Agreeing within a few writes

Zohir Bouzid, Pierre Sutra, Corentin Travers

Theoretical Computer Science ••• (••••) •••–•••

Contents lists available at ScienceDirect

Theoretical Computer Science

www.elsevier.com/locate/tcs

Agreeing within a few writes ☆

Zohir Bouzid, Pierre Sutra [a,*], Corentin Travers [b]

[a] *Télécom SudParis, France*
[b] *Université Bordeaux 1 - LaBRI, France*

ARTICLE INFO

Article history:
Received 14 July 2021
Received in revised form 19 April 2022
Accepted 21 April 2022
Available online xxxx
Communicated by L. Christoph

Keywords:
Anonymity
Asynchronous shared memory
Agreement
Consensus
Adopt-commit
Conflict detector
Solo fast algorithms
Homonym processes

ABSTRACT

The notion of adopt-commit object [29] is of pivotal importance in understanding the consensus problem. This object models an attempt of the processes to agree on some common value, and precisely captures the cost of the fast path a process takes during a solo run. In this paper, we address the problem of implementing an adopt-commit object in the shared memory model in the minimal number of write operations. We consider that the number of processes (n), their identities (c), as well as the size of the input set (m) may all vary.

Our first contribution is an algorithm that executes three write operations, a value we show optimal in the general case. We also prove that this number reduces to two when either m is known and bounded, or n identities are available. In the corner case where $n = 2$, and either $c = 2$ or the input set is finite, a single write suffices.

Further, we introduce Janus, an elegant adopt-commit implementation that executes $O(n)$ shared memory operations, including $O(\sqrt{n})$ writes. Building upon Janus, we explain how to design an adopt-commit object that executes $O(\sqrt{n-c+1})$ write operations. We prove that this last value is tight when the number of registers in use is bounded and m is unknown.

1. Introduction

Reaching agreement is a fundamental problem in distributed computing. Roughly speaking, it requires a set of processes to decide upon some subset of their input values. Consensus [28], k-set agreement [20] and total order broadcast [18] all belong to this class of problems. These tasks are the key building blocks of many of the systems at work in today modern computing infrastructures.

It is well-known that as soon as one of the processes may fail-stop, agreeing necessitates additional mechanisms than asynchronous read/write shared memory [11,28,40,41,51], namely either the enforcement of synchrony assumptions [19], or the use of hardware synchronization primitives [39]. To sidestep this problem, several researchers starting from Lamport [44] have proposed to execute a tentative fast path solely composed of read/write operations before calling a more expensive mechanism. Exploiting good runs is the prolific idea behind the notions of splitter [48], conflict detector [5], and solo-fast algorithms [7] that execute only read/write operations in the absence of contention.

In this context, a central question is to measure precisely the cost of this fast path. Several parameters are of interest here, including the number of processes (n), how many identities are available in the system ($c \leq n$), and the total number

☆ A preliminary version of this work appeared in the proceedings of the 15th International Conference On Principles Of Distributed Systems (OPODIS'11) under the title "Anonymous Agreement: The Janus Algorithm" by the same authors.

* Corresponding author.
E-mail address: pierre.sutra@telecom-sudparis.eu (P. Sutra).

https://doi.org/10.1016/j.tcs.2022.04.030
0304-3975/

Table 1
The solo-write complexity of adopt-commit objects *($n \geq 2$ is the number of processes, $c \in [1, n]$ the number of their identities, and $m \geq 2$ the size of the input set).*

	Solo-write complexity	Space bounded	
$(n = 2) \wedge ((c = 2) \vee (m < \infty))$	1	yes	Section 4
$(c = n) \vee (m < \infty)$	2	yes	
Any n, c and m.	3	no	
	$\Theta(\sqrt{n - c + 1})$	yes	Section 6

of input values (m). For instance, we would like to understand the benefits of having a distinct identity for each process ($c = n$), in comparison to the case where processes are *anonymous* ($c = 1$).

In this paper, we focus our attention on processing the fewest writes in the fast path. Our rationale behind this choice is the key observation that writing is generally more expensive than reading. For instance, in the current cache-coherent architectures, each write may invalidate the remote caches, increasing the miss rate and deteriorating performance [38,52]. Upcoming memory technologies are expected to widen this gap [32].

Contributions. The notion of adopt-commit object translates an attempt of the processes to agree on a common input value [29,54]. When implemented with solely read/write operations, it precisely captures the fast path of an agreement task. With more details, as consensus fulfills the specification of adopt-commit, any lower bound result on the complexity of adopt-commit objects directly applies to consensus as well [5]. This paper establishes several tight results on the write complexity of adopt-commit objects. Table 1 summarizes our contributions.

- We first show that two writes are necessary and sufficient in the case where the input set is bounded or n identities are available. Moreover, if the system consists of only two processes and the same conditions hold, a single write is possible.
- Then, we present an adopt-commit object that executes three write operations, a value we prove to be optimal in the general case.
- Our previous algorithm has a step complexity of $O(m)$ operations and uses $O(m)$ registers. As the set of proposed values can be arbitrarily large, such a solution might not always be practical. To sidestep this problem, we propose Janus, an adopt-commit algorithm that executes $O(n)$ shared memory operations. Janus accommodates with any set of input values and any number of identities. It exhibits a write complexity of $O(\sqrt{n})$ operations. When processes do not have identities, both values are asymptotically tight.
- Our last result combines the two previous solutions to leverage efficiently the presence of process identities. In detail, we design a solution that first solves conflicts between processes having the same identity, then moves to an agreement among processes with distinct identities. A process executing this algorithm writes to $O(\sqrt{n - c + 1})$ registers. We also prove a lower bound of $\Omega(min(\frac{\log(m)}{\log(\log(m))}, \sqrt{n - c + 1}))$ write operations for adopt-commit objects that employ a bounded amount of registers. This shows that our algorithm is (asymptotically) optimal in the case where the input set is not bounded.

Roadmap. We survey the literature in Section 2. Section 3 presents our model of distributed system and the related assumptions. We detail three algorithms with constant solo-write complexity in Section 4. The Janus algorithm is introduced in Section 5. Section 6 details the case of systems where multiple distinct identities co-exist. Section 7 closes this paper.

2. Related work

Anonymous and homonymous systems. In the common case, the processes that compose a distributed system have unique identities and may differentiate one from another. In an *anonymous* system, they instead execute the very same code. When provided with the same input, they are consequently indistinguishable. Anonymity is sometimes unavoidable in practice [26], as for instance with tiny devices [3], or file sharing applications [21]. Dealing with anonymity adds a new challenging dimension to distributed computing. It questions the benefits of having identities while relieving from the burden of managing them.

Under partial anonymity, some processes may share the same identifier. This notion was first introduced by Yamashita and Kameda [53] in the context of the leader election problem. The term *homonyms* was coined by Delporte-Gallet et al. [23] in their study of byzantine systems.

Multiple papers [5,6,14,22,36,37,50] try to circumvent the computational power of anonymous systems. Guerraoui and Ruppert [37] study shared memory distributed systems in the presence of both anonymity and failures. They propose several constructions for fundamental abstractions such as timestamping, snapshots and obstruction-free consensus. Attiya et al. [6] characterize failure-free tasks that are solvable using registers when the processes in the system are unknown. Using bivalence and covering arguments, they prove that consensus in such environments requires more than $\Omega(\log n)$ atomic registers, and at least $\Omega(\log n)$ total work.

Aspnes and Ellen [5] prove that in the presence of anonymous processes the solo time complexity of adopt-commit objects belongs to $\Theta(min(\frac{\log m}{\log\log m}), n)$. This paper pursues this line of research, showing several tight results on the write complexity of adopt-commit objects. Our results not only depend on the number of proposals (m) and of processes (n), but also on the number of identities (c) available in the system.

Anonymous memory extend anonymity to the case of registers. In this distributed computing model, processes do not share a common knowledge of the registers. This means that a shared register named x by a process might be called y by another process. Some recent works (e.g., [33]) study how to deanonymize memory, for instance by electing a distinguished leader process to execute such a task.

The consensus problem. Consensus is a fundamental abstraction in fault-tolerant distributed computing. Informally, the processes, each starting with a private value, are required to agree on one of these initial values. For shared memory systems, it is well known that asynchronous fault-tolerant consensus is impossible as soon as at least one process may fail by crashing [46]. Trivially, consensus is thus impossible in homonymous, asynchronous and failure-prone shared memory systems. The same impossibility holds for message passing asynchronous systems [28].

Since the publication of the above result, several approaches have been identified to overcome this impossibility, including randomization (e.g., [9]), strengthening the model with timing assumptions (e.g., [25]) or failure detectors (e.g., [18]) and strong synchronization primitives [39]. For anonymous systems, randomization [15], failure detectors [10,12,22], as well as additional synchrony assumptions [22] have been investigated to solve consensus.

Most consensus algorithms employ a round-based pattern to reach an agreement among the processes. In a nutshell, this pattern works as follows: When it enters a round r, a process p fetches the values which were proposed at round $r-1$. Process p picks one of these values (say v) as its proposal for round r. Then, p decides v if either *(1)* it reaches alone round r [18,29,35,45], or *(2)* no value other than v was proposed at rounds $r-1$ and $r-2$ [22,37]. In the converse case, p moves to the next round.

On the write complexity of consensus. For each process, the Paxos consensus algorithm [45] employs two single-writer multiple-readers registers: one to indicate the current round of the process, and another for its associated proposal. The solo path of Paxos contains two write operations [35]. We show in Section 4 that this value cannot be improved.

In the binary consensus problem, only 0 and 1 can be proposed. Abrahamson [1] studies this problem in the probabilistic-write model, where processes have distinct identities to label the registers. When processes are anonymous, Attiya et al. [8] show that $\Omega(\log(n))$ steps are necessary in solo runs. Delporte-Gallet and Fauconnier [22] propose an algorithm that accommodates any number of anonymous processes and executes two write operations in a solo run. This construction relies on the notion of weak (add-only) set, and is similar to the "two-track race" algorithm depicted in [37]. As we shall see hereafter, a single write operation is achievable when $n=2$, but in the general case, anonymous processes need at least two write operations to reach an agreement over two values.

By translating an optimal binary consensus algorithm into a multi-valued one (see [49] for this reduction), we obtain a general solution that executes $O(\log m)$ write operations. The algorithm of [47] accesses a splitter object then a decision register in its fast path. As a splitter object requires two writes operations when processes have distinct identities, this approach executes a total of three writes. Theorem 3 in Section 4.1 proves that this value is optimal for the general case, that is when we have no assumption on m, n or c. In Section 4.2, we present the first algorithm that matches this lower bound.

Aspnes and Ellen [5] propose two asymptotically time-optimal implementations of adopt-commit objects, one that requires the knowledge of m and another which needs the value of n. In a solo execution, the former solution writes to $O(\frac{\log m}{\log\log m})$ registers and the latter to $O(n)$. This last solution is algorithmically close to the leaky repository introduced by Delporte-Gallet et al. [24].

Building upon the above results, Capdevielle et al. [16] studies the solo-write complexity of k-set-agreement. For consensus ($k=1$), and either space-bounded or input-oblivious algorithms, the authors prove that the solo-write complexity belongs to $\Theta(min(\frac{\log(m)}{\log(\log(m))}, \sqrt{n}))$.[1] To state this tight bound, the authors use a fast path consisting of an algorithmic similar to [13,31], before calling a compare-and-swap object. In Section 6, we generalize this result to the case of homonymous systems, obtaining the tight bound $\Theta(min(\frac{\log(m)}{\log(\log(m))}, \sqrt{n-c+1}))$. The construction that matches this bound is built above the Janus algorithm, an efficient anonymous adopt-commit algorithm that we cover in Section 5. This paper also considers the case of countable yet unbounded input sets. Section 4 shows that, quite surprisingly, it is possible to achieve a constant solo-write complexity in this situation.

3. Preliminaries

This section presents our system model then the two distributed tasks we are mostly interested with, namely consensus and adopt-commit.

[1] An algorithm is input oblivious when it always accesses the same sequence of base objects (here the registers) during a solo run whatever is its input.

3.1. Model

In this paper, we focus on the usual shared memory model, where processes are asynchronous, crash-prone, and communicate with the help of linearizable registers. We recall the elements of this model below.

Shared memory model. We assume a system Π of $n \geq 2$ deterministic processes. Processes are aware that $c \in [1, n]$ identities are used in the system and that such identities range in $[1, c]$. Processes that share the same identity are said to be *homonyms* [23]. If a single identity is available in the system, processes are *anonymous* and in which case they follow the exact same code. In the opposite case ($c = n$), we say that processes are *onymous*. For some process p, a *clone of* p is a process homonymous to p that executes in lock-step with p until a certain point [27]. Notice that processes might know initially the value of n. This is mentioned where appropriate.

Processes communicate via a shared memory of multi-writer multi-reader (MWMR) linearizable registers. During a computing step, a process reads/writes either a shared, or local, register. A step is a tuple (p, o) where p is the process taking the step, and o its operation. In our algorithms, we use upper-case identifiers for shared registers and lower-case identifiers for local ones. The universal set of steps together with the concatenation operator "." and the empty execution ϵ forms the infinitary free monoid. This monoid $(E, .)$ consists in all the finite (or infinite) sequences of steps [30]. Every sequence of steps in this monoid is called a run, or an *execution*. We say that an execution is *solo* when a single process takes steps in it. An execution is *admissible* for some distributed algorithm $\mathcal{A}$ when it applies to some initial state of $\mathcal{A}$. We note $\sqsubseteq$ the partial order induced by ".", i.e., $\lambda \sqsubseteq \lambda'$ means that λ prefixes λ'.

On the course of an execution, a process may unexpectedly halt, or *crash*, and in such case it ceases taking steps. A process that does not crash is said to be *correct*. We consider that up to $n - 1$ processes may fail during an execution. An execution is *fair* when all the correct processes take an unbounded amount of steps. All the fair executions that are admissible for some algorithm $\mathcal{A}$ define *the executions of* $\mathcal{A}$.

Time complexity. We measure the time complexity of a distributed algorithm during solo executions [2,16]. More precisely, the *solo-step complexity* is the worst case number of non-local steps during solo admissible executions, and the *solo-write complexity* is the worst-case number of non-local *write* steps in these executions. For some distributed algorithm $\mathcal{A}$, $\mathit{TIME}(\mathcal{A})$ and $\mathit{WTIME}(\mathcal{A})$ are respectively the solo-step and solo-write complexity of $\mathcal{A}$.

The solo-write complexity is our main complexity measure. The rationale behind this choice is threefold. First, since there is no deterministic wait-free solution to consensus in an asynchronous read/write shared memory system [40], the worst-case number of steps is arbitrary large. As a consequence, we need to consider "good runs". Second, it is observed in practice that processes rarely contend in parallel systems [43]. As a consequence, solo executions are the common case when calling a one-shot task such as consensus. Thirdly, there are performance benefits in executing (inexpensive) read and write operations in *the fast path*, and resorting to a strong read-modify-write primitive only if contention occurs [47]. This argument is especially true for read operations, as reads are commonly faster than writes (e.g., with caching).

Distributed task. A distributed task T is defined with a set $\mathcal{I}$ of input n-vectors, a set $\mathcal{O}$ of output n-vectors and a map Δ from $\mathcal{I}$ to $2^{\mathcal{O}}$. If the input value of a process p in $I \in \mathcal{I}$ equals $\perp$, then p does not *participate* to the input vector I. Similarly if $O[p]$ equals $\perp$, p does not *decide* in O. For any distributed task $T = (\Delta, \mathcal{I}, \mathcal{O})$, we require that (i) a process may not decide $((\forall p : O'[p] \in \{O[p], \perp\} \wedge (I, O) \in \Delta) \rightarrow (I, O') \in \Delta)$, as well as (ii) a process that does not participate, does not decide $((I[p] = \perp \wedge (I, O) \in \Delta) \rightarrow O[p] = \perp)$.

Let *Values* be the universal set of (non-$\perp$) values taken by the input and output n-vectors. As we consider distributed deterministic Turing machines, *Values* is recursively enumerable. We note $m \geq 2$ the cardinality of *Values*, that is either some natural, or $\aleph_0$, the cardinality of $\mathbb{N}$. Hereafter, and without lack of generality, we shall be considering that $\mathit{Values} = \{0, 1, \ldots\}$.

An algorithm $\mathcal{A}$ solves a distributed task T when starting from some input n-vector $u \in \mathcal{I}$, it constructs a valid output n-vector $v \in \Delta(u)$. In this paper, we restrict our attention to *wait-free* solutions [42]. Such solutions ensure that in every execution a correct process outputs some value after a bounded amount of steps.

Reduction. We say that a distributed task $\mathcal{T}$ *reduces* to task $\mathcal{T}'$ when from some algorithmic solution of $\mathcal{T}'$, we may construct a solution to $\mathcal{T}$ that differs only by a constant number of write steps *during solo executions*. In which case, we shall note it $T < T'$. If both $\mathcal{T} < \mathcal{T}'$ and $\mathcal{T}' < \mathcal{T}$ hold, tasks $\mathcal{T}$ and $\mathcal{T}'$ are equivalent from the perspective of the solo-write complexity, denoted hereafter $\mathcal{T} \equiv \mathcal{T}'$.

3.2. Distributed agreement

In what follows, we define the distributed tasks we are interested with. These tasks might also be specified as concurrent objects using interval-linearizability [17].

Consensus. Consensus (*CONS*) is a distributed task defined by the unique operation $\mathit{propose}(u)$. A process p that invokes $\mathit{propose}(u)$ is *proposing* u to consensus. When $\mathit{propose}(u)$ returns a value v to p, we say that p *decides* v. Consensus requires

that in every run: (Agreement) Two processes never decide different values; and (Validity) If a process decides some value v, then v is proposed before.

Adopt-commit. The usual approach to solve consensus is to execute successive rounds during which processes try to agree on some of the proposed values (see, e.g. [18,25,45]). The notion of adopt-commit (*AC*) object [29] models an attempt of the processes to agree. More precisely, starting from $v \in Values$ a process that executes $adoptCommit(v)$ should return a response of the form (b, v'), where $b \in \{\texttt{commit}, \texttt{adopt}\}$ and $v' \in Values$. In addition, the following properties must hold: (Validity) If $(-, v)$ is returned, then some process previously invoked $adoptCommit(v)$; (Agreement) If $(\texttt{commit}, v)$ is returned, then every decision has the form $(-, v)$; and (Convergence) If every process proposes the same value v, then $(\texttt{commit}, v)$ is the only possible decision. In particular, if a process executes $adoptCommit(v)$ solo, it must return $(\texttt{commit}, v)$.

Reducing consensus to adopt-commit. We can solve consensus by successively entering adopt-commit objects, proposing to the next object the value that was returned (adopted or committed) by the previous one [54]. The *alpha* of consensus [34] and the notion of *ratifier* [4] also capture this algorithmic idea. We recall such a construction in Algorithm 1.

Algorithm 1 Reducing Consensus to Adopt-Commit – code at process p.

```
1: Shared Variables:
2:     R                                          // An unbounded array of adopt-commit objects
3:
4: Procedure propose(u)
5:     i ← 0
6:     while true do
7:         (f, u) ← R[i].adoptCommit(u)
8:         if f = commit then
9:             return u
10:        i ← i + 1
```

In detail, Algorithm 1 employs an unbounded array of adopt-commit tasks R. When a process p proposes a value to consensus, it enters the first task $R[0]$. If the task $R[0]$ returns a committed value, process p decides it. Otherwise, process p adopts this value as its new proposal, and proposes it to $R[1]$, etc.

In Algorithm 1, the implementations of adopt-commit objects may vary between two entries of array R. For instance, $R[0]$ can rely only on registers while $R[1]$ uses compare-and-swap.

From the above construction, we know that consensus reduces to the adopt-commit abstraction. The reduction also holds in the converse way: when a process execute $adoptCommit(u)$, we simply propose u to consensus and return $(\texttt{commit}, v)$, where v is the value decided in consensus. This yields to the following theorem:

Theorem 1 *([29]). $WTIME(CONS) = WTIME(AC)$.*

At the light of this result, we may simply focus on the adopt-commit abstraction to solve efficiently consensus. This is the approach we follow in the reminder of this paper. With more details, we first explain how to solve adopt-commit in three write operations. Then, we prove that this value is optimal when there is no storage constraint. Further, we depict Janus, a solution that executes $\Theta(n)$ solo work and $\Theta(\sqrt{n})$ write operations, while using $O(\sqrt{n})$ registers. Combining the previous algorithms, we then detail how to leverage the presence of identities to solve the problem in $\Theta(\sqrt{n-c+1})$ write operations.

4. Implementing adopt-commit with optimal write complexity

In what follows, we prove that adopt-commit requires three write operations and present a matching algorithm. We also consider two corner cases of interest that allow to reach a faster agreement. First, in the case where $(n = 2) \wedge (c = 2 \vee m < \aleph_0)$ we show that writing to a single register is possible. Second, when $(c = n \vee m < \aleph_0)$, we argue that two writes are necessary and sufficient.

4.1. Lower bound results

For starters, we observe that if solely the values 0 and 1 are proposed, we can implement adopt-commit in a few writes. For instance, the algorithm of Aspnes and Ellen [5] requires two writes in that case. As we shall see shortly, we may even attain a single write operation under those very circumstances.

Additional notations. Given some set $P \subseteq \Pi$ and two executions λ and λ', we say that λ is *indistinguishable* from λ' to P, written $\lambda \overset{P}{\sim} \lambda'$, when every process $p \in P$ executes the same steps during the two executions. At the light of this definition, if λ is admissible for some algorithm $\mathcal{A}$ and $\lambda \overset{\Pi}{\sim} \lambda'$, then λ' is also admissible for $\mathcal{A}$.

We note $\lambda \vdash (\texttt{commit}, u)$ (respectively, $\lambda \vdash (\texttt{adopt}, u)$), when some process commits (resp. adopts) value u during the execution λ. In what follows, the notation p_u refers to *some* process starting with input u, and σ_u is its associated solo execution. The sets $ws(u)$ and $rs(u)$ are respectively the registers read and written during σ_u.

From the specification of an adopt-commit object, every process that invokes *adoptCommit*(u) solo should return $(\texttt{commit}, u)$. The indistinguishably result below is a straightforward consequence of this observation.

Proposition 1. *For any two distinct input values u and v, $rs(u) \cap ws(v) \neq \varnothing$.*

Proof. By contradiction. If $rs(u) \cap ws(v) = \varnothing$, then $\sigma_v.\sigma_u \overset{p_u}{\sim} \sigma_u$. Thus the execution $\sigma_v.\sigma_u$ is admissible. However, the conjunction of $\sigma_u \vdash (\texttt{commit}, u)$ and $\sigma_v.\sigma_u \vdash (\texttt{commit}, v)$ leads to the contradicting fact that $\sigma_v.\sigma_u$ does not satisfy agreement. □

Below, we establish that every adopt-commit implementation executes at least two write operations. We also show that this boils down to one in the case where the system consists of a pair of processes with distinct identities. To some extent, our proof is a formal treatment of the intuition given by Lamport [44] in his seminal work on fast mutual exclusion algorithms.

Theorem 2. *If $(n = 2) \wedge ((c = 2) \vee (m < \aleph_0))$ then $WTIME(AC) = 1$; otherwise $WTIME(AC) \geq 2$.*

Proof. Let us consider a system made up of two processes $\Pi = \{p_0, p_1\}$. The fact that we need at least one write follows from Proposition 1. Then, let us consider that one of the following assumptions holds.

($m < \aleph_0$) To obtain a matching algorithm when *Values* is bounded, we map each value u to a register $R[u]$. Initially, all the registers contain $\perp \notin Values$. A process that proposes value u writes u to $R[u]$. Then, it reads all the registers $R[v]$ with $v \neq u$. If some value v appears in a register, the process adopts it. (Notice that since $n = 2$ at most a single value v may be in that case.) Otherwise, the process commits value u.

($c = 2$) In the case where $c = 2$, we employ the exact same idea. The process identities are used in lieu of the values, a process with identity i writing initially its proposal to register $R[i]$.

Next, we prove the second part of the theorem, that is if $(n > 2)$ or $(c = 1 \wedge m = \aleph_0)$ holds, we need at least two writes to the registers. We proceed by contradiction, assuming that at most one write is executed.

($n > 2$) Fix two values u and v. Execution σ_u (respectively, σ_v) is of the form $\phi_u^0.w_u^0.\phi_u^1$ (resp., $\phi_v^0.w_v^0.\phi_v^1$), where ϕ_u^0 and ϕ_u^1 (resp., ϕ_v^0 and ϕ_v^1) are sequences of read operations, and w_u^0 (resp., w_v^0) is the unique write operation. (Case $ws(u) = ws(v)$.) Let us note λ the execution $\phi_v^0\sigma_u.w_v^0.\phi_v^1$. This execution is clearly admissible. In addition, we have $\lambda \overset{p_v}{\sim} \sigma_v$ and $\lambda \overset{p_u}{\sim} \sigma_u$. This leads to $\lambda \vdash (\texttt{commit}, u)$ and $\lambda \vdash (\texttt{commit}, v)$; a contradiction. (Case $ws(u) \neq ws(v)$.) Defining $\lambda = \phi_u^0.\sigma_v.w_u^0$ and $\lambda' = \phi_v^0.\sigma_u.w_v^0$, we know that $\lambda \vdash (\texttt{commit}, v)$, while $\lambda' \vdash (\texttt{commit}, u)$. Let us then observe that the registers end-up in the same state in λ and λ'. Hence, for some process $p \notin \{p_u, p_v\}$ we have $\lambda \overset{p}{\sim} \lambda'$; a contradiction.

($c = 1 \wedge m = \aleph_0$) Recall that since $c = 1$, the input value determines the solo execution. Fix some value w. The run σ_w is bounded. On the other hand for any value $u \in Values$, Proposition 1 tells us that $rs(w) \cap ws(u) \neq \varnothing$. Hence, by the pigeonhole principle, there exists two values u and v such that $ws(u) = ws(v)$ holds. As a consequence, we may apply the same reasoning as above. □

We now focus our attention to the case where *Values* is unbounded (or unknown to the processes) and prove a larger lower bound. More precisely, we show that every adopt-commit solution executes at least three write operations in the general case.

Proposition 2. *If $n > 2$, then there do not exist u and v such that* (i) *p_u and p_v are distinct, and* (ii) *σ_u and σ_v write only to two registers in the same order.*

Proof. By contradiction. First of all, let us observe that for some value u, we may write $\sigma_u = \phi_u^0.w_u^0.\phi_u^1.w_u^1.\phi_u^2$. Then, fix u, v matching the premises of the proposition. We define $\lambda_u = \phi_v^0.\sigma_u.w_v^0$ and $\lambda_v = \phi_v^0.(\phi_u^0.w_u^0.\phi_u^1).(w_v^0.\phi_v^1.w_v^1.\phi_v^2).w_u^1$ Both of these runs are clearly admissible. Since $\sigma_u \overset{p_u}{\sim} \lambda_u$, we have that $\lambda_u \vdash (\texttt{commit}, u)$. It follows that for any λ, with $\lambda_u \sqsubseteq \lambda$, $\lambda \vdash (-, u)$ holds. On the other hand, $\sigma_v \overset{p_v}{\sim} \lambda_v$, from which it follows $\lambda_v \vdash (\texttt{commit}, v)$. As a consequence, for any λ, with $\lambda_v \sqsubseteq \lambda$, $\lambda \vdash (-, v)$ holds. Then, choose some process $q \notin \{p_u, p_v\}$; this is possible as $n > 2$. We observe that $\lambda_u \overset{q}{\sim} \lambda_v$. A contradiction. □

From Proposition 2, we deduce the following result:

Proposition 3. *Consider that $n > 2$ and $c < n$, and let $\mathcal{A}$ be some implementation of adopt-commit with $WTIME(A) = 2$. For any subset $U \subseteq Values$, if $|U| = \aleph_0$ then $|\bigcup_{u \in U} ws(u)| = \aleph_0$.*

Proof. Consider for the sake of contradiction that $\bigcup_{u \in U} ws(u)$ is bounded. Applying the pigeonhole principle to U, we may deduce that for two values u and v, σ_u and σ_v write to the two same registers in the same order. If $p_u = p_v$, since $c < n$ we may replace p_u with a clone. As a consequence, the premises of Proposition 2 holds. A contradiction. □

We are now ready to prove that three writes are necessary in the general case.

Theorem 3. *If $n > 2$, $c < n$ and $|Values| = \aleph_0$ then $WTIME(AC) > 2$.*

Proof. By contradiction. Choose some $u \in Values$. Since p_u returns after a finite number of steps, $rs(u)$ is bounded. From Proposition 1, for every $v \in Values \setminus \{u\}$, $rs(u) \cap ws(v) \neq \varnothing$. As a consequence, we can apply the pigeonhole principle to $rs(u)$ and $\{ws(v) : v \in Values \wedge v \neq u\}$. It follows that for there exists $R \in rs(u)$ and some unbounded set $U \subseteq Values$ with $\{R\} \subseteq \bigcap_{v \in U} ws(v)$.

Consider $\bigcup_{v \in U} ws(v)$, the set of registers written in $(\sigma_v)_{v \in U}$. Since U is unbounded, we deduce from Proposition 3 that $(\sigma_v)_{v \in U}$ write to an unbounded amount of registers that are not R. We may thus define $U' \subseteq U$ unbounded such that:

$$\forall v, w \in U' : ws(v) \cap ws(w) = \{R\} \tag{1}$$

From which we construct a series $(u_k)_{k \in \mathbb{N}} \subseteq U'$ satisfying:

$$\forall k \in \mathbb{N} : rs(u_k) \cap \Big(\bigcup_{k' > k} ws(u_{k'})\Big) \subseteq \{R\} \tag{2}$$

This construction goes as follows: Pick some $u_0 \in U'$. As $rs(u_0)$ is bounded, U' is unbounded and (1) holds, we may find $U_0 \subseteq U' \setminus \{u_0\}$ unbounded satisfying (2). Repeat the previous steps starting from some $u_1 \in U_0$.

Our next step is to show that for every $k \geq 0$, the register written first in σ_{u_k}, i.e., $w^0_{u_k}$ following the notation introduced above, is not register R. For the sake of contradiction, assume that this holds for some u_k and consider $\lambda = \phi^0_{u_k}.\sigma_{u_{k+1}}.w^0_{u_k}$. Applying (2), $\lambda \overset{p_{u_k}}{\sim} \phi^0_{u_k}.w^0_{u_k}$ and thus execution $\lambda' = \lambda.\phi^1_{u_k}.w^1_{u_k}.\phi^2_{u_k}$ is admissible. However, $\lambda \vdash (\texttt{commit}, u_{k+1})$, $\lambda \sqsubseteq \lambda'$ and $\lambda' \vdash (\texttt{commit}, u_k)$; a contradiction.

Now fix some $l \in \mathbb{N}$. Define $\lambda = \phi^0_{u_{l+1}}.w^0_{u_{l+1}}.\phi^1_{u_{l+1}}.\sigma_{u_l}.w^1_{u_{l+1}}$ as well as $\lambda' = \phi^0_{u_l}.\sigma_{u_{l+1}}.w^0_{u_l}$. Both runs are admissible due to equations (1) and (2), as well as the fact that R is not the first written register. We have $\lambda \vdash (\texttt{commit}, u_l)$, while $\lambda' \vdash (\texttt{commit}, u_{l+1})$. Moreover, all the registers end-up in the same state in both λ and λ'. Hence, for any process $p \notin \{p_l, p_{l+1}\}$, it is true that $\lambda \overset{p}{\sim} \lambda'$. Since such a process might exist (as $n > 2$), we reach the desired contradiction. □

4.2. Matching algorithm for the general case

We now present an adopt-commit object that executes only three write operations. Our construction is based on the notion of conflict detector introduced in [5]. We first present this abstraction then detail our approach and prove its correctness.

4.2.1. Conflict detector

Aspnes and Ellen [5] introduce the notion of conflict detector to further decompose an adopt-commit object. An m-valued conflict detector (CD) supports a single operation, $check(u)$, with $u \in Values$. This object returns *true* to indicate a conflict, that is when another value than u was checked, or *false* if no conflict occurs. More precisely, the following two properties hold: (*Convergence*) In any execution in which all $check()$ operations have the same input value, they all return *false*; and (*Conflict Detection*) In any execution that contains a $check(u)$ operation and a $check(v)$ operation, if $v \neq u$ then at least one of these two operations returns *true*.

We now recall the implementation of an adopt-commit object with the help of a conflict detector, as proposed in [5]. Algorithm 2 lists the pseudo-code of the approach. To execute $propose(u)$, a process p first inquiries the conflict detector, raising the shared flag F if a conflict occurs. Then, p fetches the content of the decision register D in variable d. If the retrieved value is null, process p stores its proposal in both registers D and variable d. In the next step of Algorithm 2, p checks the content of flag F. If the flag indicates that no conflict occurs, the content of d is committed and otherwise it is adopted.

At the light of Algorithm 2, we know that $WTIME(AC) \leq WTIME(CD) + 1$. Notice that as the converse reduction obviously holds without any additional operation, we have the following result:

Theorem 4 *([5]). $AC \equiv CD$*

Algorithm 2 Reducing Adopt-Commit to Conflict Detector [5] – code at process p.

```
1: Shared Variables:
2:     C                                              // A conflict detector
3:     D                                              // Initially, ⊥.
4:     F                                              // Initially, false.
5:
6: Procedure adoptCommit(u)
7:     if C.check(u) then
8:         F ← true
9:     d ← D
10:    if d = ⊥ then
11:        D ← u
12:        d ← u
13:    if F = true then
14:        return (adopt, d)
15:    return (commit, d)
```

Algorithm 3 Conflict detector in two writes – code at process p.

```
1: Shared Variables:
2:     ∀r ∈ [0, m] : R[r] = ⊥                         // An array of m + 1 MWMR atomic registers
3:
4: Procedure check(u)
5:     R[u + 1] ← u
6:     for all i ∈ [0, u] do
7:         if R[i] ∉ {u, ⊥} then
8:             return true
9:         if i = 0 then
10:            R[0] ← u
11:    return false
```

4.2.2. The construction

Algorithm 3 presents a conflict detector that executes two write operations during a solo execution. Similarly to the algorithm proposed in [5], this algorithm employs the idea that processes with distinct inputs access registers in distinct orders. Differently from [5], Algorithm 3 does not write to all the registers it encounters.

In detail, our algorithm works as follows: We consider an array of $m+1$ registers $R[0], \ldots, R[m]$. Upon a call to *check*(u), a process p first writes value u to register $R[u+1]$, then it reads all the registers from $R[0]$ to $R[u]$. If at some point in time, process p reads a non-null value that differs from u, p immediately returns *true* (line 8). Otherwise, p does not detect a conflict and returns *false* (line 11). To signal its presence, p writes u in register $R[0]$ over the course of the execution (line 10).

Theorem 5. *Algorithm 3 implements a wait-free CDobject, with WTIME(Algorithm 3)* $= 2$ *and TIME(Algorithm 3)* $\in O(m)$.

Proof. We first show that Algorithm 3 satisfies the two properties of a conflict detector.

- (*Convergence*) During an execution of Algorithm 3, if all the *check*() operations have the same argument, say u, then no other value than u is ever written to the registers. Hence, the test at line 7 is never true, and line 8 never occurs. Therefore, every correct process that executes *check*(u) eventually reaches line 11 and returns *false*
- (*Conflict Detection*) By contradiction. Consider some run λ during which *check*(u) and *check*(v) return both *false*, with $u \neq v$. These two operations are executed by two processes p_u and p_v, and we note respectively λ_u and λ_v the sequence of steps made by each process in λ. Furthermore, and without lack of generality, we assume that $u < v$.
  Given some register $R[i]$, let us note r_i a read from $R[i]$ and w_i a write to $R[i]$. At the light of Algorithm 3, λ_u contains operations w_{u+1} and r_0, while λ_v includes r_{u+1} and w_0. In both λ_v and λ_u, the test at line 7 does not trigger. Since p_v never writes to register $R[u+1]$, it follows that $r_{u+1} <_\lambda w_{u+1}$. As process p_u reads register $R[0]$ before writing it, we deduce $r_0 <_\lambda w_0$. On the other hand, the pseudo-code of Algorithm 3 tells us that $w_{u+1} <_{\lambda_u} r_0$ and $w_0 <_{\lambda_v} r_{u+1}$. We deduce that $<_\lambda$ is not an order; a contradiction.

At the light of its pseudo-code, Algorithm 3 is wait-free and contains two write operations. For some value $u \in$ *Values*, Algorithm 3 executes u read operations. Hence, the solo step complexity of Algorithm 3 belongs to $O(m)$. □

The solo step complexity of Algorithm 3 is independent from n, but when $m = \aleph_0$ it is not bounded. Such a result is unavoidable since Aspnes and Ellen [5] prove that $\Omega(min(\frac{\log(m)}{\log(\log(m))}, n))$ is a lower bound.

Algorithm 3 executes a total of two write operations. From the construction presented in Algorithm 2, we obtain an adopt-commit object exhibiting a solo-write complexity of three operations. Theorem 3 proves that this result is optimal in the general case.

Algorithm 4 Detecting a conflict in a single write – code at process p.

```
1: Shared Variables:
2:     ∀r ∈ [0, m − 1] : R[r] = false        // An array of m MWMR atomic registers.
3:
4: Procedure check(u)
5:     R[u] ← true
6:     for all i ∈ [0, m − 1] \ {u} do
7:         if R[i] = true} then
8:             return true
9:     return false
```

4.2.3. A corner case

When *Values* is bounded, or n identities are available in the system, we can leverage the fact that a process can read all the registers in Algorithm 3 before returning. If no previous value outside of the proposal of the process exists, it returns *false*. Every process writing to R will later detect a conflict.

We detail the variation for the case $|Values| < \aleph_0$ in Algorithm 4. When $c = n$, the algorithm is similar. The correctness of this algorithm follows from a reasoning close to the one we conducted above for Algorithm 3. It is left to the reader.

5. The Janus[2] Algorithm

The construction we previously presented can accommodate with any number of processes. On the other hand, its complexity depends on m. As we frequently encounter $m \gg n$ in practice, an algorithm whose complexity depends on n might be of more interest.

In this section, we present the Janus algorithm, a wait-free adopt-commit algorithm for anonymous shared-memory distributed systems. Janus executes $O(n)$ operations in a solo run, including $O(\sqrt{n})$ writes. We shall see later that these two values are optimal.

5.1. Description of Janus

Algorithm 5 depicts the pseudo-code of Janus. This adopt-commit algorithm works with anonymous processes, and the knowledge of the input values is not required beforehand. In particular, this set may be unbounded ($m = \aleph_0$). On the other hand, the total number of processes in the system (n), must be known in advance. Janus employs $\mathcal{K} \in \mathbb{N}$ shared registers, denoted hereafter $R[1], \ldots, R[\mathcal{K}]$. The execution proceeds in $\mathcal{K}$ asynchronous rounds, and each register $R[r]$ is used only in rounds $r \leq r' \leq \mathcal{K}$. A process p starts the algorithm when it invokes *adoptCommit*(u), with $u \in Values$. Process p stores the current round to which it participates in variable *rnd*. It also maintains the proposal it currently favors, or *estimate*, in the local variable *est*.

During round r, process p writes its estimate to register $R[r]$, then it looks back to see if another estimate appears in some register $R[r' < r]$. If this is the case, p raises flag C. Then, process p moves to a higher round. Once p has executed $\mathcal{K}$ such rounds, it commits *est* if C equals *false*, and adopts it otherwise.

With more details, process p executes the following steps in Janus.

- (lines 8 to 13) Process p first checks if a value has been already written to $R[rnd]$ (line 8). If this occurs, p immediately enters round $r \geq rnd$, where r is the greatest round for which a value has been written to the associated register $R[r]$, thus possibly skipping rounds $rnd, \ldots, r-1$ (line 9). In addition, p adopts the value currently stored in $R[r]$ as its new estimate. Otherwise, i.e., when $R[rnd] = \perp$, p writes its estimate to that register (line 13).
- (lines 14 to 16) Writing to register $R[rnd]$ is not sufficient to commit the value stored in *est*. Indeed, several other processes might be performing concurrently operations to the registers. In particular, a process entering round *rnd* might adopt $est' \neq est$ and attempts to commit such a value at a later time.
  Consequently, before moving to the next round, process p checks that no conflict is detected so far. This means that registers $R[1], \ldots, R[rnd-1]$ still store *est* (line 14). For large enough values of $\mathcal{K}$, this condition prevents any other value than *est* from being written to $R[rnd]$. We establish in Lemma 5 that for $\mathcal{K} \geq 2 \cdot \lceil \sqrt{n} \rceil + 1$ such a property holds. In case process p observes an estimate different than *est*, it raises the conflict flag C (line 15). Then, p can move to round $rnd + 1$.
- (lines 17 to 19) When process p has executed $\mathcal{K}$ such rounds, it checks flag C. If no conflict occurred, that is $C = false$, p returns $(\texttt{commit}, est)$; otherwise p returns $(\texttt{adopt}, est)$;

[2] In Roman religion and mythology, Janus is the god of gates. Most often he is depicted as having two heads, facing opposite directions (Wikipedia). The choice of the name is explained by the fact that each process in our algorithm has to look in two directions: forward, to check if a process has already started a new round, and backward to see if some process entered a previous round.

Algorithm 5 The Janus Algorithm – code at process p.

```
1:  Shared Variables:
2:      ∀r ∈ [1, K], R[r] = ⊥                          // A set of MWMR linearizable registers.
3:      C ∈ [true, false]                               // Initially, false.
4:
5:  adoptCommit(u) :=
6:      rnd ← 1
7:      while rnd ≤ K do
8:          if (R[rnd] ≠ ⊥) then                        // Existence of an estimate with higher priority.
9:              r ← max({j ≥ rnd : R[j] ≠ ⊥})
10:             est ← R[r]
11:             rnd ← r
12:         else
13:             R[rnd] ← est
14:         if rnd > 1 ∧ ∃r ∈ [1, rnd − 1] : R[r] ≠ est then   // Look for conflicts.
15:             C ← true
16:         rnd ← rnd + 1                               // Move to the next round.
17:     if C then
18:         return (adopt, est)
19:     return (commit, est)
```

5.2. Correctness

Fix some execution λ of Janus. Recall that since we consider linearizable registers, λ is a sequence of read/write operations on the shared registers and the local variables. Accordingly, we shall say that an operation op in λ *occurs at time* τ if op is at position τ in the execution λ. In what follows, we shall note var_p the local variable var of process p. The *execution of the (asynchronous) round r by p* consists of the sequence of steps taken by p during which $rnd_p = r$ holds.

A process executing round r writes its estimate est to register $R[r]$, provided it observes that no other value has been previously written to $R[r]$ (line 8). The following Lemma implies that if p performs such a write, then est has been previously written to $R[1], \ldots, R[r-1]$.

Lemma 1. *Consider $r > 1$. Suppose that a write operation op with parameter v is performed on $R[r]$. Then, a write operation op' with value v occurs on $R[r-1]$ before op.*

Proof. Suppose that op is performed by some process p. Observe that when this occurs (line 13), $rnd_p = r$ and $est_p = v$, that is v is the estimate of p at the beginning of round r. Since $r > 1$, the previous value of rnd_p is $r-1$ (line 16). We consider two cases according to the line at which p sets rnd_p to $r-1$.

- Process p sets rnd_p to $r-1$ at line 11. Thus, p executes line 10 and picks v' as its new estimate, where v' is the value p read from $R[r-1]$. As p does not modify again est_p in round $r-1$, v' is the value of est_p when p enters round r. Therefore, $v' = v$ and thus v was written before to $R[r-1]$.
- Process p sets rnd_p to $r-1$ at line 16. As p does not change the values of rnd_p at line 11, p reads $\perp$ from $R[r-1]$ and thus writes its current estimate v' in $R[r-1]$ (line 13). From the pseudo-code of Janus, v' is the estimate of p when p enters round r. Therefore, $v' = v$ and again, v was written to $R[r-1]$ before op. □

It follows from the previous result that Janus satisfies the validity requirement of adopt-commit (AC) objects. We prove precisely this property in the lemma that follows.

Lemma 2 *(Validity). Every adopted or committed value is a proposed value.*

Proof. Consider that a process p adopts or commits some value $est_p = v$ in Janus (lines 18 to 19). Clearly, $rnd_p = \mathcal{K} + 1$ at that time. At the beginning of round $\mathcal{K}$, p writes v to $R[\mathcal{K}]$ (line 13), or v is the value p read from $R[\mathcal{K}]$ (lines 10 and 11). Hence, in both cases, value v was written to $R[\mathcal{K}]$. It follows from Lemma 1 that v was written in each register $R[i]$, $1 \leq i \leq \mathcal{K}$. In particular, v was written in $R[1]$. The validity clause of AC follows from the fact that the values written in $R[1]$ are the processes' proposals. □

The above lemma directly implies that Janus satisfies the convergence property of AC.

Lemma 3 *(Convergence). If every process proposes the same value v, then* (commit, v) *is the only possible output.*

Proof. If some value v is adopted or committed, then from Lemma 2, value v is proposed. Hence, a register $R[r]$ may only contain either v or $\perp$, its initial value. Now, let us observe that *(i)* from Lemma 1, if $R[r > 1] = v$ holds then necessarily v

was written by some process in $R[r-1]$ previously, and *(ii)* if some process enters a round $r > 1$, necessarily $R[r-1] \neq \perp$ (lines 10 to 13). This implies that line 14 never triggers, and that C always equals *false*. As a consequence, we may conclude that $(\texttt{commit}, v)$ is the unique return value. □

From the pseudo-code of Janus every correct process eventually decides. Hence, the following lemma:

Lemma 4 *(Wait-freedom). Janus is wait-free.*

Proof of agreement. We now turn our attention to the agreement property To this end, we divide every execution λ in *epochs* as follows. Epoch $e_{i\geq 1}$ is the interval that starts with the first write to register $R[i]$ in λ, or if $i = 1$ the first operation in λ, and that ends immediately before the first write (if any) performed to register $R[i+1]$. Given some operation *op*, we say that *op occurs at epoch* e_i when *op* is in the interval e_i. Obviously, there is a single epoch during which an operation takes place. Moreover, if a write to $R[j]$ occurs at e_i, then $j \leq i$.

The following lemma is central to the proof of the agreement property. Informally, this lemma states that if a process writes v in $R[\mathcal{K}]$, then no other value than v can be written to $R[\mathcal{K}]$.

Lemma 5. *Let v be an adopted or committed value. It is true that* (i) *value v is written to $R[\mathcal{K}]$, and* (ii) *if value v is committed, for every value v' written to $R[\mathcal{K}]$, it holds that $v' = v$.*

The agreement property follows immediately:

Lemma 6 *(Agreement). If $(\texttt{commit}, v)$ is returned, then every decision has the form $(-, v)$;*

Proof. Consider some committed value v, and a value v' such that some process returns $(-, v')$. From (i) in Lemma 5, we know that both v and v' are written to register $R[K]$. Then, item (ii) tells us that, since v is committed, $v = v'$ holds. □

We devote the remaining of this section to the proof of Lemma 5. As we pointed out previously, if some process p commits a value v necessarily $est_p = v$ when p returns $(\texttt{commit}, v)$. Moreover, at the time p executes the last iteration of the while loop, we have $rnd_p = \mathcal{K}$. Thus, the pseudo-code from lines 8 to 13 implies that $R[\mathcal{K}] = v$ at some point in time before process p returns. This shows that item (i) in Lemma 5 holds.

To prove that (ii) is true as well, we proceed by contradiction. Let us name H the negation of item (ii) in Lemma 5; namely:

$$\text{Two distinct values } u \text{ and } v \text{ are written to } R[\mathcal{K}]. \tag{H}$$

In the following, we show that to satisfy (H), the system must consist of at least $n+1$ processes.

For $i, j \in [1, \mathcal{K}]$, let us note $\mathcal{W}_j^i$ the set of processes that perform a write operation to register $R[j]$ during epoch e_i. This means that a process p belongs to $\mathcal{W}_j^i$ if and only if there exists a write operation to $R[j]$ by p that occurs at epoch e_i. From the definition of an epoch, we know that if $j > i$, then $\mathcal{W}_j^i = \varnothing$.

We first state two technical lemmas.

Lemma 7. *Suppose that p performs a write operation op on $R[i]$. The last operation preceding op performed by p is a read on $R[i]$, and the value returned by that operation is $\perp$.*

Proof. Immediate from the code of Janus at lines 8 to 13. □

Lemma 8. *Denote by op and op' two write operations performed by the same process p. Suppose that: (1) op occurs at epoch e_i, (2) op' is a write to register $R[j]$ with $j \neq i$, and (3) op precedes op'. Then, $j > i$ and op' occurs at some epoch $e_{j'>i}$.*

Proof. By Lemma 7, p reads from $R[j]$ immediately before executing op', and this read operation returns $\perp$. Let op'' denote that operation. It follows from the third condition of the Lemma that op'' occurs after op, which in turn occurs after some non-$\perp$ value has been written to $R[i']$ for each $i' \leq i$ (from Lemma 1). Since the read operation op'' performed on $R[j]$ returns $\perp$, we conclude that $j > i$. Hence, by definition of an epoch, op' takes place in $e_{j'\geq j}$. □

The lemmas below precise how the sizes of $(\mathcal{W}_j^i)_{i,j}$ and the round numbers are related. They are instrumental in showing that (H) does not hold.

Lemma 9. *Under (H), it is true that: $\forall i \in [1, \mathcal{K}-1] : |\mathcal{W}_i^i| \geq 2$.*

Proof. From (H), at least two values u and v are written to $R[\mathcal{K}]$. Lemma 1 tells us that in such a case both u and v are written to $R[i]$ for every $i \in [1, \mathcal{K}-1]$. For some $R[i]$, we show that two such writes occurs precisely at epoch e_i. Consider the first write of value v to $R[i]$. By definition, this operation occurs at epoch $e_{i'}$ for some $i' \geq i$.

For the sake of contradiction, suppose that $i' > i$. By applying inductively Lemma 1, when v is written to $R[i]$ for the first time, register $R[i+1]$ does not contain $\perp$. Let p be the process that performs the first write of v to $R[i+1]$ and note w_p^{i+1} this operation.

According to the code of Janus we know that *(1)* Process p performs w_p^{i+1} while executing round $i+1$; *(2)* Operation w_p^{i+1} is preceded by a read operation on $R[i+1]$ at line 8 by p that returns $\perp$, an operation we denote r_p^{i+1}; and *(3)* During round i, there is a read operation from $R[i]$ that returns value v, or a write of v by p to $R[i]$. This last operation is denoted op_p^i.

Operations op_p^i, r_p^{i+1} and w_p^{i+1} occur in that order. Since the first write of v to $R[i]$ occurs at $e_{i'}$, op_p^i occurs at some epoch $e_{i''}$ with $i'' \geq i'$. Therefore, operation r_p^{i+1} occurs after a write to $R[i+1]$, from which we conclude that r_p^{i+1} returns a non-$\perp$ value. The pseudo-code at lines 8 to 11 tells us that in such a case p does not write to $R[i+1]$; a contradiction.

We just show that a write of v to $R[i]$ occurs at epoch e_i. A symmetrical argument can be applied to value u. For some process p, rnd_p is strictly growing. Hence, for each $i \in [1, \mathcal{K}-1]$, a process performs at most one write operation to $R[i]$. This shows that $|\mathcal{W}_i^i| \geq 2$. □

Lemma 10. *If (H) holds, then:* $\forall i, j \in [1, \mathcal{K}-1] \times [1, i-1] : |\mathcal{W}_j^i| \geq 1$.

Proof. Choose some i in $[1, \mathcal{K}-1]$. As a starter, we establish that two read operations that return respectively v and u occur at epoch e_i.

Since value v is written in $R[\mathcal{K}]$, v is also written to $R[i+1]$ (Lemma 1). Let p the process that performs the first write of v to $R[i+1]$. From the code of Janus, p executes round i before performing that write operation, and v is the estimate of p at the end of that round. Hence, at the beginning of round i, p either reads v in $R[i]$ or writes v in $R[i]$. Moreover, the read operation on $R[i+1]$ performed by p at the beginning of round $i+1$ returns $\perp$ (otherwise p does not perform a write to $R[i+1]$). Therefore, every operation performed by p while executing round i occurs at epoch e_i. In particular, for every $j \in [1, i-1]$, the read of $R[j]$ performed by p at line 14 occurs at e_i. This read must return v. Otherwise, p raises flag C, and value v is not committed.

Similarly, by considering the process that performs the first write of u in $R[i+1]$, we obtain that a read operation on $R[j]$ returning u occurs at e_i.

As two read operations on $R[j]$ return two different values occur in e_i, there must exist a write operation on $R[j]$ that occurs at e_i. We thus conclude that $\mathcal{W}_j^i \neq \varnothing$. □

Lemma 11. *Suppose that (H) holds. Choose* $i, j \in [1, \mathcal{K}-1] \times [1, i-1]$ *and* $i', j' \in [1, \mathcal{K}-1] \times [1, i'-1]$. *It is true that:* $(i \leq i' \wedge \mathcal{W}_j^i \cap \mathcal{W}_{j'}^{i'} \neq \varnothing) \rightarrow (i = i' \wedge j = j') \vee (i < j')$

Proof. Pick $p \in \mathcal{W}_j^i \cap \mathcal{W}_{j'}^{i'}$. By definition, a write operation by p occurs at e_i and $e_{i'}$. Assume first that $i = i'$. Lemma 8 tells us that two consecutive write operations by the same process should occur in distinct epoch. Hence, $j = j'$ holds. Otherwise, $i < i'$ and applying again Lemma 8, we obtain $i < j'$. □

We are now ready to prove Lemma 5.

Proof of Lemma 5. Assume for the sake of contradiction that (H) is satisfied, and consider the following set:

$$S = \left\{ (i, j) : \left\lceil \frac{\mathcal{K}-1}{2} \right\rceil \leq i \leq \mathcal{K}-1 \ \wedge \ 1 \leq j \leq \left\lceil \frac{\mathcal{K}-1}{2} \right\rceil \right\}$$

In what follows, we count the distinct processes that appear in the union of the sets $\mathcal{W}_j^i$ with $(i, j) \in S$. We show that there are at least $n+1$, reaching a contradiction.

Let $(i, j) \neq (i', j') \in S$ such that $i \leq i'$. By definition of S, $i \geq j'$ and thus it follows from Lemma 11 that $\mathcal{W}_j^i \cap \mathcal{W}_{j'}^{i'} = \varnothing$. Hence,

$$\left| \bigcup_{(i,j) \in S} \mathcal{W}_j^i \right| = \sum_{(i,j) \in S} |\mathcal{W}_j^i|$$

From Lemmas 9 and 10, we have $|\mathcal{W}_j^i| \geq 1$ for each $(i, j) \in S$ and $|\mathcal{W}_i^i| \geq 2$ for each $(i, i) \in S$. Therefore,

$$\left| \bigcup_{|(i,j)\in S} \mathcal{W}_j^i \right| \geq \left\lceil \frac{\mathcal{K}-1}{2} \right\rceil \cdot \left\lfloor \frac{\mathcal{K}-1}{2} \right\rfloor + 1$$

Finally, as $\mathcal{K} = 2 \cdot \lceil \sqrt{n} \rceil + 1$, we get

$$\left| \bigcup_{(i,j)\in S} \mathcal{W}_j^i \right| \geq n+1 \quad \square$$

From Lemma 5, we know that no two distinct value are written to $R[\mathcal{K}]$. Consequently, the agreement property holds, closing the proof of the agreement property.

Theorem 6. *Janus implements a wait-free adopt-commit object.*

Proof. Follows from Lemmas 2, 3, 4 and 6. $\square$

5.3. Time complexity

The theorem below proves that the solo step complexity of Janus is $O(n)$; this is optimal [5]. It also establishes that the solo-write complexity of this algorithm is $O(\sqrt{n})$. As we shall see in Section 6, this last value is tight.

Theorem 7. *TIME(Janus)* $\in O(n)$ *and WTIME(Janus)* $\in O(\sqrt{n})$.

Proof. Consider a solo execution of some process p. During this execution, p executes $\mathcal{K} = 2\lceil \sqrt{n} \rceil + 1$ rounds, then decides. Name $\{1, \ldots, \mathcal{K}\}$ the rounds executed by p, and consider some round i. According to the pseudo-code of Janus, during round i process p executes a single write (line 13), and reads i shared registers (lines 8 and 14). As a consequence, the step complexity of the algorithm is $O(n)$ and its write complexity belongs to $O(\sqrt{n})$. $\square$

6. When identities help

In this section, we combine Janus and Algorithm 3 to efficiently leverage the presence of identities. When c identities are available, our solution has a solo-write complexity of $\Omega(\sqrt{n-c+1})$ operations. Before delving into its algorithmic details, we first establish that this value is optimal when m is unknown and the memory footprint is bounded.

6.1. A lower bound result

For some process p, recall that a clone of p refers to a process homonymous to p that executes in lock-step with p. This process is indistinguishable from p to other processes. In particular, for any process p and any execution λ during which less than n processes take steps, we may always consider an execution λ' indistinguishable from λ to all processes and that includes a clone of p.

Let $\mathcal{A}$ be some implementation of an adopt-commit object. For some input value v, we note A_v the permutation over $ws(v)$ following the order in which a process in σ_v first writes to a register in $ws(v)$.

Proposition 4. $(\exists u, v \in \mathit{Values} : A_u = A_v) \rightarrow \mathit{WTIME}(\mathcal{A}) \in \Omega(\sqrt{n-c+1})$

Proof. Choose two values u and v that satisfy $A_u = A_v$. In what follows, we construct with the help of clones an execution λ that is indistinguishable from σ_u for process p_u and from σ_v for process p_v.

The construction. Let us define $A_u = A_v = \langle R_1, \ldots, R_k \rangle$. Given a register $R_i \in A_u$, note $w_{i,u}$ the first write to R_i during σ_u. We define symmetrically operation $w_{i,v}$. For each register $R_{i\in[1,k]}$, we schedule iteratively the operations in σ_u and σ_v in λ as follows: We schedule in λ the operation $w_{i-1,u}$ (if such an operation exists) in σ_u, then every operation that follows $w_{i-1,u}$ and precedes $w_{i,u}$. By definition of A_u, observe that none of these operations is a write to some register $R_{j\geq i}$. Similarly, we then schedule $w_{i-1,v}$ and all the operations in σ_v between $w_{i-1,v}$ and $w_{i,v}$. The previous construction is iterated until we have scheduled all the operations of σ_u and σ_v. Then, we add clones of p_u and p_v as follows: For some register R_i, let $w_{i,u,j}$ be the last write to register $R_{j<i}$ by p_u prior to $w_{i,u}$. We add an operation $w_{i,u,j}$ by a clone of p_u right after $w_{i,u}$. After this block write, the clone stops. Similarly for v, we add a write $w_{i,v,j}$ over $R_{j<i}$ right after $w_{i,v}$ by a clone of p_v for every $j < i$.

Correctness. In order to prove that λ is an admissible run, we first consider some read of a register R_i by p_u, and examine the following cases: (case $R_i \in A_u$) Every read from R_i is either *(i)* before $w_{i,u}$ and thus it sees the initial state because no

operation of p_u, nor of p_v, has written R_i yet, or *(ii)* after some $w_{j\geq i,u}$ and it sees the result either of *(ii-a)* the operation $w_{j,u,i}$ by a clone of p_u, or *(ii-b)* the result of some write of p_u after $w_{j,u,i}$. Hence, in all the situations above, the read of p_u in λ sees the same result as in σ_u. (Case $R_i \notin A_u$.) In such a case p_v never writes R_i. As a consequence, all such reads by p_u in σ_u are the same as in λ. It follows from the previous reasoning that $\lambda \overset{p_u}{\sim} \sigma_v$. A symmetrical argument leads to $\lambda \overset{p_v}{\sim} \sigma_v$. Hence, λ is an admissible run.

Execution λ makes use of $2 + 2\sum_{i=1}^{k}(i-1) = k^2 - k + 2$ processes. In an homonymous system with c identities, we may split processors in a group of $c-1$ processes having $c-1$ identities, and a group of $n-c+1$ processes with the same identity. Hence, λ is not constructible when $k^2 - k + 2 > n - c + 1$. On the other hand, we have $WTIME(\mathcal{A}) \geq k$. This leads to the fact $WTIME(\mathcal{A}) \in \Omega(\sqrt{n-c+1})$. □

Proposition 4 implies the following result:

Theorem 8. $WTIME(AC) \in \Omega(min(\frac{\log(m)}{\log(\log(m))}, \sqrt{n-c+1}))$.

Proof. Consider an *AC* implementation $\mathcal{A}$ that uses a bounded amount of registers, say k. Without lack of generality, we assume that some solo execution σ_u of $\mathcal{A}$ writes to the k registers, that is $WTIME(\mathcal{A}) = k$. There are $\sum_{i=0}^{k} C_i^k$ set of registers to write, and for each such set of size i, $i!$ possible ways to write the registers first in some execution σ_u. This leads to $\sum_{i=0}^{k} P_i^k \leq k! \times e$ possible choices of writing first the (at most) k registers. As a consequence, the pigeon hole principle tells us that if $m \geq k! \times e$, for some pair $u, v \in Values$, the premises of Proposition 4 apply, i.e., $A_u = A_v$, leading to $WTIME(\mathcal{A}) \in \Omega(\sqrt{n-c+1})$. On the other hand, if $m < k! \times e$ then $WTIME(\mathcal{A}) \in \Omega(\frac{\log(m)}{\log(\log(m))})$. Hence, $WTIME(AC) \in \Omega(min(\frac{\log(m)}{\log(\log(m))}, \sqrt{n-c+1}))$. □

In [5], Aspnes and Ellen present a wait-free implementation of an m-valued adopt-commit objects from multi-reader multi-writer registers that works in anonymous systems. This algorithm makes use of a bounded amount of registers and executes $O(\frac{\log m}{\log\log m})$ write operations, reaching the left part of the above lower bound. The section that follows details an implementation of adopt-commit that satisfies a solo-write complexity of $O(\sqrt{n-c+1})$ operations.

6.2. An asymptotically optimal solution

To match the above lower bound, we proceed as follows: First, processes having the same identity agree on a common proposed value. To this end, we employ c instances of Janus, with $\mathcal{K}$ set to $2\lceil\sqrt{n-c+1}\rceil + 1$, one per identity. Then, processes execute the constant write time algorithm presented in Section 4, agreeing on an identity (and thus some associated proposed value). This defines the *fast path* of our algorithm. However, it might happen that, despite proposing all the same value, processes do not reach agreement at the end of this path. To solve such an issue and attain convergence, a process which returns (`adopt`,−) in the fast path executes an instance of Janus with $\mathcal{K}$ set to $2\lceil\sqrt{n}\rceil + 1$,

6.2.1. Algorithmic details

We depict our solution in Algorithm 6. This algorithm makes use of the following four variables: c instances of Janus (variables $I[1..c]$), an array of c registers (variables $D[1..c]$), an instance of Algorithm 2 using Algorithm 3 (variable C), and an additional instance of Janus (variable J).

These variables are employed as follows. In a first step, processes having the same identity agree on a common value (line 8). To this end, a process p accesses instance $I[p]$, proposing its input. Since c identities are available in the system, at most $n-c+1$ processes may access some instance $I[p]$. Hence, for each instance $I[p]$, we set $\mathcal{K}$ to $2\lceil\sqrt{n-c+1}\rceil + 1$. In a second step, processes compete to pick an identity and the proposed value that was chosen during the first step (lines 9 to 10). This tentative operation to commit a value employs variable C. If a process does not succeed in committing a value after the above two steps, it executes an additional instance of Janus accessible through variable J.

6.2.2. Correctness

Theorem 9 proves that Algorithm 6 is a correct implementation of *AC* in a distributed homonymous system where c identities available.

Theorem 9. *Algorithm 6 implements a wait-free adopt-commit object.*

Proof. In what follows, we prove that Algorithm 6 precisely implements a wait-free adopt-commit object:

(Wait-freedom) Variables $I[1..c]$, C and J are all wait-free *AC* implementations. As a consequence, at the light of the pseudo-code of Algorithm 6, this algorithm is also wait-free.

Algorithm 6 Adopt-Commit for Homonymous Systems – code at process p.

```
1: Shared Variables:
2:     I[c]                                  // An array of c instances of Janus with K = 2⌈√(n − c + 1)⌉ + 1.
3:     D[c]                                  // An array of c registers; initially ∀i ∈ [1, c] : D[i] = ⊥.
4:     C                                     // Algorithm 2 using Algorithm 3 with m = c
5:     J                                     // Janus with K = 2. ⌈√n⌉ + 1.
6:
7: Procedure adoptCommit(u)
8:     D[p] ← I[p].adoptCommit(u)
9:     (g, q) ← C.adoptCommit(p)
10:    (f, est) ← D[q]
11:    if f = adopt ∨ g = adopt then
12:        (f, est) ← J.adoptCommit(est)
13:    return (f, est)
```

(Validity) Consider that a value u is adopted or committed at line 13. Value u is either retrieved from the array D at line 10, or it results from the call at line 12. In the former case, from the validity property of an adopt-commit object, we observe that value u is necessarily proposed at line 8 to some instance $I[k]$, with $k \in [1, c]$. Hence, value u is the input value of some process. In the latter case, u is the output of instance J. From the validity property of an adopt-commit object, u is fetched from array D at line 10. Hence this case boils down to the previous one.

(Agreement) Let us consider that some process p commits a value u, while a process q adopts or commits some value v. As a starter, we observe that every value decided at line 13 in Algorithm 6 is necessarily retrieved either at line 10, or at line 12. Assume first that u is computed at line 10. We observe that both flags f and g equals `commit` when p tests them at line 11. It follows that *(i)* p fetches $(\texttt{commit}, u)$ from some register $D[\hat{p}]$ at line 10, for some $\hat{p} \in [1, c]$, and *(ii)* C returns $(\texttt{commit}, \hat{p})$ at line 9. Only processes having identity $\hat{p}$ might write to $D[\hat{p}]$ at line 8, and precisely, they write down the result of the call to object $I[\hat{p}]$. From item *(i)*, solely a tuple of the form $(-, u)$ can be written to $D[\hat{p}]$. From item *(ii)*, we deduce that q returns $(-, \hat{p})$ at line 9. Hence, q reads from $D[\hat{p}]$ as well at line 10. It follows that q reads $(-, u)$ from register $D[\hat{p}]$. Thus, u is the sole value that can be proposed at line 12. From the convergence property of object J, u is committed by q. If v is computed at line 10, a similar argument holds. In the last case where both values u and v are retrieved at line 12, the agreement property of object J implies that $u = v$.

(Convergence) Consider that all the processes propose the same value, say u. Since the validity property holds, $(-, u)$ is the sole decision a process may take. As detailed above, Algorithm 6 consists of two paths. The fast path spans lines 8 to 10. If a process p fails to commit a value in this path, i.e., in the advent where f or g equals `adopt` at line 11, p falls back to the slow path and executes line 12. Now, as u is the sole proposed value, process p necessarily retrieves u from variables $D[1..c]$ at line 10. It follows that every process accessing variable J proposes value u. By the convergence property of object J, p commits value u at line 12. □

Below, we establish that Algorithm 6 reaches the optimal solo-write complexity.

Theorem 10. *WTIME*(*Algorithm 6*) $\in O(\sqrt{n - c + 1})$

Proof. If some process p calls solo an adopt-commit object with value u, the convergence property implies that p commits u. As a consequence, a process that executes solo Algorithm 6 only takes the fast path, skipping line 12. This fast path consists in a call to an instance of Janus, with $\mathcal{K}$ set to $2\lceil\sqrt{n - c + 1} + 1\rceil$, then a call to an instance of Algorithm 3. From Theorems 5 and 7, we obtain that *WTIME*(Algorithm 6) belongs to $O(\sqrt{n - c + 1})$. □

7. Conclusion

This paper focuses on the minimal number of write operations a process should execute to reach an agreement with its peers in a distributed system. To that regard, we contribute several tight lower bound results on the solo-write complexity of adopt-commit objects, a pivotal abstraction at core of every consensus algorithm.

In detail, we first present an algorithm that executes three write operations, a value we show optimal in the general case. We show that this number reduces to 2 when m is bounded and known, or when n identities are available in the system. We also prove that a single write is necessary and sufficient in the corner case when $(n = 2) \wedge ((c = 2) \vee (m < \aleph_0))$. Further, we introduce Janus an efficient implementation for the anonymous case that executes $O(n)$ shared memory operations, including $O(\sqrt{n})$ writes. The lower bound result of Aspnes and Ellen [5] implies that the time complexity of Janus is optimal.

Building upon Janus, we then address the question of leveraging the presence of c identities in the system. We design a solution that implements an adopt-commit object in solely $O(\sqrt{n - c + 1})$ write operations, and we prove that, when m is not known and the number of registers in use bounded, this value is optimal.

Declaration of competing interest

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

References

[1] K. Abrahamson, On achieving consensus using a shared memory, in: Proceedings of the Seventh Annual ACM Symposium on Principles of Distributed Computing, PODC'88, ACM, New York, NY, USA, ISBN 0-89791-277-2, 1988, pp. 291–302, http://doi.acm.org/10.1145/62546.62594.

[2] D. Alistarh, R. Gelashvili, G. Nadiradze, Lower bounds for shared-memory leader election under bounded write contention, CoRR, arXiv:2108.02802 [abs], 2021, https://arxiv.org/abs/2108.02802.

[3] D. Angluin, J. Aspnes, Z. Diamadi, M.J. Fischer, R. Peralta, Computation in networks of passively mobile finite-state sensors, in: Proceedings of the Twenty-Third Annual ACM Symposium on Principles of Distributed Computing, PODC'04, ACM, New York, NY, USA, ISBN 1-58113-802-4, 2004, pp. 290–299, http://doi.acm.org/10.1145/1011767.1011810.

[4] J. Aspnes, A modular approach to shared-memory consensus, with applications to the probabilistic-write model, in: Proceedings of the 29th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing, PODC'10, ACM, New York, NY, USA, ISBN 978-1-60558-888-9, 2010, pp. 460–467, http://doi.acm.org/10.1145/1835698.1835802.

[5] J. Aspnes, F. Ellen, Tight bounds for adopt-commit objects, Theory Comput. Syst. (ISSN 1432-4350) 55 (3) (2014) 451–474, https://doi.org/10.1007/s00224-013-9448-1.

[6] H. Attiya, A. Gorbach, S. Moran, Computing in totally anonymous asynchronous shared memory systems, Inf. Comput. (ISSN 0890-5401) 173 (2) (Mar. 2002) 162–183, https://doi.org/10.1006/inco.2001.3119.

[7] H. Attiya, R. Guerraoui, P. Kouznetsov, Computing with reads and writes in the absence of step contention, in: P. Fraigniaud (Ed.), Distributed Computing: 19th International Conference, DISC 2005, Proceedings, Cracow, Poland, September 26–29, 2005, Springer Berlin Heidelberg, Berlin, Heidelberg, ISBN 978-3-540-32075-3, 2005, pp. 122–136.

[8] H. Attiya, O. Ben-Baruch, D. Hendler, Lower bound on the step complexity of anonymous binary consensus, in: C. Gavoille, D. Ilcinkas (Eds.), Distributed Computing, Springer Berlin Heidelberg, Berlin, Heidelberg, ISBN 978-3-662-53426-7, 2016, pp. 257–268.

[9] M. Ben-Or, Another advantage of free choice (extended abstract): completely asynchronous agreement protocols, in: Proceedings of the Second Annual ACM Symposium on Principles of Distributed Computing, PODC'83, ACM, New York, NY, USA, ISBN 0-89791-110-5, 1983, pp. 27–30, http://doi.acm.org/10.1145/800221.806707.

[10] F. Bonnet, M. Raynal, Brief announcement: the price of anonymity: optimal consensus despite asynchrony, crash and anonymity, in: Proceedings of the 28th ACM Symposium on Principles of Distributed Computing, PODC'09, ACM, New York, NY, USA, ISBN 978-1-60558-396-9, 2009, pp. 294–295, http://doi.acm.org/10.1145/1582716.1582773.

[11] E. Borowsky, E. Gafni, Generalized flp impossibility result for t-resilient asynchronous computations, in: Proceedings of the Twenty-Fifth Annual ACM Symposium on Theory of Computing, STOC'93, ACM, New York, NY, USA, ISBN 0-89791-591-7, 1993, pp. 91–100, http://doi.acm.org/10.1145/167088.167119.

[12] Z. Bouzid, C. Travers, Anonymity-preserving failure detectors, in: C. Gavoille, D. Ilcinkas (Eds.), Distributed Computing, Springer Berlin Heidelberg, Berlin, Heidelberg, ISBN 978-3-662-53426-7, 2016, pp. 173–186.

[13] Z. Bouzid, P. Sutra, C. Travers, Anonymous agreement: the janus algorithm, in: Proceedings of the 15th International Conference on Principles of Distributed Systems, OPODIS'11, Springer-Verlag, Berlin, Heidelberg, ISBN 9783642258725, 2011, pp. 175–190.

[14] Z. Bouzid, M. Raynal, P. Sutra, Anonymous obstruction-free (n, k)-set agreement with n-k+1 atomic read/write registers, Distrib. Comput. 31 (2) (2018) 99–117, https://doi.org/10.1007/s00446-017-0301-7.

[15] H. Buhrman, A. Panconesi, R. Silvestri, P.M.B. Vitányi, On the importance of having an identity or is consensus really universal?, in: Proceedings of the 14th International Conference on Distributed Computing, DISC'00, Springer-Verlag, London, UK, ISBN 3-540-41143-7, 2000, pp. 134–148, http://dl.acm.org/citation.cfm?id=645957.756683.

[16] C. Capdevielle, C. Johnen, P. Kuznetsov, A. Milani, On the uncontended complexity of anonymous agreement, Distrib. Comput. (ISSN 0178-2770) 30 (6) (Dec. 2017) 459–468, https://doi.org/10.1007/s00446-017-0297-z.

[17] A. Castañeda, S. Rajsbaum, M. Raynal, Unifying concurrent objects and distributed tasks: interval-linearizability, J. ACM (ISSN 0004-5411) 65 (6) (Nov. 2018).

[18] T.D. Chandra, S. Toueg, Unreliable failure detectors for reliable distributed systems, J. ACM (ISSN 0004-5411) 43 (2) (Mar. 1996) 225–267, https://doi.org/10.1145/226643.226647.

[19] T.D. Chandra, V. Hadzilacos, S. Toueg, The weakest failure detector for solving consensus, J. ACM (ISSN 0004-5411) 43 (4) (July 1996) 685–722, https://doi.org/10.1145/234533.234549.

[20] S. Chaudhuri, Agreement is harder than consensus: set consensus problems in totally asynchronous systems, in: Proceedings of the Ninth Annual ACM Symposium on Principles of Distributed Computing, PODC'90, ACM, New York, NY, USA, ISBN 0-89791-404-X, 1990, pp. 311–324, http://doi.acm.org/10.1145/93385.93431.

[21] T. Chothia, K. Chatzikokolakis, A survey of anonymous peer-to-peer file-sharing, in: Proceedings of the 2005 International Conference on Embedded and Ubiquitous Computing, EUC'05, Springer-Verlag, Berlin, Heidelberg, ISBN 3-540-30803-2, 2005, pp. 744–755.

[22] C. Delporte-Gallet, H. Fauconnier, Two Consensus Algorithms with Atomic Registers and Failure Detector Ω, Springer Berlin Heidelberg, Berlin, Heidelberg, ISBN 978-3-540-92295-7, 2009, pp. 251–262.

[23] C. Delporte-Gallet, H. Fauconnier, R. Guerraoui, A.-M. Kermarrec, E. Ruppert, H. Tran-The, Byzantine agreement with homonyms, in: Proceedings of the 30th Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing, PODC'11, ACM, New York, NY, USA, ISBN 978-1-4503-0719-2, 2011, pp. 21–30, http://doi.acm.org/10.1145/1993806.1993810.

[24] C. Delporte-Gallet, H. Fauconnier, E. Gafni, L. Lamport, Adaptive register allocation with a linear number of registers, in: Proceedings of the 27th International Symposium on Distributed Computing - Volume 8205, DISC 2013, Springer-Verlag New York, Inc., New York, NY, USA, ISBN 978-3-642-41526-5, 2013, pp. 269–283.

[25] C. Dwork, N. Lynch, L. Stockmeyer, Consensus in the presence of partial synchrony, J. ACM (ISSN 0004-5411) 35 (2) (Apr. 1988) 288–323, https://doi.org/10.1145/42282.42283.

[26] H. Federrath (Ed.), International Workshop on Designing Privacy Enhancing Technologies: Design Issues in Anonymity and Unobservability, Springer-Verlag New York, Inc., New York, NY, USA, ISBN 3-540-41724-9, 2001.

[27] F. Fich, M. Herlihy, N. Shavit, On the space complexity of randomized synchronization, in: Proceedings of the Twelfth Annual ACM Symposium on Principles of Distributed Computing, PODC'93, ACM, New York, NY, USA, ISBN 0-89791-613-1, 1993, pp. 241–249, http://doi.acm.org/10.1145/164051.164078.

[28] M.J. Fischer, N.A. Lynch, M.S. Paterson, Impossibility of distributed consensus with one faulty process, J. ACM (ISSN 0004-5411) 32 (2) (Apr. 1985) 374–382, https://doi.org/10.1145/3149.214121.
[29] E. Gafni, Round-by-round fault detectors (extended abstract): unifying synchrony and asynchrony, in: Proceedings of the Seventeenth Annual ACM Symposium on Principles of Distributed Computing, PODC'98, ACM, New York, NY, USA, ISBN 0-89791-977-7, 1998, pp. 143–152.
[30] P. Gastin, Recognizable and rational languages of finite and infinite traces, in: C. Choffrut, M. Jantzen (Eds.), STACS, vol. 91, Springer Berlin Heidelberg, Berlin, Heidelberg, ISBN 978-3-540-47002-1, 1991, pp. 89–104.
[31] G. Giakkoupis, M. Helmi, L. Higham, P. Woelfel, An $O\sqrt{N}$ space bound for obstruction-free leader election, in: Proceedings of the 27th International Symposium on Distributed Computing - Volume 8205, DISC 2013, Springer-Verlag New York, Inc., New York, NY, USA, ISBN 978-3-642-41526-5, 2013, pp. 46–60.
[32] P.B. Gibbons, How emerging memory technologies will have you rethinking algorithm design, in: Proceedings of the 2016 ACM Symposium on Principles of Distributed Computing, PODC'16, ACM, New York, NY, USA, ISBN 978-1-4503-3964-3, 2016, p. 303, http://doi.acm.org/10.1145/2933057.2933124.
[33] E. Godard, D. Imbs, M. Raynal, G. Taubenfeld, From Bezout's identity to space-optimal election in anonymous memory systems, in: Proceedings of the 39th Symposium on Principles of Distributed Computing, PODC'20, New York, NY, USA, Association for Computing Machinery, ISBN 9781450375825, 2020, pp. 41–50.
[34] R. Guerraoui, M. Raynal, The information structure of indulgent consensus, IEEE Trans. Comput. (ISSN 0018-9340) 53 (4) (Apr. 2004) 453–466, https://doi.org/10.1109/TC.2004.1268403.
[35] R. Guerraoui, M. Raynal, The alpha of indulgent consensus, Comput. J. (ISSN 0010-4620) 50 (1) (Jan. 2007) 53–67, https://doi.org/10.1093/comjnl/bxl046.
[36] R. Guerraoui, E. Ruppert, What Can Be Implemented Anonymously?, Springer Berlin Heidelberg, Berlin, Heidelberg, ISBN 978-3-540-32075-3, 2005, pp. 244–259.
[37] R. Guerraoui, E. Ruppert, Anonymous and fault-tolerant shared-memory computing, Distrib. Comput. (ISSN 1432-0452) 20 (3) (2007) 165–177, https://doi.org/10.1007/s00446-007-0042-0.
[38] J.L. Hennessy, D.A. Patterson, Computer Architecture, Fifth Edition: A Quantitative Approach, 5th edition, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, ISBN 012383872X, 2011, ISBN 9780123838728.
[39] M. Herlihy, Wait-free synchronization, ACM Trans. Program. Lang. Syst. (ISSN 0164-0925) 13 (1) (Jan. 1991) 124–149, https://doi.org/10.1145/114005.102808.
[40] M. Herlihy, Asynchronous Consensus Impossibility, Springer US, Boston, MA, ISBN 978-0-387-30162-4, 2008, pp. 1–99.
[41] M. Herlihy, N. Shavit, The topological structure of asynchronous computability, J. ACM (ISSN 0004-5411) 46 (6) (Nov. 1999) 858–923, https://doi.org/10.1145/331524.331529.
[42] M. Herlihy, N. Shavit, On the nature of progress, in: A.F. Anta, G. Lipari, M. Roy (Eds.), OPODIS, in: Lecture Notes in Computer Science, vol. 7109, Springer, ISBN 978-3-642-25872-5, 2011, pp. 313–328.
[43] M. Herlihy, V. Luchangco, M. Moir, Obstruction-free synchronization: double-ended queues as an example, in: 23rd International Conference on Distributed Computing Systems (ICDCS 2003), 19–22 May 2003, Providence, RI, USA, 2003, pp. 522–529.
[44] L. Lamport, A fast mutual exclusion algorithm, ACM Trans. Comput. Syst. (ISSN 0734-2071) 5 (1) (Jan. 1987) 1–11, https://doi.org/10.1145/7351.7352.
[45] L. Lamport, The part-time parliament, ACM Trans. Comput. Syst. (ISSN 0734-2071) 16 (2) (1998) 133–169.
[46] M.C. Loui, H.H. Abu-Amara, Memory Requirements for Agreement Among Unreliable Asynchronous Processes, vol. 4, JAI Press, 1987, pp. 163–183.
[47] V. Luchangco, M. Moir, N. Shavit, On the Uncontended Complexity of Consensus, Springer Berlin Heidelberg, Berlin, Heidelberg, ISBN 978-3-540-39989-6, 2003, pp. 45–59.
[48] M. Moir, J.H. Anderson, Fast, long-lived renaming (extended abstract), in: G. Tel, P.M.B. Vitányi (Eds.), Distributed Algorithms, 8th International Workshop, WDAG'94, Proceedings, Terschelling, the Netherlands, September 29 – October 1, 1994, in: Lecture Notes in Computer Science, vol. 857, Springer, ISBN 3-540-58449-8, 1994, pp. 141–155.
[49] A. Mostefaoui, M. Raynal, F. Tronel, From binary consensus to multivalued consensus in asynchronous message-passing systems, Inf. Process. Lett. (ISSN 0020-0190) 73 (5–6) (Mar. 2000) 207–212, https://doi.org/10.1016/S0020-0190(00)00027-2.
[50] E. Ruppert, The anonymous consensus hierarchy and naming problems, in: Proceedings of the 11th International Conference on Principles of Distributed Systems, OPODIS'07, Springer-Verlag, Berlin, Heidelberg, ISBN 3-540-77095-X, 2007, pp. 386–400, ISBN 978-3-540-77095-4, http://dl.acm.org/citation.cfm?id=1782394.1782422.
[51] M. Saks, F. Zaharoglou, Wait-free k-set agreement is impossible: the topology of public knowledge, SIAM J. Comput. (ISSN 0097-5397) 29 (5) (Mar. 2000) 1449–1483, https://doi.org/10.1137/S0097539796307698.
[52] D.J. Sorin, M.D. Hill, D.A. Wood, A Primer on Memory Consistency and Cache Coherence, 1st edition, Morgan & Claypool Publishers, ISBN 1608455645, 2011, ISBN 9781608455645.
[53] M. Yamashita, T. Kameda, Leader election problem on networks in which processor identity numbers are not distinct, IEEE Trans. Parallel Distrib. Syst. (ISSN 1045-9219) 10 (9) (Sept. 1999) 878–887, https://doi.org/10.1109/71.798313.
[54] J. Yang, G. Neiger, E. Gafni, Structured derivations of consensus algorithms for failure detectors, in: Proceedings of the Seventeenth Annual ACM Symposium on Principles of Distributed Computing, PODC'98, ACM, New York, NY, USA, ISBN 0-89791-977-7, 1998, pp. 297–306.