the response valid guarantees that other correct processes eventually receive $f+1$ responses.

BFT-Archipelago satisfies Validity, Agreement and Leaderless Termination, just like Archipelago.

Theorem IV.2. *In every $\diamond$synchronous-1 execution of BFT-Archipelago, every correct process decides.*

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. The full proof is deferred to Appendix H.

V. Discussion and Complexity Analysis

Termination. Archipelago satisfies termination for $n \geq 3$, meaning that in an eventually synchronous [12] execution, every correct process eventually decides. In such an execution, Archipelago needs at most 5 rounds, after the global stabilization time [19] and round synchronization (i.e., all processes start and end a round at the same time). The proof is deferred to Appendix E.

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 [6] that allows chaining together multiple BFT protocols, called Abstract instances that can abort to 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 [19] and time-optimal [18], [21]. BFT-Archipelago is also competitive with PBFT [13] and DBFT [16], having the same communication complexity. The detailed proof is deferred to Appendix I.

VI. Related Work

Given that the need for a leader has been recognized to hamper the performance of consensus algorithms [1], [7], [8], [11], [16], [24], [25], [27], [33], [35], [38], [39], it is surprising that the concept of a leaderless protocol has never been precised.

With the recent need to scale consensus to large blockchain networks, the "leadership" problem has been exacerbated and several protocols have been designed to mitigate the problem. Crain et al. [16] proposed DBFT for blockchains. DBFT bypasses the leader bottleneck by relying on n binary consensus instances, each using a weak coordinator to converge even if sufficiently many correct processes propose distinct values. DBFT is not leaderless according to our definition due its requirement of a weak coordinator. Maofan et al. [39] replaced the leader's large proposals of PBFT [13] by smaller message digests to obtain HotStuff. The throughput of HotStuff drops to zero when the leader fails and until some view-change completes [38].

In a brief announcement [30], 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 [31], Lamport explains that during a period of asynchrony, if the virtual leader election fails, then the consensus algorithm may not progress but should not violate safety as long as it tolerates malicious leaders [30]. 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 [9] 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 [33], RBFT [7]) do not necessarily rely on a leaderless consensus algorithm.

Moraru et al. [35] used multiple "command leaders" in EPaxos. Each leader commits one command as long as commands are compatible. However, in the general case, where 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 acknowledgement 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. Despite being specified in TLA+, EPaxos specification was recently shown incorrect [37], indicating that designing a multi-leader algorithms is error prone.

VII. Concluding remarks

Our definition of leaderless is general and can be applied to different problems. It relies on the ability to tolerate 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. The challenge to address when building a leaderless algorithm is that of terminating despite the occurrence of such interruptions.

References

[1] Ittai Abraham, Guy Gueta, Dahlia Malkhi, Lorenzo Alvisi, Rama Kotla, and Jean-Philippe Martin. Revisiting fast practical Byzantine fault tolerance. Technical Report 1712.01367, arXiv, 2017.

[2] Marcos K. Aguilera, Carole Delporte-Gallet, Hugues Fauconnier, and Sam Toueg. Communication-efficient leader election and consensus with limited link synchrony. In *PODC*, 2004.

[3] Karolos Antoniadis, Rachid Guerraoui, Dahlia Malkhi, and Dragos-Adrian Seredinschi. State machine replication is more expensive than consensus. In *DISC*, 2018.

[4] James Aspnes, Hagit Attiya, and Keren Censor. Max registers, counters, and monotone circuits. In *PODC*, 2009.

[5] Hagit Attiya, Amotz Bar-Noy, and Danny Dolev. Sharing memory robustly in message-passing systems. *JACM*, 42(1):124–142, 1995. doi:10.1145/200836.200869.

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

[7] Pierre-Louis Aublin, Sonia Ben Mokhtar, and Vivien Quéma. RBFT: redundant Byzantine fault tolerance. In *ICDCS*, pages 297–306, 2013.

[8] Loïck Bonniot, Christoph Neumann, and François Taïani. PnyxDB: a lightweight leaderless democratic Byzantine fault tolerant replicated datastore. In *SRDS*, 2020.

[9] Fatemeh Borran and André Schiper. A leader-free Byzantine consensus algorithm. In *ICDCN*, 2010.

[10] Zohir Bouzid, Achour Mostfaoui, and Michel Raynal. Minimal synchrony for Byzantine consensus. In *PODC*, 2015.

[11] Ethan Buchman, Jae Kwon, and Zarko Milosevic. The latest gossip on BFT consensus. Technical Report 1807.04938, arXiv, 2018.

[12] Christian Cachin, Rachid Guerraoui, and Luìs Rodrigues. *Introduction to Reliable and Secure Distributed Programming*. Springer, 2011.

[13] Miguel Castro and Barbara Liskov. Practical Byzantine fault tolerance and proactive recovery. *TOCS*, 20(4), 2002.

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

[15] Pierre Civit, Seth Gilbert, and Vincent Gramoli. Brief announcement: Polygraph: Accountable Byzantine agreement. In *DISC*, 2020.

[16] Tyler Crain, Vincent Gramoli, Mikel Larrea, and Michel Raynal. DBFT: Efficient leaderless Byzantine consensus and its application to blockchains. In *NCA*, 2018.

[17] Kyle Croman, Christian Decker, Ittay Eyal, Adem Efe Gencer, Ari Juels, Ahmed E. Kosba, Andrew Miller, Prateek Saxena, Elaine Shi, Emin Gün Sirer, Dawn Song, and Roger Wattenhofer. On scaling decentralized blockchains. In *Financial Cryptography*, pages 106–125, 2016.

[18] Danny Dolev and H. Raymond Strong. Authenticated algorithms for Byzantine agreement. *SIAM Journal on Computing*, 12(4):656–666, 1983.

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

[20] Christian Fernández-Campusano, Mikel Larrea, Roberto Cortiñas, and Michel Raynal. Eventual leader election despite crash-recovery and omission failures. In *PRDC*, 2015.

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

[22] Eli Gafni. Round-by-round fault detectors: Unifying synchrony and asynchrony. In *PODC*, 1998.

[23] Eli Gafni and Leslie Lamport. Disk Paxos. *Distributed Computing*, 16(1):1–20, 2003.

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

[25] Divya Gupta, Lucas Perronne, and Sara Bouchenak. BFT-Bench: Towards a practical evaluation of robustness and effectiveness of BFT protocols. In *DAIS*, 2016.

[26] Robert Gurwitz and Robert Hinden. Ip - local area network addressing issues, 1982. https://www.rfc-editor.org/ien/ien212.txt.

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

[28] Leslie Lamport. The part-time parliament. *TOCS*, 16(2):133–169, 1998.

[29] Leslie Lamport. Paxos made simple. *ACM Sigact News*, 32(4):18–25, 2001.

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

[31] Leslie Lamport. Brief announcement: Leaderless Byzantine Paxos. In *DISC*, 2011.

[32] Leslie Lamport, Robert Shostak, and Marshall Pease. The Byzantine generals problem. *TOPLAS*, 4(3):382–401, 1982.

[33] Yanhua Mao, Flavio P. Junqueira, and Keith Marzullo. Mencius: Building efficient replicated state machines for WANs. In *OSDI*, 2008.

[34] Cristian Martín, Mikel Larrea, and Ernesto Jiménez. Implementing the omega failure detector in the crash-recovery failure model. *Journal of Computer and System Sciences*, 75(3):178–189, 2009.

[35] Iulian Moraru, David G. Andersen, and Michael Kaminsky. There is more consensus in egalitarian parliaments. In *SOSP*, 2013.

[36] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *ATC*, 2014.

[37] Pierre Sutra. On the correctness of egalitarian paxos. *Inf. Process. Lett.*, 156:105901, 2020.

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

[39] Maofan Yin, Dahlia Malkhi, Michael K. Reiter, Guy Golan-Gueta, and Ittai Abraham. HotStuff: BFT consensus with linearity and responsiveness. In *PODC*, 2019.

APPENDIX

A. *Consensus Algorithm for Synchronous*-1 *Executions that Violates Agreement in an* $\diamond$*Synchronous*-1 *Execution*

We present Algorithm 6, an example of a consensus algorithm that decides in every synchronous-1, but that violates safety (i.e., agreement) when executed in an $\diamond$synchronous-1 execution.

Algorithm 6 Consensus algorithm that correctly decides in every synchronous-1 execution

```
1:  Shared state:
2:    $Reg[n] \leftarrow \{\bot, \dots, \bot\}$  ▷ array of $n$ single-writer multi-reader registers

3:  ▷ process $p_i$ proposes value $v$
4:  procedure propose($v$):
5:    ▷ first round
6:    $Reg[i] \leftarrow v$
7:    $vals \leftarrow \mathsf{collect}(Reg) \setminus \{\bot\}$
8:    if $\exists\langle \mathsf{commit}, cv\rangle \in vals$ then
9:      $dv \leftarrow cv$  ▷ $p_i$ was suspended in the first round, hence adopt committed value
10:   else
11:     $dv \leftarrow \max(\{v : v \in vals \lor \langle\cdot, v\rangle \in vals\})$
12:
13:   ▷ second round
14:   $Reg[i] \leftarrow \langle \mathsf{commit}, dv\rangle$
15:   return $dv$
```

Algorithm 6 satisfies validity, agreement, and decides in finite time in every synchronous-1 execution. Clearly, Algorithm 6 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, then the algorithm can violate safety.

We prove that Algorithm 6 satisfies validity, agreement, and decides in finite time in every synchronous-1 execution below. *Validity.* Each process writes the proposed value in $Reg[i]$ (line 6) and then collects (line 7) all the values written in Reg. Hence, variable $vals$ contains only proposed values. Then, if there is a $\langle \mathsf{commit}, cv\rangle$ pair in $vals$ the algorithm decides cv, stores $\langle \mathsf{commit}, cv\rangle$ 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 $\langle \mathsf{commit}, cv \rangle$ in $Reg[i]$ and returns (line 14).

Agreement. Algorithm 6 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 $\langle \mathsf{commit}, v \rangle$ (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 $\langle \mathsf{commit}, dv \rangle$ 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.

In a model with $n = 2$ processes, Algorithm 6 could violate agreement, 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.

B. *Correctness of the Adopt-commit-max Object*

Algorithm 2 satisfies CA-Validity (the max function preserves validity) and CA-Termination (Algorithm 2 does not use waiting or loops). To prove CA-Agreement and CA-Commitment, we first prove the following lemma.

Lemma A.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 $\langle \mathsf{commit}, v \rangle$ to B, collect B and see only entries equal to $\langle \mathsf{commit}, v \rangle$ and finally return $\langle \mathsf{commit}, v \rangle$.

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 $\langle \mathsf{commit}, v \rangle$ entry written by p, otherwise q would adopt v (remember that by Lemma A.1, q cannot see any entry $\langle \mathsf{commit}, v' \rangle$ with $v' \neq v$ in B since p writes $\langle \mathsf{commit}, v \rangle$ 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 $\langle \mathsf{adopt}, \cdot \rangle$ in B; B contains only entries of the form $\langle \mathsf{commit}, \cdot \rangle$. By Lemma A.1, all such entries have equal values, so all processes that return must commit.

C. *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 $\langle \mathsf{adopt}, v\rangle$ 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 $\langle \mathsf{commit}, v\rangle$ response for an adopt-commit-max object $C[c]$ and process p' received a $\langle \mathsf{commit}, v'\rangle$ 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 $\langle \mathsf{commit}, v\rangle$ or $\langle \mathsf{adopt}, v\rangle$ 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 $\langle \mathsf{commit}, v\rangle$. Similarly, all upcoming adopt-commit-max-objects return $\langle \mathsf{commit}, v\rangle$ contradicting the fact that $C[c']$ ($c < c'$) responds with $\langle \mathsf{commit}, v'\rangle$ with $v' \neq v$.

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 $\langle \mathsf{commit}, v\rangle$ (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 $\langle \mathsf{adopt}, v\rangle$ (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 Figure 2. Figure 2 has a pattern that repeats every 5 rounds (light-green boxes). In Figure 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 $\langle \mathsf{adopt}, v' \rangle$. Process p_2 takes a B step in the fifth round after being suspended in the third and fourth rounds, p_2 writes $\langle \mathsf{commit}, v \rangle$ in $C[0].B[2]$, and then during the collect of B, p_2 sees that $\langle \mathsf{adopt}, v' \rangle$ is written in $C[0].B[1]$ and p_2 returns $\langle \mathsf{commit}, v \rangle$ (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.

D. *Proof of Archipelago's Leaderless Termination*

In this section, we prove the Theorem IV.1. Note that we prove Theorem IV.1 that Archipelago terminates in finite time in every synchronous-1 execution, irrespectively of the initial state (i.e., any state that corresponds to a configuration in a well-formed execution). Therefore, in every $\diamond$synchronous-1 execution, eventually the execution becomes synchronous-1 and hence Archipelago decides in finite time.

Theorem A.2 (Theorem IV.1)**.** *Archipelago satisfies leaderless termination for $n \geq 3$.*

To prove Theorem IV.1 we first need to prove some auxiliary lemmas.

Lemma A.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 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 A.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 $\langle \mathsf{commit}, v \rangle$ when collecting B (line 10) and either returns $\langle \mathsf{commit}, v \rangle$ (line 12 and then line 30) and decides, or returns $\langle \mathsf{adopt}, v \rangle$ (line 13). Due to CA-agreement, p cannot return $\langle \mathsf{commit}, v' \rangle$ $\langle \mathsf{adopt}, v' \rangle$ 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 A.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

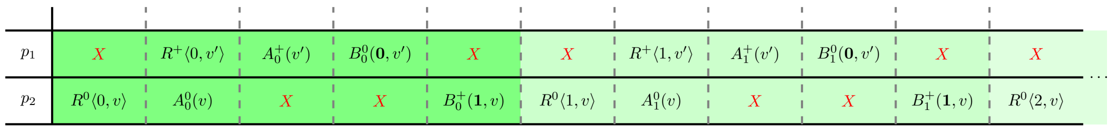

Fig. 2. With 2 processes, Archipelago might never decide in a synchronous-1 execution $(v' > v)$.

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 $B_i^0(\mathbf{1}, v)$. $\square$

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 A.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 $v' \geq v > v_m$. $\square$

To prove Theorem IV.1 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 A.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.

Figure 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 A.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 Figure 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 A.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 A.4 and A.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 A.4,

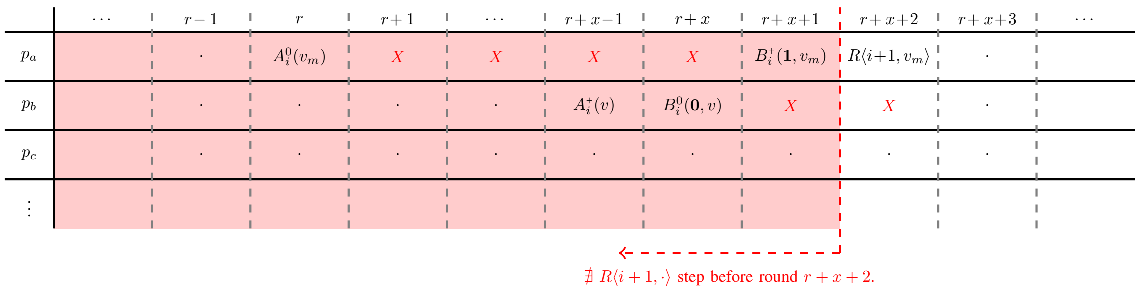

Fig. 3. Execution pattern that appears when the minimum value propagates to the next adopt-commit-max object ($x \geq 2$).

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 A.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 figure 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 A.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 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 A.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 figure 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 A.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 figure 6.

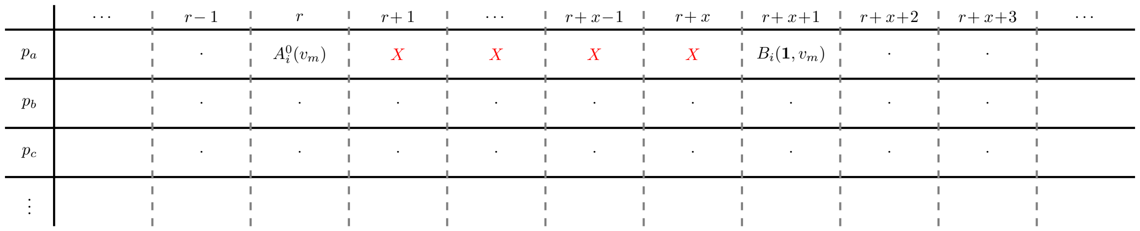

Fig. 4. Long suspension of process p_a with value v_m

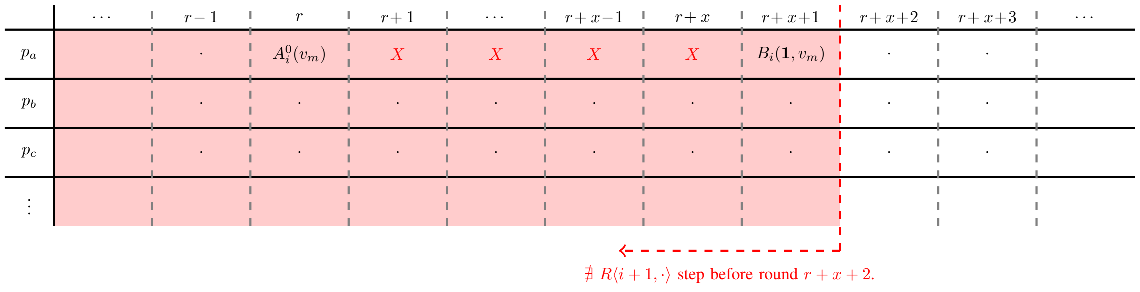

Fig. 5. Impossibility of an $R\langle i+1, \cdot\rangle$ step before round $r+x+2$

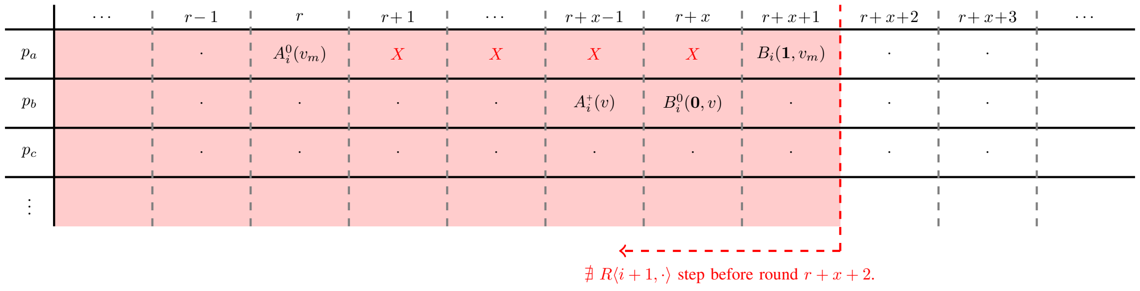

Fig. 6. process p_b performs a step $A_i(v)$ with $v > v_m$ in round $r+x-1$ and $B_i^0(\mathbf{0}, v)$ in round $r+x$

Due to Lemma A.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 A.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 Figure 7 that is the exactly the same execution pattern as the one in Figure 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

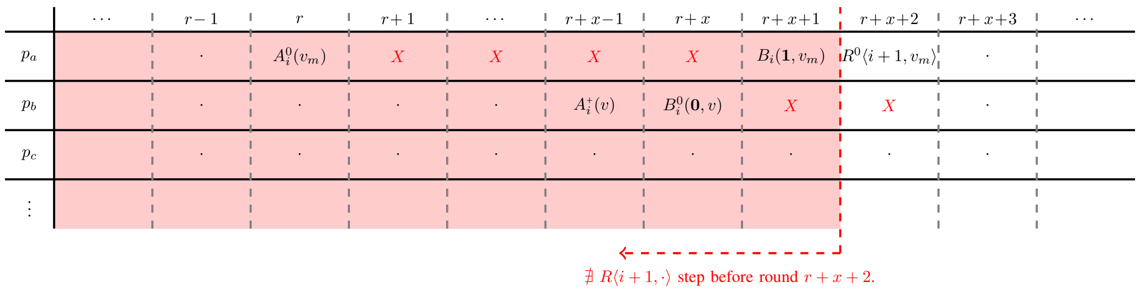

Fig. 7. Execution pattern that appears when the minimum value propagates to the next adopt-commit-max object ($x \geq 2$)

as shown in Figure 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 A.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 A.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 A.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). Because 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 A.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 A.7, we know that if there is a $j \geq i+3$ with $A_j(v_m)$, then the execution looks like Figure 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 Figure 3 but for adopt-commit-max object $C[i+1]$. See Figure 9. Again, let $p_{a'}$ and $p_{b'}$ be the processes described in Lemma A.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 A.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

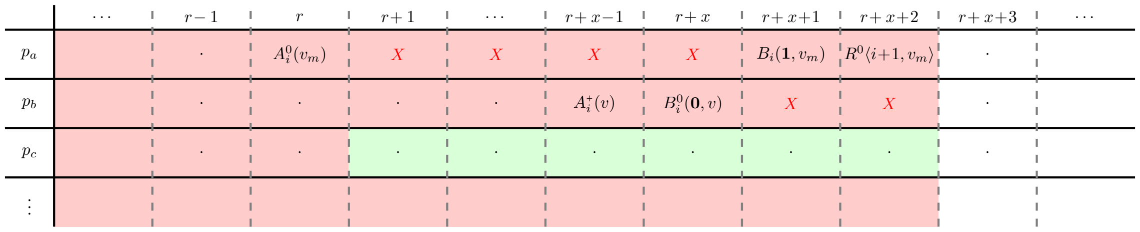

Fig. 8. Lemma A.8 (1)

$r' + x' + 2$. 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 A.7, this pattern of execution should appear for adopt-commit-max object $C[i+2]$. Consider Figure 10. Again, let $p_{a''}$ and $p_{b''}$ be the processes described in Lemma A.7 with respect to A_{i+2} and let $p_{c''}$ be any other process. By Lemma A.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 A.7.

□

Lemma A.8 implies Theorem IV.1, 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 consensus, 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.

E. Archipelago in the Common Case

In this section we show that Archipelago terminates in any $\diamond$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 $\diamond$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 $A_i^0(v)$ and then $B_i^0(\mathbf{1}, v)$ and due to Lemma A.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 $A_i^0(v)$ and $B_i^0(\mathbf{1}, v)$ and hence the algorithm decides (see Lemma A.4).

The above discussion implies that Archipelago satisfies termination, thus meaning that in an $\diamond$synchronous execution, Archipelago decides. Furthermore, note that the Archipelago can withstand up to $f = n-1$ faulty processors and decides in an $\diamond$synchronous execution. Naturally, the message passing variant of

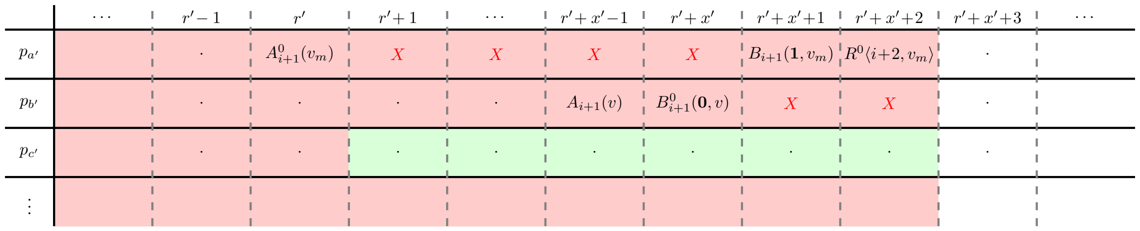

Fig. 9. Lemma A.8 (2)

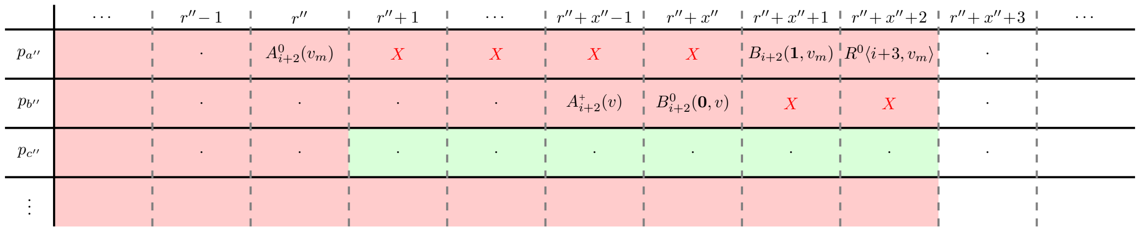

Fig. 10. Lemma A.8 (3)

Archipelago (Section IV-B) can only withstand up to $f = (n-1)/2$ faulty processors.

F. *Safety of Archipelago in Message-Passing*

All results and line numbers in this sub-section refer to Algorithm 4.

Lemma A.9. *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 A.10. *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 A.11 q's B-response to p' must include a $(true, v)$ tuple — a contradiction. If q responded to p' before p, then by Lemma A.11 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 A.12.
- $\mathcal{S}$ contains more than one value v. This is impossible by Lemma A.12.

□

Lemma A.11. *For a fixed i, if a process p sends a B-response $(\mathsf{B\text{-}response}, i, B[i])$ to some process q at time t and p sends a B-response $(\mathsf{B\text{-}response}, i, B[i]')$ 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 A.12. *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 A.11. We have reached a contradiction. □

G. *Archipelago: Proof of correctness in message-passing*

In this section we prove the validity of Archipelago in its message-passing version.

1) Safety: In this section We prove the properties of Validity and Agreement for the OFT-Archipelago algorithm.

Theorem A.13 (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. □

Theorem A.14 (Agreement). *Let p_1 and p_2 be two correct processes. If p_1 and p_2 return $< commit, v_1 >$ and $< commit, v_2 >$ 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. Then this means both p_1 and p_2 saw, during their B-step line 29, at least $f+1$ $\langle true, v_1 \rangle$ 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 $f+1$ processes. Hence, there is at least one of these processes which is correct and 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 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-broadcasted 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. □

2) Leaderless Termination:

Lemma A.15 (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).
Hence all correct processes will A-broadcast v. All A-answers will contain only v and hence all processes will A-return $\langle true, v \rangle$.
Hence all correct processes B-broadcast $\langle true, v \rangle$ and can only receive valid B-responses containing only $\langle true, v \rangle$. Therefore, all correct processes will B-return $\langle commit, v \rangle$. □

Lemma A.16 (Iterative elimination of values)**.** *Eventually only one value can be R-broadcasted or all correct processes commit.*

Proof. Assume we have reached GST. We will study what happens during the B-step and the following R-step. Remember that no two different values can be B-broadcasted 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-broadcasted 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-broadcasted 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-broadcasted 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-broadcasted 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-broadcasted.
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-broadcasted. □

Theorem A.17 (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 commited (otherwise all processes are R-broadcasting the same value and lemma A.15 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-broadcasted 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 A.15). 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 A.15. In both cases, termination happens within $O(n)$ rounds. □

3) Complexity: The detailed proof for the complexity of each step is given for BFT-Archipelago in Section I. Each step requires $O(n^2)$ messages each of length $O(1)$ bits. OFT-Archipelago takes $O(n)$ rounds to terminate, hence the overall complexity is $O(n^3)$ messages and bits.
The space complexity is $O(1)$.

H. BFT-Archipelago: Proof of Correctness

1) Proof of Safety:

Theorem A.18 (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 behavior are impossible under our certificate system.

Lemma A.19. *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-broadcasted at step i in order to obtain a valid certificate) and return a $< false, . >$. □

Lemma A.20. *Let (i, v) be the tuple that is R-broadcasted 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-broadcasted such a $(rank, value)$, which is impossible according to lemma A.19. □

Theorem A.21 (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-broadcasted 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 A.20). 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 A.22. *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 broadcasted 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.

□

2) Proof of Leaderless Termination: In this section we will prove Leaderless Termination for BFT-Archipelago (IV.2). But before that, we need a few lemmas.

Lemma A.23 (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 A.24. *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 synchronisation.*

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-broadcasted, it will be R-returned by all correct awake processes. □

Before we prove IV.2, let us recall the definition of leaderless Termination :
Leaderless Termination In every $\diamond$synchronous-1 execution of BFT-Archipelago, every correct process decides.

For pedagogic purpose, we give here a proof of leaderless termination with the hypothesis that processes only get interrupted for whole rounds. That hypothesis is realistic if we can implement a computational primitive of atomic multi-destination broadcast. that primitive is available in Local Area Networks [26].

Proof. Assume by the time we reach GST, round synchronisation and round R for every correct, uninterrupted process, no process has yet commited.

There are at least $2f + 1$ uninterrupted correct processes performing an R-step and settling on a common $(rank, value)$ from the $2f + 1$ initial correct processes, at least $f + 1$ have stayed uninterrupted and will perform an A-step with said value.
Because of round synchronisation, there are no processes who performed or are performing an A-step at the same time with another value and a valid certificate, then the $f + 1$ correct processes will A-return $\langle true, val\rangle$ and then at least one process is able to B-broadcast $\langle true, val\rangle$ and commit. Therefore, all other processes then have to either adopt that value or commit it at most one rank later, therefore every correct process commits.
Now assume that by the time we reach GST, round synchronisation and round R for every correct, uninterrupted process, some processes have already commited some value v_1 at rank i (and possibly also $i + 1$). Thanks to lemma A.20, we know that no correct process can jump past a rank where a $\langle commit, v_1\rangle$ has been B-broadcasted (let us call that rank j) with a value different than v_1. Correct processes can only be interrupted for a finite time, hence they will eventually go through step B at rank j. Since there are $2f + 1$ correct processes uninterrupted at the time of the B-broadcast of $\langle commit, v_1\rangle$ and $2f + 1$ correct processes uninterrupted at the time of requesting B-answers at rank j, there will be at least $f + 1$ uninterrupted correct processes at both times which will B-answer with $\langle commit, v_1\rangle$. Hence all processes performing a B-step with a rank equal to j will see $\langle commit, v_1\rangle$ and either commit it and decide or adopt v_1. Therefore, at rank $j + 1$ the only valid value to be R-broadcasted is v_1. Thanks to the lemma A.23, we know that all correct processes performing steps R,A and B at rank $j + 1$ will commit. Hence all correct processes commit and terminate. If some process had commited before round synchronization, then all processes will have to either adopt or commit the commited value during their first round B, and therefore will have to commit it during their next step B (since they all propose the same value and due to lemma A.23). □

We now relax the hypothesis of atomic multi-destination broadcast.

Proof. Assume by the time we reach GST, round synchronisation and round R for every correct, uninterrupted process, no process has yet commited. There are at least $2f + 1$ uninterrupted correct processes performing an R-step. Some correct processes may get interrupted while broadcasting their value or their answer to other processes. Therefore, it is possible that all correct processes did not receive the same values during their R-step and will not return the same value. Let us call v_{min} the maximal value that was received by all correct processes, and v_{max} the maximal value that was received by some correct processes only, with $v_{max} > v_{min}$ (otherwise they would all R-return v_{min} and commit it at the next B-step).
Then there are still at least $f + 1$ correct processes taking an A-step. It is possible that all of these $f+1$ correct processes have all R-returned v_{min} and only some processes that will get interrupted during the A-step have R-returned v_{max} or some other value in between. Now two things can happen: either at least one of the processes that will be able to B-broadcast without getting interrupted at the next round sees only v_{min}, and in this case everyone will commit v_{min} because of lemma A.19. Either all of these processes which will be able to B-broadcast entirely next round (there is at least one) will see at least one of the bigger values between v_{min} and v_{max} and will return $\langle false, v\rangle$ with $v > v_{min}$. Then all processes that will advance through step B at rank i will see that higher value and adopt it, hence v_{min} will be abandoned forever (see lemma A.24). Note that processes cannot ignore that value due to lemma A.20.
Therefore, it takes 3 rounds to get rid of one value. By calling the Synchronizer as many times as necessary, we can remove all of the values but one. The lemma A.23 ensures us that then all correct processes will eventually commit that remaining value. □

I. BFT-Archipelago: Complexity

In this section we prove the complexity of BFT-Archipelago in terms of the number of messages exchanged, the amount of computation needed, the communication complexity in bits and the storage complexity.

1) Message complexity: How many messages will be exchanged at each step ?

a) R-step: During Step R, a process:

- broadcasts $3f$ messages
- receives $2f + 1$ messages
- returns $3f$ messages

b) A-step: During Step A, a process:

- broadcasts $3f$ messages
- receives $2f + 1$ messages
- returns $3f$ messages

c) B-step: During Step B, a process:

- broadcasts $3f$ messages
- receives $2f + 1$ messages
- returns $3f$ messages

d) upon delivering (.,j,v):

- One message is received
- $O(n)$ messages are sent

e) Reliability check: During the reliability check, no message is sent nor received; the function is executed locally.

f) Max function: This function is also a calculus function executed locally.

g) Propose: The procedure calls R, A and B. Each of them calls for one "upon delivering ..." each time they broadcast ; thus, the total complexity is $3 * (3f + 2f + 1 + 3f) * (3f + 1) = (24f + 1) * (3f + 1) = O(n^2)$ per rank i that is tried and per process executing the code (which is the same as per all processes executing the code); the overall complexity is $(24f + 1)^2 * (3f + 1) = (576f^2 + 48f + 1) * (3f + 1) = O(n^3)$ per rank i that is tried by a quorum of processes.

2) Computation complexity:

a) max: The complexity of the max function is $O(n)$

b) find $f + 1$ identical values: In order to find if there are at least $f + 1$ identical values in an array of $n > f + 1$ values and extract any that might exist, we can either :

- if we have an order relationship on the values set, we can use an algorithm in time complexity $O(nlog(n))$ and space complexity $O(n)$: just sort the values and check if there is a sequence of at least $f + 1$ identical values.
- Another method has a $O(n)$ time complexity and a $O(n)$ space complexity with dynamical memory allocation, $O(n^2)$ with static memory allocation. We just have to put each value in a stack labeled with the value, and stop when one of these stacks is as large as $f + 1$.

c) Reliability check: The first four lines are $O(n)$ checks, which should be at most reading and comparisons. Then, following the cases:

- Case R: the complexity is the same as a B-step, hence it is $O(n)$ (see below)
- Case A: the max is $O(n)$ and the line 18 is $O(n^2)$, because we have to check for each value if it appears at least f times
- Case B: The complexity is $O(n)$ to run through an A-step

d) Propose & "upon delivering": The complexity here is $O(1)$, with also a call to the reliability-check function.

e) R-step: The complexity is $O(n^2)$, because we have to check for each value if it appears at least f times.

f) A-step: The complexity is $O(n)$ writings plus $O(n)$ operations to find if there are $2f + 1$ identical values (we just have to sort them by value with fusion sort, then scan them looking for a sequence of same values long enough) and at worse a call to max and to reliability-check.

g) B-step: The complexity is $O(n)$ writings plus $O(n)$ operations to find if there are $2f + 1$ identical values.

h) Total complexity: Each proposal of a value v for a given i amounts at a calculus complexity of $O(n^2)$ for one of the processors. The overall calculus made by BFT-Archipelago for a proposal of one i, v by one process is $O(n^3)$.

i) calls to reliability check complexity: For a proposal of one v by one i, each process makes one ($O(n)$) calls to the reliability check function ; Hence, BFT-Archipelago makes $O(n^2)$ calls to that function.

j) Number of signature verification: Each certificate contains $O(n)$ signatures to be verified. During one step, we need to check $O(n^2)$ certificates, hence $O(n^3)$ signatures.

3) Message length:

a) Broadcast: What is the expected message length in bits ? When broadcasting, we broadcast several things :

- A local register (R, A or B). The length of the register is $O(1)$ in terms of messages, each one having its certificate
- *i* and *v*, of fixed length $O(1)$
- a *flag*, of length $1 = O(1)$
- A signature, of length $O(1)$

b) Response: A response contains $O(1)$ messages (one or two to be precise). Each of this messages is certified as having been rightfully broadcasted, but only by the $(2f+1)$ answers that the processes have received. Hence the length of an answer is $O(1)$.

c) Total communication complexity: For all correct processes to go through a whole rank, it takes $O(n) processes * O(n) broadcasts = O(n^2)$ broadcasts of length $O(1)$ bits and for each broadcast as there are $O(n^2)$ responses exchanged of length $O(1)$ bits, for a total amount of $O(n) * (O(n^2)*O(1)+O(n^2)*O(n)*O(1)) = O(n^4)$ bits exchanged. Since, after GST, we need $O(n)$ rounds taken by all processes to decide, the complexity for our algorithm to decide is $O(n^4)$ bits.

4) Storage: We need to store all messages, hence the storage complexity is the same as the bit complexity.

5) Number of rounds: A detailed in the Leaderless Termination proof, it takes $O(n)$ rounds at worst for all correct processes to decide.

J. BFTU-Archipelago: Proof of Correctness

It is possible to modify BFT-Archipelago into BFTU-Archipelago that works without authentication. The idea is to replace the authenticated broadcasts by reliable broadcasts. To this end, each process stores every message ever received or sent in memory but does not send certificates. By "valid certificate", we mean the sum of responses which allow to prove that a certain broadcast is correct (could have been sent by a correct process). Each step requires $O(n^2)$ messages reliably-exchanged, but $O(n)$ messages each sent to all. Each message reliably-exchanged demands $O(n^2)$ messages each of length $O(1)$ bits. BFTU-Archipelago takes $O(n)$ rounds to terminate, hence the overall complexity is $O(n^4)$ messages and bits, with the need to stock all $O(n^4)$ bits in memory.

In this section we prove that BFTU-Archipelago is correct.

1) Safety: In this section we prove the properties of Validity and Agreement for the BFTU-Archipelago algorithm.

Theorem A.25 (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. □

The following lemma is still valid, see the proof in the authenticated case:

Lemma A.26. *There cannot be a $\langle true, v_1\rangle$ and a $\langle true, v_2\rangle$ B-broadcasts with valid certificates and $v_1 \neq v_2$.*

Lemma A.27. *If a correct process B-delivers $2f+1$ B-responses containing only $\langle true, v_1\rangle$ 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 that process p_1 has properly delivered $2f+1$ B-responses $\langle true, v_1\rangle$. Then it means that for each answer, at least $2f+1$ processes have sent a "$ok(\langle true, v_1\rangle)$", amongst which $f+1$ are correct processes which were uninterrupted while sending "$ok(\langle true, v_1\rangle)$". (Hence all correct processes will eventually receive at least $f+1$ "$ok(\langle true, v_1\rangle)$".) Since p_1 received $2f+1$ B-answers, then it means at least $2f+1$ processes, amongst which at least $f+1$ correct processes, have seen the value B-broadcasted by p_1. Therefore those processes have to adopt v_1 and more importantly

Algorithm 7 BFTU-Archipelago in message passing with $n = 3f + 1$

```
Local State:
    $i$, the current rank, initially 0
    $R$, a set of tuples, initially empty
    $A[0, 1, \dots]$ and $B[0, 1, \dots]$, two
        sequences of sets, all initially empty

procedure propose($v$):
    while true do
        $\langle i, v' \rangle \leftarrow$ R-Step($v$)
        $\langle flag, v'' \rangle \leftarrow$ A-Step($i, v'$)
        $\langle contr, val \rangle \leftarrow$ B-Step($flag, i, v''$)
        if $contr =$ commit then return val
        else $i \leftarrow i + 1$, $v \leftarrow val$

procedure R-Step($v$):
    reliable-broadcast($R, i, v$)
    wait until receive valid (R-response, $i$, $R$) from $2f + 1$ processes
    $R \leftarrow R \cup$ {union of all $R$s received in previous line}
    $\langle i', v' \rangle \leftarrow$ max($R$)
    $R \leftarrow$ max($R$)
    return $\langle i', v' \rangle$

upon reliable-delivering $(R, j, v)$ from $p$:
    if reliability check(R,j,v) then
        $R \leftarrow \langle j, v \rangle$
        reliable-send(R-response, $j$, $R$) to all processes

procedure A-Step($i, v$):
    reliable-broadcast($A, i, v$)
    wait until receive valid (A-response, $i$, $A[i]$) from $2f + 1$ processes
    $\mathcal{S} \leftarrow$ union of all $A[i]$s received
    if ($\mathcal{S}$ contains at least 2f+1 A-answers containing only the same value $val$) then
        return $\langle$true, $val\rangle$
    else return $\langle$false, max($\mathcal{S}$)$\rangle$

upon reliable-delivering $(A, j, v)$ from $p$:
    if reliability check(A, j, v) then
        Add $v$ to $A[j]$
        reliable-send(A-response, $j$, $A[j]$) to all processes
    else Ignore message from $p$

procedure B-Step($flag, i, v$):
    reliable-broadcast($B, i, flag, v$)
    wait until receive valid
 (B-response, $i$, $B[i]$) from $2f + 1$ proc.
    $\mathcal{S} \leftarrow$ { all $B[i]$s received }
    if $2f{+}1$ sets in $\mathcal{S}$ contain only $\langle$true, $val\rangle$ for some $val$ then return $\langle$commit, $val\rangle$
    else if 1 set in $\mathcal{S}$ contain some entry $\langle$true, $val\rangle$ then return $\langle$adopt, $val\rangle$
    else return $\langle$adopt, $max(\mathcal{S})\rangle$

upon reliable-delivering $(B, j, flag, v)$ from $p$:
    if reliability check(B, j, v) then
        Add $\langle flag, v \rangle$ to $B[j]$
        reliable-send(B-response, $j$, $B[j]$) to all processes
    else Ignore message from $p$

max($vals$):    ▷ vals is a list of pairs (i,v)
    return $max(vals)$

reliable-broadcast or reliable-send(message):
    send to all "init(message)"

upon receiving init(message):
    send echo(message) to all

upon receiving echo(message):
    send ok(message) to all

upon receiving ok(message):
    wait for 2f+1 ok(message) from different processes
    deliver(message)

Compile Certificate(historic):
List all $2f + 1$ answers received during the previous steps up until the beginning

Reliability check(X,i,v):
    if S contains at least f+1 answers to the reliable-broadcast then return true
    check that certificate contains $2f + 1$ messages
    if detected a Byzantine process then
        return Byzantine user's ID
    if X == R then
        check (i,v) is correct according to B-answers received and step B
    else if X == A then
        check (i,v) is correct according to R-answers received and step R
    else if X == B then
        check (i,flag,v) is correct according to A-answers received and step A
    if All checks pass then
        return true
    else
        broadcast to all the identity of the Byzantine process
```

will show v_1 in their B-responses. Hence any other process will have to wait for $2f{+}1$ correct processes and will therefore have a response from at least one of the processes aware of "$\langle true, v_1 \rangle$". Hence no correct process can move past rank i without adopting or committing v_1.

We now need to prove that a Byzantine process cannot R-broadcast another value than v_1 at rank $i + 1$ or higher. The reason is that in order for such a R-broadcast to be accepted by any correct process p, p must wait until it has delivered all the messages that are part of the certificate of the R-broadcast. As stated before, there is no possibility that $2f + 1$ different processes will deliver B-answers containing only a value different from v_1 at rank i. hence, the R-broadcast of the Byzantine process will be ignored forever. □

Lemma A.28. *Let (i, v) be the tuple that is R-broadcasted 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-broadcasted such a (rank,value), which is impossible according to lemma A.19. Indeed, correct processes will not accept an answer until they have properly delivered the broadcast that serves of justification to that answer. □

Theorem A.29 (Agreement)**.** *Let p_1 and p_2 be two correct processes. If p_1 and p_2 return $< commit, v_1 >$ and $< commit, v_2 >$ 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. Then this means both p_1 and p_2 saw, during their B-step line 41, 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 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 41, at least $2f+1$ sets containing only $\langle true, v_1\rangle$. More precisely, it means that each of those $2f+1$ answers each have been okayed by $2f+1$ processes; hence at least $f+1$ correct uninterrupted processes have relayed this "okay($\langle true, v_1\rangle$)", meaning that no other correct uninterrupted process had yet B-broadcasted 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 cannot pretend to have advanced past rank i without correct processes checking they have properly delivered the messages that led to it, i.e. actually going there while acting like a normal process (cf lemma A.28). 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. □

2) Leaderless Termination: Lemma A.23 is still valid :

Lemma A.30 (Commitment)**.** *If no process R-broadcast anything other than the same (i, v), then all correct processes must output $\langle commit, v\rangle$.*

Lemma A.31 (Eventual delivery)**.** *A message reliably-sent by a correct uninterrupted process p to all processes is delivered by p within 3 rounds.*

Proof. At each round, $2f+1$ correct processes are uninterrupted. Each process has to be uninterrupted for three rounds in order to send an "ok(.)" message. During the first round, $2f+1$ processes emit a "init(.)" message. Then round 2, at least $f+1$ processes emit a "echo()" message. Then round 3, at least $2f+1$ correct processes emit a "deliver(.)" message, having received enough $(f+1)$ "echo(.)". Hence it takes at worst 3 steps to get a reliable-send delivered. □

Lemma A.32 (Iterative elimination of values)**.** *Eventually only one value can be R-broadcasted with proper certificate or all correct processes commit.*

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 A.26, no two different values can be B-broadcasted

with the label $true$ and a valid certificate. Hence only two cases are available: either all values B-broadcasted 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-broadcasted 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 at the time of sending the "ok(.)" message. 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-broadcasted the "init(.)" message 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-broadcasted 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-broadcasted (with a valid certificate). □

Theorem A.33 (Leaderless Termination)**.** *In every* $\diamond$*synchronous*-1 *execution of ByzArchipelago, every correct process decides.*

Proof. Once we have reached the Global Stabilization Time, all correct processes will advance through ranks (i) and steps (R,A,B). This takes a finite time as explained in lemma A.31 (any valid message sent eventually gets delivered by all correct processes). At each rank, they may either B-commit and decide or adopt. If they adopt, thanks to lemma A.32, correct processes will either commit or arrive at a point where there is only one possible value left. When that is the case, thanks to the lemma A.23, all remaining undecided correct processes will decide on this value. Because the time to remove a value is finite and the number of values to be removed is also finite, the algorithm decides in finite time. □

K. BFTU-Archipelago Complexity

a) reliable-send: Sending a message reliably to all takes $O(n^2)$ messages:

- $3f + 1$ ”init(.)” are eventually sent by the sender
- $(3f + 1)^2 = O(n^2)$ ”echo(.)” are then sent ($3f + 1$ per correct process)
- $(3f + 1)^2 = O(n^2)$ ”ok(.)” are sent ($3f + 1$ per correct process)

Sending a message reliably takes 3 rounds of communications.

The detail of the proof is exactly the same a for BFT-Archipelago and can be found in section I.

Each step requires $O(n^2)$ messages reliably-exchanged, but $O(n)$ messages each sent to all. Each message reliably-exchanged demands $O(n^2)$ messages each of length $O(1)$ bits. BFTU-Archipelago takes $O(n)$ rounds to terminate, hence the overall complexity is $O(n^4)$ messages and bits. We also need to stock the content of all messages for the certificates, which hence amounts to (at worst) $O(n^4)$ messages per consensus instance.