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, retain the maximum value 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], [8], [12], [17], [26]–[28], [34], [37], [42], [43]. 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 performance drastically [28], [39], sometimes by two orders of magnitude [26]. Besides, replacing the leader requires a view-change protocol that is so complex that research prototypes often omit it [19] or suffer from 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 [12], [43]. Another is to bypass the leader bottleneck by having multiple proposers [16], [17], [42] before reverting to a weak coordinator to converge. A third one is to tolerate multiple leaders for different consensus instances [26], [34], [37], 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 [15]. 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], [11].

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 (denoted by $\diamond$synchronous-1) system. In a synchronous-1 system, the classical idea of exchanging values in rounds and adopting the maximum one would not work, because the adversary can suspend the process with the maximum value for as long as it wants.

Then we present three leaderless consensus algorithms, each for a specific setting. The first algorithm, called *Archipelago*[1], works in shared memory and builds upon a new variant of the classical adopt-commit object [24] that returns maximum values to help different processes converge towards the same

[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.

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 [14] and can execute optimistically a fast path to terminate in two message exchanges under good conditions. Interestingly, all our algorithms are optimal both in terms of resilience and time complexity.

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. The full proofs are deferred to the companion technical report [3].

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* 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 same configurations and events in the same order.

Synchronous$-k$ execution. We can now define what it means for an execution to be synchronous in shared-memory. Our definition is inspired by the notion of synchrony in a message passing model where there is a bound on the time needed for a message to propagate from one process to another and for the receiver to process this message. In a message passing model, we can divide time into rounds [21] such that, in each round, every process p: (i) sends a message to every other process in the system, and (ii) delivers any message that was sent to p and performs local computation.

To adapt synchrony to the shared memory model, we also assume that processes take steps in rounds. Specifically, in each round, every process p_i (i) performs a write in some $\mathcal{R}_j[i]$ and (ii) collects all the values written in array $\mathcal{R}_j$. 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 [13], each process proposes a value by invoking a propose(v) function and then all processes have to decide on a single value. Consensus is defined by the following three properties. *Validity* states that a value decided was previously proposed. *Agreement* states that no two processes decide different values, and *termination* states that every correct process eventually decides. We say that a consensus algorithm *decides* in an execution α if a propose(v) function call by some process p returns in α.

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. 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 the companion technical report [3] 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 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 [9], [17] and reducing the impact of a single point of failure on performance [8], [42] 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, described in the companion technical report [3], share the same communication complexity as PBFT [14] and DBFT [17], namely $O(n^4)$ bits. Finally, since BFT-Archipelago can be written as an Abstract [7] (see Section V), it is compatible with leader-based consensus instances and inherits an optimal fast path in good executions.

Paxos: a counter example. Consider Algorithm 1, a leader-based algorithm that, when combined with a leader election, corresponds to Paxos [29] in shared memory (or more specifically to Disk Paxos [25] 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 [30]. During the prepare phase, p_i stores its current timestamp value to $R[i]$ (line 7) and either retrieves the value val of R associated with the highest timestamp (line 8), or (if no such value exists) sets val to its own value v. During the propose phase, p_i stores the pair $\langle val, ts\rangle$ to array $R[i]$ (line 11) and examines whether the highest timestamp in R is the one that p_i wrote (line 12). If this is the case, the algorithm decides (line 13), otherwise p_i increases ts and repeats the

	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.

Algorithm 1 Leader-based consensus algorithm

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

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

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

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 [24], 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* [24] 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 [24] 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 [24] and that the main properties of an adopt-commit object remain the same. These modifications are crucial for the leaderless termination of Archipelago.

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 ⊥

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

We defer the correctness proof of Algorithm 2, which is similar to that of an adopt-commit object [24], to the companion technical report [3].

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 [5]. Its write can be implemented by letting each process write to a single-writer multi-reader register whereas its readmax can be implemented by collecting all values written by all processes and taking the maximum. In a synchronous-1 execution, the processes converge towards one value and there is an adopt-commit-max object where all processes propose this exact single value. Then, due to CA-commitment property of the adopt-commit-max object, the adopt-commit-max outputs $\langle \mathsf{commit}, \cdot\rangle$ and Archipelago decides in finite time.

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

Algorithm 3 Archipelago leaderless consensus

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

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

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

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 $C[c'].\mathsf{propose}(v')$ (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 $\langle \mathsf{adopt}, v''\rangle$ 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 [15]. 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 [13], [22], [35]. 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* [35], 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. Archipelago does not satisfy leaderless termination when $n = 2$. The proof that Archipelago is a leaderless consensus algorithm is deferred to the companion technical report [3].

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.

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, ...], a sequence of sets, all initially empty
5:   B[0, 1, ...], a sequence of sets, all initially empty

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

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

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

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

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

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

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

$\diamond$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 [21]. We say that a process p is *suspended* [4] in a round r, if p does not send any messages in r and does not receive any messages sent by other processes in round r.

Adapting Archipelago to message passing. One might be tempted to apply the ABD emulation [6] 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. Although this combination of writes and reads can break atomicity, we show in the companion technical report [3] that it does not violate safety during asynchronous periods.

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 $n \leq 3f$ [33], 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 of presentation, we also assume authentication. The alternative unauthenticated variant and the proof that the result generalizes to the $\diamond$synchronous$-k$ model, where $k \leq f$ and $f - k$ processes can be Byzantine, are deferred to the companion technical report [3].

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

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

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

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

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 the companion technical report [3].

V. Discussion and Complexity Analysis

Termination. In addition to leaderless termination (Theorem IV.1), Archipelago satisfies termination for $n \geq 3$, meaning that in an eventually synchronous [13] execution, every correct process eventually decides. In such an execution, Archipelago needs at most 5 rounds, after the global stabilization time [21] and round synchronization (i.e., all processes start and end a round at the same time).

Fast path of BFT-Archipelago. The common-case performance of BFT-Archipelago can be improved by executing an optimistic fast path under favorable conditions (e.g., synchrony, no failures, no contention), and falling back to a robust path when these conditions are not met. This can be achieved with the Abstract scheme [7] as it allows chaining multiple BFT protocols, called Abstract instances, that can abort and fall back to the next instance. In particular, the *Backup* wrapper allows any full BFT protocol to become an Abstract instance.

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, ...] and B[0, 1, ...], two
5:       sequences of sets, all initially empty
6:     C a sequence of broadcasts ID with the
7:       number of answers they have received

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

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

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

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

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

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

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

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

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

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 [21] and time-optimal [20], [23]. BFT-Archipelago is also competitive with PBFT [14] and DBFT [17], having the same communication complexity. The detailed proof is deferred to the companion technical report [3].

VI. Related Work

Given the notorious impact of a leader on consensus performance [1], [8], [9], [12], [17], [26]–[28], [34], [37], [42], [43], it is surprising that the leaderless concept has never been precised.

The leader has become a limitation to scale consensus to large blockchain networks. Crain et al. [17] consider the Democratic BFT (DBFT) consensus algorithm as leaderless. DBFT is a multivalue consensus algorithm at the heart of the Red Belly Blockchain [18] whose n proposers bypass the leader bottleneck. It spawns n concurrent binary consensus instances, each relying on a weak coordinator to help converge when many correct processes propose distinct values. Although DBFT could use n different weak coordinators, its binary consensus is not leaderless according to our definition.

In a brief announcement [31], 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 [32], Lamport explains that during a period of asynchrony, if the virtual leader election fails, then the consensus algorithm may not progress [31]. Our adopt-commit-max object of Archipelago allows processes to converge towards a unique value, hence sharing similarities with the proposal of some virtual leader. Yet, neither a leaderless definition nor a virtual leader specification were given by Lamport.

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

Interestingly, SMR algorithms that rely on multiple leaders (e.g., Mencius [34], RBFT [8]) do not necessarily rely on a leaderless consensus algorithm.

Moraru et al. [37] used multiple "command leaders" in EPaxos. Each leader tries to commit one command. When commands have dependencies only one of the leaders can get its command committed at a time, as if there were successive leader-based consensus instances. If a leader fails

after receiving a positive 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.

Recently, some errors [40], [41] were found in both randomized [36], [38] and multi-leader consensus algorithms [37], indicating that getting rid of the leader is error prone.

VII. Concluding remarks

Our definition of leaderless is general. 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 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, Antoine Desjardins, Vincent Gramoli, Rachid Guerraoui, and Igor Zablotchi. Leaderless consensus. Technical report, EPFL, 2021. URL: https://infoscience.epfl.ch/record/282657?&ln=en.

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

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

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

[7] 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.

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

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

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

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

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

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

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

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

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

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

[18] Tyler Crain, Christopher Natoli, and Vincent Gramoli. Red Belly: a secure, fair and scalable open blockchain. In *S&P*, 2021.

[19] 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.

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

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

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

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

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

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

[26] 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.

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

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

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

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

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

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

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

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

[35] 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.

[36] Andrew Miller, Yu Xia, Kyle Croman, Elaine Shi, and Dawn Song. The honey badger of BFT protocols. In *ACM CCS*, pages 31–42, 2016.

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

[38] Achour Mostefaoui, Hamouma Moumen, and Michel Raynal. Signature-free asynchronous byzantine consensus with $t < n/3$ and $o(n^2)$ messages. In *PODC*, 2014.

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

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

[41] Pierre Tholoniat and Vincent Gramoli. Formally verifying blockchain Byzantine fault tolerance. In *FRIDA*, 2019. Available at https://arxiv.org/pdf/1909.07453.pdf.

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

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