On the Uncontended Complexity of Anonymous Consensus *

Claire Capdevielle[1], Colette Johnen[2], Petr Kuznetsov[3], and Alessia Milani[4]

1 **Univ. Bordeaux, LaBRI, UMR 5800, F-33400 Talence, France**
claire.capdevielle@labri.fr
2 **Univ. Bordeaux, LaBRI, UMR 5800, F-33400 Talence, France**
johnen@labri.fr
3 **Télécom ParisTech, Paris, France**
petr.kuznetsov@telecom-paristech.fr
4 **Univ. Bordeaux, LaBRI, UMR 5800, F-33400 Talence, France**
milani@labri.fr

Abstract

Consensus is one of the central distributed abstractions. By enabling a collection of processes to agree on one of the values they propose, consensus can be used to implement any generic replicated service in a consistent and fault-tolerant way.

In this paper, we study *uncontended* complexity of anonymous consensus algorithms, counting the number of memory locations used and the number of memory updates performed in operations that encounter no contention. We assume that contention-free operations on a consensus object perform "fast" reads and writes, and resort to more expensive synchronization primitives, such as CAS, only when contention is detected. We call such concurrent implementations *interval-solo-fast* and derive one of the first nontrivial tight bounds on space complexity of anonymous interval-solo-fast consensus.

1998 ACM Subject Classification F.1.1 Models of Computation

Keywords and phrases space and time complexity, lower bounds, consensus, interval contention, solo-fast

1 Introduction

Consensus is one of the central distributed abstractions. By enabling a collection of processes to agree on one of the values they propose, consensus can be used to implement any generic replicated service in a consistent and fault-tolerant way. Therefore, complexity of consensus implementations has become one of the most important topics in the theory of distributed computing.

It is known that consensus cannot be solved in an asynchronous read-write shared memory system in a deterministic and fault-tolerant way [7, 16]. The difficulty stems from handling contended executions. One way to circumvent this impossibility is to only guarantee progress (using reads and writes) in executions meeting certain conditions, e.g., in the absence of *contention*. Alternatively, a process is guaranteed to decide in the *wait-free*

* Partially supported by the ANR project DISPLEXITY (ANR-11-BS02-014). This study has been carried out in the frame of the Investments for the future Programme IdEx Bordeaux-CPU (ANR-10-IDEX-03-02). The third author was supported by the ANR project DISCMAT, under grant agreement N ANR-14-CE35-0010-01.

manner, but stronger (and more expensive) synchronization primitives, such as *compare-and-swap*, can be applied in the presence of contention.

We are interested in consensus algorithms in which a *propose* operation is allowed to apply primitives other than reads and writes on the base objects only in the presence of *interval contention*, i.e., when another *propose* operation is concurrently active. These algorithms are called *interval-solo-fast*.

Ideally, interval-solo-fast algorithms should have an optimized behavior in *uncontended* executions. It appears therefore natural to explore the uncontended complexity of consensus algorithms: how many memory operations (reads and writes) need to be performed and how many distinct memory locations need to be accessed in the absence of interval contention?

In general, interval-solo-fast consensus can be solved with only constant uncontended complexity [17]. We therefore restrict our study to *anonymous* consensus algorithms, i.e., algorithms not using process identifiers and, thus, programming all processes identically. Besides intellectual curiosity, practical reasons to study anonymous algorithms in the shared memory model are discussed in [10].

Our results. On the lower-bound side, we show that any anonymous interval-solo-fast consensus algorithm exhibits non-trivial uncontended complexity that depends on n, the number of processes, and m, where m is the size of the set V of input values that can be proposed. More precisely, we show that, in the worst case, a *propose* operation running *solo*, i.e., without any other process invoking *propose*, must write to $\Omega(\min(\sqrt{n}, \log m/\log\log m))$ distinct memory locations. This metrics, which we call *solo-write complexity*, is upper-bounded by the *step complexity* of the algorithm, i.e., the worst-case number of all base-object primitives applied by an individual operation. In the special case of *input-oblivious* algorithms, where the sequence of memory locations written in a solo execution does not depend on the input value, we derive a stronger lower bound of $\Omega(\sqrt{n})$ on solo-write complexity. Our proof only requires the algorithm to ensure that operations terminate in solo executions, so the lower bounds also hold for *abortable* [2, 11] and obstruction-free [13] consensus implementations.

On the positive side, we show that our lower bound is tight. Our matching consensus algorithm is based on our novel *value-splitter* abstraction, extending the classical *splitter* mechanism [15, 18, 4], interesting in its own right. Informally, a value-splitter exports a single operation *split* that takes a value in a value set V as a parameter and returns a boolean response so that (1) if *split*(v) completes before any other *split* operation starts, then it returns *true*, and (2) all processes that obtain *true* proposed the same value.

We describe a simple transformation of a value-splitter into anonymous and interval-solo-fast consensus, using the classical splitter-based algorithm and incurring constant overhead with respect to the value-splitter complexity [17]. Then, we present two value-splitter read-write implementations that combined with the consensus algorithm provide the matching upper bound $O(\min(\sqrt{n}, \log m/\log\log m))$.

The first one is a novel anonymous and input-oblivious implementation of a value-splitter that exhibits $O(\sqrt{n})$ space and solo-write complexity.

The second one is not input-oblivious, and is a slight modification of the *weak conflict detector* proposed in [1], exhibiting $O(\log m/\log\log m)$ space and step complexity.

Our results are summarized in Table 1. It is interesting to notice that the step complexities are $O(n)$ for the first algorithm and $O(\log m/\log\log m)$ for the second one. Aspnes and Ellen [1] showed that any anonymous consensus protocol has to execute $\Omega(\min(n, \log m/\log\log m))$ steps in solo executions. Thus, our consensus algorithms have also asympotically optimal step complexity.

Input-oblivious	Not input-oblivious	
$\Omega(\sqrt{n})$	$\Omega(\min(\sqrt{n}, \frac{\log m}{\log\log m}))$	
$O(\sqrt{n})$	if $\sqrt{n} \leq \frac{\log m}{\log\log m}$, $O(\sqrt{n})$	if $\sqrt{n} \geq \frac{\log m}{\log\log m}$, $O(\frac{\log m}{\log\log m})$ [17, 1]

Table 1 Space and solo-write complexity for anonymous interval-solo-fast consensus

Overall, our results imply one the first nontrivial *tight* lower bound on the space complexity for consensus known so far, along with a concurrent result on the space complexity of *solo-terminating* anonymous consensus [8].[1] Our results also show that there is an inherent gap between anonymous and non-anonymous consensus algorithms: non-anonymous consensus has constant uncontended complexity [17].

Related work. The idea of optimizing concurrent algorithms for uncontended executions was suggested by Lamport in his "fast" mutual exclusion algorithm [15].

Fich et al. [6] have shown that any solo-terminating (and, as a result, obstruction-free) read-write (non-anonymous) consensus protocol must use $\Omega(\sqrt{n})$ memory locations. Gelashvili [8] proved a stronger $\Omega(n)$ lower bound for the anonymous case. Attiya et al. [2] showed that any *step-solo-fast* (where operations only apply reads and writes in the absence of interleaving steps) either use $O(\sqrt{n})$ space or incur $O(\sqrt{n})$ memory *stalls* per operation. No obstruction-free or step-solo-fast algorithm matching these lower bounds is known so far: existing algorithms typically expose $O(n)$ space complexity. These lower bounds focus on *step contention* and do not extend to uncontended executions, where no interval contention is encountered.

Our *value-splitter* abstraction is inspired by the splitter mechanism in [18, 4], originally suggested by Lamport [15]. Differently from the original splitter object, more than one process can return *true* but all these processes have the same input value. The novel input-oblivious value-splitter implementation we present is inspired by the obstruction-free leader election algorithm recently proposed by Giakkoupis et al. [9].

Bouzid et al. [3] presented an anonymous consensus algorithm with asymptotically optimal solo write and step complexity. But it relies on a failure detector (can be transformed into obstruction-free though) and requires *unbounded* space.

A preliminary version of this paper has been presented as a brief announcement [5].

Roadmap. The rest of the paper is organized as follows. We give preliminary definitions in Section 2. We present our lower bound in Section 3 and our upper bound in Section 4. We conclude the paper in Section 5.

2 Preliminaries

The model of computation.

We consider a standard asynchronous shared-memory model in which $n > 1$ processes communicate by applying atomic (or linearizable [14]) *primitive* operations on shared variables, called *base objects*. We assume every base object maintains a *state* and exports a subset of

[1] Informally, a solo-terminating algorithm ensures that every process running solo from any configuration eventually terminates.

the *Read*, *Write* and *Compare-And-Swap* (CAS) primitives. The primitive *Read*(R) returns the value of R, and *Write*(R, v) sets the state of R to v. The primitive *CAS*(R, e, v) checks if the state of R is e and, if so, sets the state of R to v and returns *true*; otherwise, the state remains unchanged and *false* is returned. A *register* is a base object that exports only the Read and Write primitives.

Algorithms and executions.

To implement a (high-level) object from a set of base objects, processes follow an *algorithm* $\mathcal{A}$, associating each process p with an deterministic automaton $\mathcal{A}_p$. To avoid confusion between the base objects and the implemented one, we reserve the term *operation* for the object being implemented and we call *primitives* the operations on base objects. We say that an operation is *performed* on a high-level object and that a primitive is *applied* to a base object.

Each process has a local state that consists of the values stored in its local variables and a programme counter. A computation of the system proceeds in *steps* of an algorithm performed by the processes. Each step is one of the following: (1) an invocation of a high-level *operation*, (2) a primitive operation on a base object that returns a response and results in a change of a process's state, or (3) a response of a (high-level) operation. A *configuration* specifies the state of each base object and the local state of each process at one moment. In an *initial configuration*, all base objects have the initial values specified by the algorithm and all processes are in their initial states.

A process is *active* if an operation has been invoked by the process but the operation has not yet produced a matching response; otherwise the process is called *idle*. We assume that an operation can only be invoked on an idle process and only active processes take steps. A configuration is *quiescent* if every process is idle in it.

An *execution fragment* of an algorithm is a (possibly infinite) sequence $C_1, \phi_1, \ldots, C_i, \phi_i, \ldots$ of configurations alternating with steps, where each step is the application of a primitive ϕ_i to configuration C_i resulting in configuration C_{i+1}. For any finite execution fragment α ending with configuration C and any execution fragment α' starting at C, the execution $\alpha\alpha'$ is the concatenation of α and α'; in this case α' is called an *extension* of α. An *execution* is an execution fragment starting from the initial configuration C_0.

In an infinite execution, a process is *correct* if it takes an infinite number of steps or is idle from some point on. Otherwise, the process is called *crashed*.

In a *solo* execution, only one process takes steps. An operation invoked by a process in a given execution is *completed* if its invocation is followed by a matching response. An operation invoked a process p in an execution E is *uncontended* if no process other than p is active between its invocation and response steps. We also say that p executes its operation in absence of *interval contention*.

Finally, we say that an operation executes in the absence of *step contention* if all the steps of the operation are contiguous in the execution.

Consensus.

The *consensus* object exports one operation *propose*(v), where v is an *input* taken from some domain V ($|V| \geq 2$). The output values must satisfy the following properties:

- *Agreement*: all output values are the same
- *Validity*: Every output value is one of the input values.

Properties of algorithms.

An algorithm is *wait-free* if in every execution, each correct process completes each of its operation in a finite number of its own steps [12].

A wait-free algorithm is *interval-solo-fast* if, in absence of interval contention, a process only applies Read and Write primitives. A wait-free algorithm is *step-solo-fast* [2] if a process is allowed to apply only Reads and Writes in the absence of *step contention*, i.e.. when its steps are not interleaved with the steps of another process.

An algorithm is *input-oblivious* if a process accesses the same sequence of base objects in any solo execution of the algorithm, regardless of its input.

An algorithm $\mathcal{A}$ is *anonymous* if $\mathcal{A}_p$ does not depend on p, i.e., the algorithm programs the processes identically, regardless of their identifiers.

In this paper we are concerned with two complexity metrics: *space complexity*, i.e., the number of base objects an algorithm uses, and *solo-write complexity*, i.e., the maximal number of writes performed in a solo execution of a single operation of an algorithm, taken over all possible input values. Note that solo-write complexity is upper-bounded by the *step complexity* of the algorithm, i.e., the number of base-object accesses a single operation may perform.

3 Lower bounds for interval-solo-fast consensus

Consider any n-process anonymous implementation of interval-solo-fast consensus with a set V of input values, $|V| = m$. In this section, we show that the implementation must have an execution in which some propose operation, running solo, performs $\Omega(\min(\sqrt{n}, \log m/\log\log m))$ writes on distinct objects. Obviously, the implementation must use $\Omega(\min(\sqrt{n}, \log m/\log\log m))$ base objects.

We also show that in the special case when the algorithm is, additionally, *input-oblivious* the lower bounds become $\Omega(\sqrt{n})$.

Overview of the proof.

By the way of contradiction, assume that there exists an interval-solo-fast anonymous consensus algorithm $\mathcal{A}$ such that at most k distinct base objects are written in any solo execution of $\mathcal{A}$ and $k < \min(\sqrt{n}, \Gamma^{-1}(m))$. Here Γ^{-1} is the inverse of the factorial function $\Gamma(m) = m!$. Recall that $\Gamma^{-1}(m) = \Theta(\log m/\log\log m)$.

We are going to establish a contradiction by showing that the algorithm has an execution in which two different values are returned. In executions we are going to iteratively construct, no process encounters interval contention and, thus, no process applies primitives other than Reads and Writes.

Let C_0 be the initial configuration of $\mathcal{A}$. For each $v \in V$, let α_v denote the execution of $\mathcal{A}$ in which a process, starting from C_0, invokes *propose*(u) and runs solo and until the operation completes. Since the algorithm is anonymous, α_v does not depend on the process identifier.

For a given $v \in V$, consider the sequence of base objects written in α_v, ordered by the times they are *first written* in α_v. There are m possible values v (and, thus, possible executions α_v), and at most $k!$ possible orders in which base objects can be written for the first time in executions α_v, $v \in V$.

Since $k < \Gamma^{-1}(m)$, we have $k! < m$ and, thus, there must be two values v and w such that the sequences of base objects written in α_v and α_w, in the order of the times they are

first written, are identical. (In an input-oblivious protocol, v and w can be any two distinct values, regardless of the relation between m and k.) Let us denote this sequence of base objects by $r_1, \ldots, r_k$ and fix it for the rest of the proof.

To construct the desired execution with different returned values and establish a contradiction, we assume that half of the processes propose v and the other half propose w. In each iteration of the construction, we "wake up" a subset of the processes in each of the two halves and let them run as *clones*, i.e., run them lock-step so that they ignore the presence of each other, until they are about to write to a base object for the first time. On the way, we carefully maintain the invariant that each previously written base object is *covered* by "enough" processes in each of the two halves: a process p *covers a base object* r *in a given configuration* C if p is about to write to r in C. Intuitively, these "covering" write operations, once applied, ensure that one half of the processes will not be able to "notice" the presence of the other half in an extended execution. As a result, in the subsequent iteration, we can extend the execution in a way that "enough" processes in each of the two halves cannot distinguish it from a solo run.

Using the assumption $k < \sqrt{n}$, we ensure that at the end of the kth iteration, we have at least one process p_i proposing v and at least one process p_j proposing w, and both p_i and p_j believe that they run solo. Moreover, in the resulting configuration C_k, each of the k base objects $r_1, \ldots, r_k$ is covered by at least one process proposing v with the value last written to it by p_i and at least one process proposing w with the value last written to it by p_j. Therefore, we can extend C_k with a block write of the processes proposing v and then let p_i run until completion, without being able to distinguish the resulting execution from α_v. Thus, p_i must eventually return v. But then we can extend the resulting execution with a block write of the processes proposing $w$and let p_j run until completion, without being able to distinguish the current execution from α_w. Thus, p_j will have to return w, which establishes the contradiction.

Notations and definitions.

We now introduce some instrumental notions and definition.

Recall that α_u denotes the complete solo execution of *propose*(u) from the initial configuration C_0. For $u = v, w$, $1 \leq i \leq k$, let $\alpha_{i,u}$ denote the longest prefix of α_u which only contains writes on base objects in $\{r_1, \ldots, r_i\}$. Let $\alpha_{0,u}$ denote the longest prefix of α_u in which no writes takes place. By the definition, $\alpha_{k,u} = \alpha_u$, and for all $0 \leq i \leq k-1$, the next event of α_u immediately after $\alpha_{i,u}$ is a write on r_{i+1}.

For $j = 1, \ldots, k$, let $x_{j,i,u}$ denote the value of r_j in the configuration right after $\alpha_{i,u}$. Recall that for $j = i+1, \ldots, k$, no write on r_j takes place in $\alpha_{i,u}$ and, thus, $x_{j,i,u}$ is the initial value of r_j.

For $i, j = 1, \ldots, k$ and $u = v, w$, let $I_{j,i,u}$ be a binary indicator that r_j is written in $\alpha_{i,u}$ *after* the last event of $\alpha_{i-1,u}$. Note that $I_{i,i,u} = 1$ for all $i = 1, \ldots, k$, and $I_{j,i,u} = 0$ for all $1 \leq i < j \leq k$.

For example, consider the solo execution of a *propose*(u) operation depicted in Figure 1. Here, $I_{1,2,u}$ is the binary indicator that r_1 is written in $\alpha_{2,u}$ *after* the last event of $\alpha_{1,u}$, i.e., in the execution fragment δ. If there is a write on r_1 in δ then $I_{1,2,u} = 1$. Otherwise, $I_{1,2,u} = 0$.

For $1 \leq i, j < k$, we define $s_{j,i,u}$ as 1 plus the maximal number of *consecutive* prefixes $\alpha_{t,u}$ such that $i < t < k$ and r_j is *not* written in $\alpha_{t,u}$ after the last event of $\alpha_{t-1,u}$, i.e., $s_{j,i,u} = \min\{\ell > i | \ell = k \vee I_{j,\ell,u} = 1\} - i$.

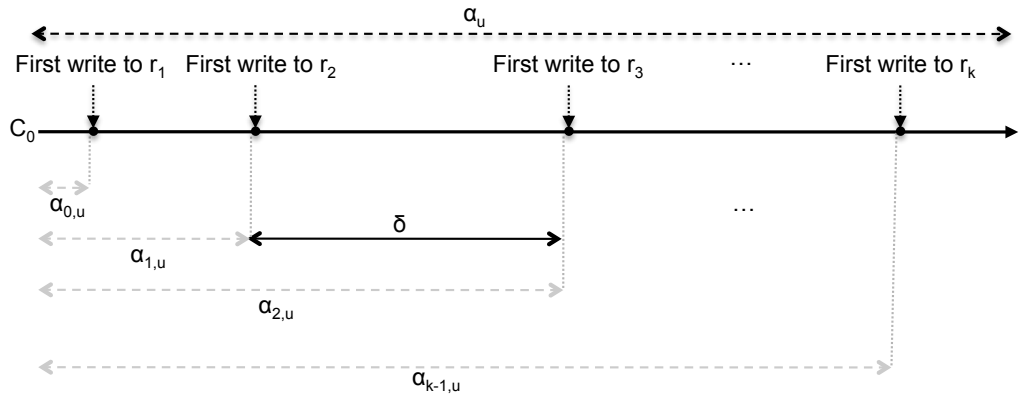

Figure 1 Solo execution of *propose*(u) by a process p, denoted α_u.

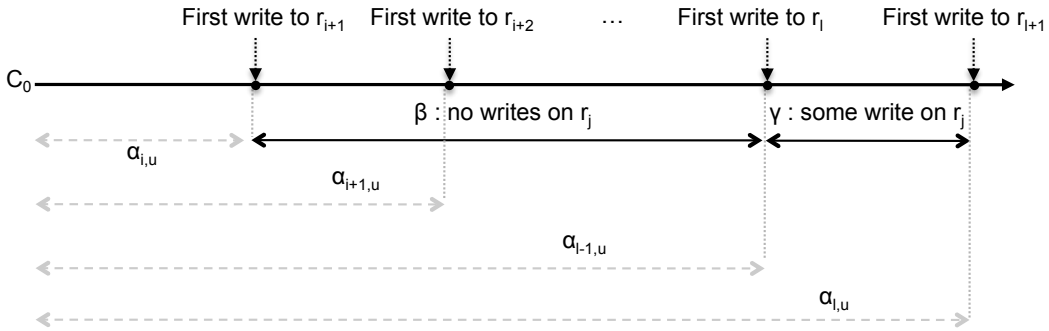

Figure 2 Definition of $s_{j,i,u}$: β contains $\ell - i$ consecutive fragments in which r_j is not written.

Figure 2 depicts a fragment of the execution α_u and graphically explains the notation $s_{j,i,u}$. In particular, for a given base object r_j and a given prefix $\alpha_{i,u}$, we consider the longest sequence of consecutive distinct fragments between the first write to r_t up to (but not including) the first write to r_{t+1}, which contain no writes on r_j, starting from $t = i+1$. This sequence of fragments is denoted by β here. Then $s_{j,i,u}$ is simply the number of consecutive fragments in β plus one, i.e., $\ell - i$ in this case, as the fragment γ between the first write to r_ℓ up to the first write to $r_{\ell+1}$ contains a write to r_j.

Clearly, $s_{j,i,u} \geq 1$ and $s_{j,k-1,u} = 1$, for all $i, j = 1, \ldots, k-1$. Also, it is easy to check that $\sum_{i=1}^{k-1} I_{j,i,u} s_{j,i,u} = k - j$ for all $j = 1, \ldots, k-1$. Thus, $\sum_{\ell=1}^{k-1} \sum_{j=1}^{k-1} I_{j,\ell,u} s_{j,\ell,u} = (k^2 - k)/2$.

Cloning configurations.

We now introduce the central notion of our lower-bound proof:

▶ **Definition 1.** A configuration C_i is called *i-cloning*, $1 \leq i \leq k$, if it satisfies the following conditions:

- For each $u = v, w$, $j = 1, \ldots, i-1$, r_j is covered by $s_{j,i-1,u}$ processes writing $x_{j,i-1,u}$.
- For each $u = v, w$, there are at least $(k^2 - k + 2)/2 - \sum_{\ell=1}^{i-1} \sum_{j=1}^{i-1} I_{j,\ell,u} s_{j,\ell,u}$ processes that do not distinguish the execution from $\alpha_{i-1,u}$ and, thus, cover base object r_i with value $x_{i,i,u}$.
- Each base object in $\{r_i, \ldots, r_k\}$ stores the initial value.

▶ **Lemma 2.** *Let $\mathcal{A}$ be any n-process m-valued interval-solo-fast anonymous consensus algorithm. If at most k distinct base objects are written in any solo execution of $\mathcal{A}$, where $k < \min(\sqrt{n}, \Gamma^{-1}(m))$, then $\mathcal{A}$ has a k-cloning configuration.*

Proof. By induction on k, we construct a k-cloning configuration starting from the initial configuration C_0 of $\mathcal{A}$.

We divide the processes in two groups of size at most $(k^2-k+2)/2$ where every process in one group proposes value v and every process in the other group proposes value w. This is possible, since $k < \sqrt{n}$.

Base case. Let γ be the concatenation of executions of $\mathcal{A}$ in which, starting at C_0, a process runs in isolation until it is about to write to base object r_1 for the first time. Recall that r_1 is the first base object written in both α_v and α_w, so no process can distinguish γ from its solo execution and, thus, *gamma* is indeed an execution of $\mathcal{A}$.

It is easy to see that, since no process writes in γ, $C_1 = C_0\gamma$ is a 1-cloning configuration. Indeed, half of the processes cannot distinguish $C_0\gamma$ from $\alpha_{0,v}$ and the other half from $\alpha_{0,w}$, and all base objects are in their initial states.

As an *induction hypothesis*, consider an i-cloning configuration C_i, for some $1 \leq i < k$.

For each $u \in \{v, w\}$ we then perform the following procedure.

First for each $j = 1, \ldots, i-1$, we let one of the processes covering r_j with $x_{j,i-1,u}$ complete its write. By the induction hypothesis, there are at least $s_{j,i-1,u} \geq 1$ such processes.

Then we wake up $(k^2-k+2)/2 - \sum_{j=1}^{i-1}\sum_{\ell=1}^{i-1} I_{j,\ell,u}s_{j,\ell,u}$ processes that cannot distinguish the execution from $\alpha_{i-1,u}$ and run them *lock-step* (without noticing each other) until they are about to perform their write on r_{i+1}. No such process can distinguish the execution from $\alpha_{i,u}$, and thus, we indeed obtain an execution of $\mathcal{A}$. If $\alpha_{i,u}$ contains a write on some r_m, $m = 1, \ldots, i$, after the last event of $\alpha_{i-1,u}$, then $s_{m,i,u}$ of these processes are stopped just before they perform the *last* write on r_m in $\alpha_{i+1,u}$. This can be done because $I_{m,i,u} = 1$ for every such m and $\sum_{\ell=1}^{i}\sum_{j=1}^{i-1} I_{j,\ell,u}s_{j,\ell,u} \leq \sum_{\ell=1}^{k-1}\sum_{j=1}^{i-1} I_{j,\ell,u}s_{j,\ell,u} = (k^2-k)/2 < n/2$.

Let γ be the resulting extension of C_i and $C_{i+1} = C_i\gamma$ be the resulting configuration. Notice that all base objects in $\{r_{i+1}, \ldots, r_k\}$ still store the initial value in C_{i+1}.

Now consider any $j = 1, \ldots, i$ and $u = v, w$. If r_j is not written in $\alpha_{i+1,u}$, then, by the induction hypothesis and the construction of γ, r_j is covered by $s_{j,i,u} = s_{j,i-1,u} - 1$ processes writing $x_{j,i,u} = x_{j,i-1,u}$. Otherwise, by construction, r_j is covered by $s_{j,i,u}$ processes writing $x_{j,i,u}$.

Finally, for each $u \in \{v, w\}$, since $\sum_{j=1}^{i} I_{j,i,u}s_{j,i,u}$ additional processes are used to cover base objects $r_1, \ldots, r_i$, at least $(k^2-k+2)/2 - \sum_{\ell=1}^{i}\sum_{j=1}^{i} I_{j,\ell,u}s_{j,\ell,u}$ remaining processes cannot distinguish $C_i\gamma$ from $C_0\alpha_i$ and, thus, these processes must cover r_{i+1}.

Hence, C_{i+1} is $(i+1)$-cloning, and, by induction, $\mathcal{A}$ has a k-cloning configuration. ◀

▶ **Theorem 3.** *Any n-process m-valued interval-solo-fast anonymous consensus algorithm must have space complexity $\Omega(\min(\sqrt{n}, \log m/\log\log m))$ and solo-write complexity $\Omega(\min(\sqrt{n}, \log m/\log\log m))$. Moreover, if the algorithm is input-oblivious, then the bounds become $\Omega(\sqrt{n})$.*

Proof. Suppose, by contradiction, that an n-process m-valued interval-solo-fast anonymous consensus algorithm uses k base objects such that $n = k^2 - k + 2$ and $k < \Gamma^{-1}(m)$.

By Lemma 2, there exists a k-cloning configuration C_k for some input values v and w. Note that in C_k, for each $u \in \{v, w\}$, every base object r_j, $j = 1, \ldots, k$, is covered by exactly $s_{j,k-1,u} = 1$ process writing value $x_{j,k-1,u}$. Also, exactly $n/2 - \sum_{\ell=1}^{k-1}\sum_{j=1}^{k-1} I_{j,\ell,u}s_{j,\ell,u} = \sum_{j=1}^{k-1}(k-j) = k(k-1)/2 + 1 - k(k-1)/2 = 1$ process cannot distinguish the execution from $\alpha_{k-1,u}$ and, thus, this process must cover r_k with $x_{k,k,u}$.

Now we take $u \in \{v, w\}$, and let the single process covering r_j, $j = 1, \ldots, k-1$ with value $x_{j,k-1,u}$ perform its write. Then we let the single process proposing u and covering r_k run solo. Notice that the process cannot distinguish the execution from $\alpha_{k,u}$ and, thus, it should eventually terminate by outputting value u.

In the resulting execution two different input values v and w are decided, implying a contradiction.

Thus, since $\Gamma^{-1}(m) = \Theta(\log m / \log\log m)$, the algorithm has a solo execution in which $\Omega(\min(\sqrt{n}, \log m / \log\log m))$ distinct base objects are written. Moreover, if the algorithm is input-oblivious, then a k-cloning configuration exists for any two values u and w, and the lower bounds become $\sqrt{n}$. ◀

Remark 1. Lemma 2 shows that having at least $k^2 - k + 2$ processes is sufficient to construct a k-cloning configuration and, thus, establish a contradiction. The lower bound can be refined to $(k^2 - k)/2 + 2$ if we alternate the executions of processes proposing v with the executions of processes proposing w in each iteration of the inductive construction of C_k. Indeed, if processes proposing w were the last to execute in the construction of C_i, then every base object r_j, $j = 1, \ldots, i-1$ stores $x_{j,i-1,w}$, so in the next iteration, we may run processes proposing w first without the need to use the processes covering r_j with $x_{j,i-1,w}$. This allows us to spare half of the covering processes, implying $(k^2 - k)/2 + 2$ processes in total, which makes k closer to the upper bound $\sqrt{2n}$ we present in the next section. For the sake of simplicity, we chose to show the rougher (but asymptotically equivalent) lower bound.

4 Optimal interval-solo-fast consensus

In this section we present an algorithm that implements an interval-solo-fast consensus. This algorithm is similar to the *splitter-based* consensus algorithm in [17], except that we replace the `splitter` object with the `value-splitter` object that we introduce in this paper.

Value-splitters

A splitter provides processes with a single operation *split*() that returns a boolean response, so that (i) if a process runs solo, it must obtain *true* and (ii) *true* is returned to at most one process. A value-splitter exports a single operation *split*(v) ($v \in V$, for some input domain V) and relaxes property (ii) of splitters by allowing *multiple* processes to obtain *true* as long as they have the same input value. More precisely:

▶ **Definition 4.** A value-splitter supports a single operation, *split*() taking a parameter in V and returning a boolean response, and ensures that, for all $v, v' \in V$, and in every execution:

1. **VS-Agreement.** If invocations *split*(v) and *split*(v') return *true*, then $v = v'$.
2. **VS-Solo execution.** If a *split*(v) operation completes before any other *split*(v') operation is invoked, then it returns *true*.

We use a value-splitter object to construct an anonymous consensus algorithm. The algorithm incurs only a constant overhead with respect to the implementation of the value-splitter it uses and is interval-solo-fast assuming that the underlying value-splitter is interval-solo-fast.

Then we describe two anonymous interval-solo-fast implementations of a value-splitter. The first one is input-oblivious and exhibits $O(\sqrt{n})$ solo-write and space complexity, regardless of the number m of possible inputs. The second one exhibits complexities

$O(\log m / \log\log m)$, regardless of the number of processes n. The two algorithms provide a matching upper bound to our $\Omega(\min(\sqrt{n}, \log m / \log\log m))$ lower bound.

4.1 Consensus using value-splitter

The pseudocode of our consensus algorithm is given in Algorithm 1. The value decided by the consensus is written in a variable D, initially $\perp \notin V$. The first steps by a process p are to check if D stores a non-$\perp$ value and if yes, return this value. Otherwise, the process accesses the value-splitter object VS.

If it obtains *true* from its invocation of $VS.split(v)$, p writes its input value v in a register F. Then, it reads a register Z to check if some other process has detected contention and if the value of Z is *false* (no contention) p decides its own value. Before returning the decided value, process p writes it in D. The write primitives on F and D, with a read of Z in between are intended to ensure that either process p detects that some other process is around and resorts to applying a CAS primitive on D, or the contending process adopts the input value of p.

If p obtains *false* from the value-splitter, it sets Z to *true* (contention is detected). Recall that this may happen if more than one process accessed the value-splitter, regardless of their input values. Then, p reads register F and, if F stores a non-$\perp$ value, adopts the value as its current proposal. Finally, it applies the CAS primitive on D with its proposal and decides the value read in D.

Notice that, assuming that the value-splitter is interval-solo-fast, a process running in the absence of interval contention reaches a decision applying only reads and writes.

```
   Shared variables:
   D, F, initially ⊥
   Z, initially false
   value-splitter VS
   Procedure: propose(v)
 1 if (t := Read(D)) ≠ ⊥ then return t
 2 if VS.split(v) then
 3 |   Write(F, v);
 4 |   if ¬(Read(Z)) then
 5 |   |   Write(D, v);
 6 |   |   return v
 7 |   end
 8 else
 9 |   Write(Z, true);
10 |   if (t := Read(F)) ≠ ⊥ then v := t;
11 end
12 CAS(D, ⊥, v);
13 res := Read(D);
14 return res
```

Algorithm 1: Interval-solo-fast consensus

In the following we prove that Algorithm 1 indeed implements interval-solo-fast consensus, assuming that VS is an interval-solo-fast implementation of a value-splitter. We show that such implementations exist in the next subsection.

Proofs of Algorithm 1

▶ **Lemma 5** (Agreement). *No two processes return different values.*

Proof. Given that only values written to D can be returned, it is sufficient to show that at most one value can be written in D.

By the algorithm D is updated in lines 12 and 5. Note that, since a CAS succeeds in updating the value of D in line 12 only if D stores $\perp$ and, since D is updated with a non-$\perp$ value in V, at most one process may succeed. D is updated at line 5 only if the corresponding process obtains *true* from the value-splitter. By the VS-Agreement property of value-splitters, at most one distinct value can be written in D in line 5.

Thus, the only possibility for two different values to be written in D is when one process , say p, applies a CAS in line 12 and updates D with a value v and another process writes $v' \neq v$ in D in line 5.

Note that p must have obtained *false* from the value-splitter, otherwise it would try to update D with value v. Thus, before applying CAS on D, p has read F in line 10. We establish the contradiction by showing that p must have necessarily read v' in F and adopt it as its preferred value (line 10).

By the VS-Agreement property of value-splitters, at most one non-$\perp$ value can be found in F. Thus, since q has written v' to F in line 3, the only possible case is that p reads F before any other process writes to it. But then p has previously set the "contention flag" Z to *true* in line 9. Therefore, after q writes v' in F it must find Z set to *true* ("contention is detected") and resort to CAS instead of writing in D in line 5—a contradiction. ◀

▶ **Lemma 6** (Interval-solo-fast). *Any operation that runs in the absence of interval contention applies only reads and writes.*

Proof. If a process p invokes its *propose* operation and finds a non-$\perp$ value in D, then p returns after having applied a single read on D, so the claim follows.

Otherwise, suppose that p initially finds $D = \perp$ and applies the CAS primitive (line 12). We show that there is an operation that overlaps with the *propose* of p.

By inspecting the pseudo-code, it is easy to see that p applies the CAS primitive only if (1) it has read $Z = true$ (line 4) or (2) it has obtained *false* from VS. In both cases, by the VS-Solo Execution property, there must be another process q that has invoked $VS.split(v)$ before p has completed its *Propose* operation.

By the algorithm, before completing its operation, q writes its decided (non-$\perp$) value in D. Given that p has initially found $\perp$ in D, we deduce that the operation of q has not completed before the operation of p has started.

Thus, the two operations overlap. The assumption that the value-splitter is interval-solo-fast and the fact the algorithm contains no loops or waiting statements, implies the claim. ◀

Finally, we use Lemmata 5 and 6 to prove:

▶ **Theorem 7.** *If VS is an interval-solo-fast implementation of a value-splitter, then Algorithm 1 implements interval-solo-fast consensus with space complexity $O(k)$ and solo-write complexity $O(s)$, where k is the space complexity and s is the solo-write complexity of VS.*

The complexity claims follow directly from the pseudo-code.

4.2 Interval-solo-fast value-splitter implementations.

Input-oblivious value-splitter.

Algoritm 2 describes our anonymous and input-oblivious implementation of a value-splitter. The algorithm only uses an array R of k registers where $k^2 - 3k + 6 > 2n$ and is, trivially,

interval-solo-fast. Thus, by Theorem 3, the space complexity of the algorithm is asymptotically optimal.

In the algorithm, a process p performing operation $split(v)$ tries to write its input value to registers $R[0], \ldots, R[k-1]$. Each time, before writing to $R[i]$, p reads $i+1$ registers to verify that $R[0], \ldots, R[i-1]$ store v and $R[i]$ stores the initial value $\perp$. If this is not the case, contention is detected and the operation returns *false*. After the last write to $R[k-1]$, the operation returns *true*. Note that several processes proposing the same value and executing lock-step may return *true*.

```
   Shared variables:
   Array of registers R[0...k-1] with k^2 - 3k + 6 > 2n. Initially ⊥
   Procedure: split(v)
1  Lastwritten := -1;
2  while (Lastwritten ≤ k - 1) do
3  |  i := 0;
4  |  while (i ≤ Lastwritten) do
5  |  |  if Read(R[i]) ≠ v then return false
6  |  |  i++;
7  |  end
8  |  if Read(R[Lastwritten + 1]) ≠ ⊥ then return false;
9  |  Lastwritten++;
10 |  Write(R[Lastwritten], v);
11 end
12 return true;
```

Algorithm 2: Anonymous and input-oblivious value-splitter

Note also that the solo-write complexity of Algorithm 2 is $k = O(\sqrt{n})$. Since, for $i = 1$ to k, in the ith iteration, a process reads i registers, the algorithm also has optimal step complexity of $O(n)$ [1].

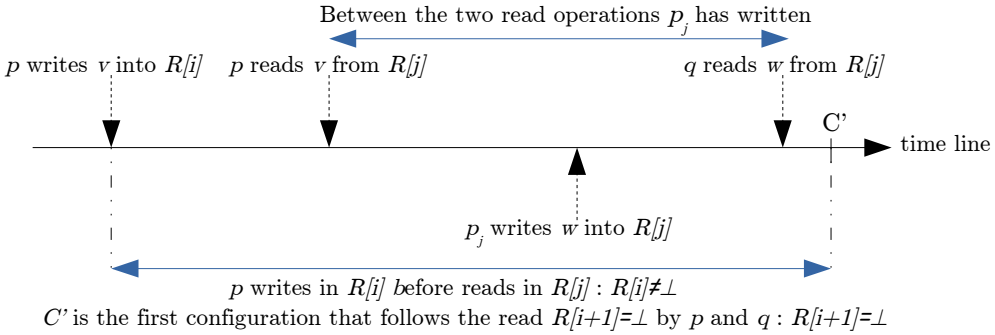

Figure 3 Execution for Lemma 8, assuming that p reads $R[j]$ before q

The following lemma will be instrumental in showing that Algorithm 2 satisfies the VS-Agreement property.

▶ **Lemma 8.** *If an execution E, two processes p and q write in $R[i]$ and $R[i+1]$ for some $0 < i < k-1$, two different values v and w, then there is a set P_i of at least i processes (different from p and q) and the following conditions are satisfied: (1) at the configuration that immediately succeeds the last write operation executed by processes in P_i, $R[i+1] = \perp$; (2) E passes through a configuration C such that $R[i] \neq \perp$ in C and each process in P_i*

executes exactly one write operation after C.

Proof. Fix an i such that $0 < i < k-1$ and let p and q be two processes that write, respectively, values v and w in both $R[i]$ and $R[i+1]$, where $v \neq w$.

By the pseudocode of Algorithm 2, before writing in $R[i+1]$, a process reads $R[0], R[1], \ldots, R[i+1]$, and the value it reads from $R[j]$ is its input value for $0 \leq j \leq i$ and the initial value for $j = i+1$.

Consider the sequences of read operations executed by p and q, respectively, after their write in $R[i]$ and before writing in $R[i+1]$. Let C' be the configuration immediately after both p and q perform their reads of $R[i+1]$ that return $\perp$ in E. By the algorithm, writes in $R[i+1]$ by both p and q follow C' in E.

Also, since for each $j = 0, \ldots, i-1$ p reads v in $R[j]$ and q reads w in $R[j]$, there is a process p_j that has written in $R[j]$ between these two read operations. We show that this is the last write of p_j. Indeed, before performing the next write (on $R[j+1]$), p_j reads all registers and in particular it will read $R[i]$, where $i > j$. Since the write by p_j follows the read on $R[j]$ either by process p or by process q, it follows the write into $R[i]$ by the corresponding process. Thus, in the configuration immediately before the write into $R[j]$ by p_j we have $R[i] \neq \perp$. The check in line 8 implies that p_j cannot write to any register after $R[j]$. Note that p_j must be different from p and q: otherwise, we contradict the fact that both p and q write in $R[i]$, $i > j$.

Finally, since the last write operation of p_j preceeds configuration C', at the configuration immediately after this write $R[i+1]$ stores the initial value. This is illustrated in Figure 3 for the case when p reads $R[j]$ before q. Moreover, for each $j, \ell \in \{0, 1, \ldots i-1\}$ with $j \neq \ell$, $p_j \neq p_l$. Thus, the set P_i of i processes p_j, $j = 1, \ldots, i-1$, satisfies the two conditions of the lemma. ◀

▶ **Lemma 9** (VS-Agreement). *If invocations split(v) and split(v') return true, then $v = v'$.*

Proof. Suppose, by contradiction, that *split*(v) invoked by process p and *split*(w) invoked by process q both return *true* with $v \neq w$. Recall that a process has to write its input value in all the registers to return *true*. Then for each $0 \leq i \leq k-1$, p and q have written in register $R[i]$ the value v and w respectively. For each $i = 1, \ldots k-2$, let P_i be the i processes, as defined in Lemma 8.

Consider any two set P_i, P_j, $0 < i < j < k-1$. We show that $P_i \cap P_j = \emptyset$. Indeed, by the definition of P_i, in the configuration when the processes in P_i have completed all their writes, $R[i+1]$ stores $\perp$ and, by the algorithm, since $j > i$, $R[j]$ also stores $\perp$. But, by the definition of P_j, each process in P_j has executed a write operation after a configuration where $R[j] \neq \perp$. Thus, P_i and P_j are disjoint.

Recall p and q write to $R[k-1]$ and, thus, do not belong to $\cup_{i=1}^{k-2} P_i$. Hence, we have at least $2 + \sum_{i=1}^{k-2} i = 2 + \frac{k^2-3k+2}{2}$ processes in total, which contradicts the hypothesis that $k^2 - 3k + 6 > 2n$. ◀

▶ **Theorem 10.** *Algorithm 2 is an interval-solo-fast anonymous input-oblivious implementation of a value-splitter with solo-write and space complexities in $O(\sqrt{n})$.*

Proof. Since only read-write registers are used, the algorithm is trivially interval-solo-fast.

By Lemma 9, the algorithm satisfies the *VS-Agreement* property. We prove in the following that the *VS-Solo execution* property is also satisfied. Consider any solo execution E in which a *split*(v) by a process p completes and suppose, by contradiction, that the operation returns $false$. By inspecting the pseudocode, it is easy to see that the value of *Lastwritten* is equal to the index of the last register p wrote or to -1 if no such writes exists. To return

false p has either read a value different from its input (line 5) or a value different from $\perp$ in a register p has not yet written (line 8). But this contradicts the fact that E is a solo execution. Thus, the algorithm satisfies the Solo-Execution property of value-splitters.

◀

Non-input-oblivious value-splitter.

For completeness, we briefly describe an anonymous value-splitter algorithm based on earlier work [1] that exhibits $O(\log m / \log \log m)$ complexity.

A trivial adaptation of the `weak conflict-detector` proposed in [1] implements an interval-solo-fast value-splitter. A weak conflict-detector exports a single operation *check*(v) with an input v and return *true* (conflict is detected) or *false* (no conflict is detected). If no two operations are invoked with different inputs, then no operation returns *true*, otherwise, at least one operation returns *false*.

Our value-splitter implementation presented in Algorithm 3 is obtained by the weak conflict-detector algorithm in [1], where the output is determined as the negation of the outcome of the weak conflict-detector.

Shared variables:
Registers $R[1..k]$, initially $\perp$

Procedure: *split*(v)
1 **for** $i := 1..k$ **do**
2 $\quad t := Read(R[\pi_v(i)])$;
3 $\quad$ **if** $t = \perp$ **then** $Write(R[\pi_v(i)], v)$;
4 $\quad$ **if** $t \neq v$ **then return** $false$;
5 **end**
6 **return** $true$;

Algorithm 3: Non-input-oblivious value-splitter

The algorithm uses an array R of k registers, where $k! = m$. Each input value v of a *split* operation determines a unique permutation π_v of the registers in R that is used as the order in which the processes access the registers. Therefore, the algorithm is not input-oblivious. In its i-th access, a process executing *split*(v) first reads register $R[\pi_v(i)]$; if $\perp$ is read, the process writes v to it; If a value $v' \neq v$ is read, it returns *false* (contention is detected). If the process succeeds in writing v in all registers prescribed by π_v, it returns *true*. The algorithm is also trivially anonymous and interval-solo-fast.

▶ **Theorem 11.** *Algorithm 3 implements anonymous interval-solo-fast m-valued value-splitter with solo-write and space complexity in $O(\log m / \log \log m)$.*

Proof. If an operation *split*(v) runs solo, then no value other than v can be found in any $R[\pi_v(i)]$ (line 2). Thus the *VS-Solo Execution* property is ensured.

Suppose, by contradiction, that two operations, *split*(v), performed by p_v, and *split*(v'), performed by $p_{v'}$, return *true*. Let j, ℓ be two indexes in $\{1, \ldots, k\}$ such that j appears before ℓ in π_v but ℓ appears before j in $\pi_{v'}$. By the algorithm, before returning *true*, p_v and $p_{v'}$ have read, respectively, v and v' in both $R[j]$ and $R[\ell]$.

Without loss of generality, let v be written to $R[j]$ before v' is written to $R[\ell]$. By the algorithm, before any process performing *split*(v') reads $R[j]$ in line 2 (and, thus, writes v' to $R[j]$ in line 3), v' has been written to $R[\ell]$, and, by the assumption, v has been written to $R[j]$. Hence, the process will not find $\perp$ in $R[j]$ and will not write to $R[\ell]$—a contradiction. Therefore, the algorithm satisfies the VS-Agreement property.

Since every operation performs k writes and k reads, where $k! = m$, the step and space complexities of the algorithm are $O(\log m / \log \log m)$. ◀

5 Concluding remarks

In this paper, we present matching lower and upper bounds $\Theta(\min(\sqrt{n}, \log m/\log\log m))$ on the space and solo-write complexity of anonymous interval-solo-fast consensus, which appears to be one of the first non-trivial tight bound for consensus, along with a concurrent result on the space complexity of solo-terminating anonymous consensus [8]. Given that *non-anonymous* interval-solo-fast algorithms can be achieved with only constant space and step complexities [17], our results exhibits a complexity gap between anonymous and non-anonymous consensus. The proof of our lower bound is based on constructing executions in which no process is aware of interval contention and, thus, the lower bounds also apply to *abortable* [2, 11] consensus algorithms, where operations are allowed to return a specific *abort* response when interval contention is detected, and be-reinvoked later. An interesting open question is whether a matching abortable consensus algorithm can be found.

References

1 J. Aspnes and F. Ellen. Tight bounds for adopt-commit objects. *Theory of Computing Systems*, 55(3):451–474, 2014.

2 H. Attiya, R. Guerraoui, D. Hendler, and P. Kuznetsov. The complexity of obstruction-free implementations. *J. ACM*, 56(4), 2009.

3 Z. Bouzid, P. Sutra, and C. Travers. Anonymous agreement: The janus algorithm. In *Principles of Distributed Systems - 15th International Conference, OPODIS 2011, Toulouse, France, December 13-16, 2011. Proceedings*, pages 175–190, 2011.

4 H. Buhrman, J. A. Garay, J.-H. Hoepman, and M. Moir. Long-lived renaming made fast. In *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '95, pages 194–203, 1995.

5 C. Capdevielle, C. Johnen, P. Kuznetsov, and A. Milani. Brief Announcement: On the Uncontended Complexity of Anonymous Consensus. In *DISC*, Oct. 2015.

6 F. Fich, M. Herlihy, and N. Shavit. On the space complexity of randomized synchronization. *J. ACM*, 45(5):843–862, Sept. 1998.

7 M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, Apr. 1985.

8 R. Gelashvili. On the Optimal Space Complexity of Consensus for Anonymous Processes. In *DISC*, Oct. 2015.

9 G. Giakkoupis, M. Helmi, L. Higham, and P. Woelfel. An o(sqrt n) space bound for obstruction-free leader election. In *Distributed Computing - 27th International Symposium, DISC 2013, Jerusalem, Israel, October 14-18, 2013. Proceedings*, pages 46–60, 2013.

10 R. Guerraoui and E. Ruppert. Anonymous and fault-tolerant shared-memory computing. *Distributed Computing*, 20(3):165–177, 2007.

11 V. Hadzilacos and S. Toueg. On deterministic abortable objects. In *Proceedings of the 2013 ACM Symposium on Principles of Distributed Computing*, PODC '13, pages 4–12, New York, NY, USA, 2013. ACM.

12 M. Herlihy. Wait-free synchronization. *ACM Trans. Prog. Lang. Syst.*, 13(1):123–149, Jan. 1991.

13 M. Herlihy, V. Luchangco, and M. Moir. Obstruction-free synchronization: Double-ended queues as an example. In *ICDCS*, pages 522–529, 2003.

14 M. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990.

15 L. Lamport. A fast mutual exclusion algorithm. *ACM Trans. Comput. Syst.*, 5(1):1–11, Jan. 1987.

16 M. Loui and H. Abu-Amara. Memory requirements for agreement among unreliable asynchronous processes. *Advances in Computing Research*, 4:163–183, 1987.

17 V. Luchangco, M. Moir, and N. Shavit. On the uncontended complexity of consensus. In F. Fich, editor, *Distributed Computing*, volume 2848 of *Lecture Notes in Computer Science*, pages 45–59. Springer Berlin Heidelberg, 2003.

18 M. Moir and J. H. Anderson. Wait-free algorithms for fast, long-lived renaming. *Sci. Comput. Program.*, 25(1):1–39, Oct. 1995.