A complexity-based classification for multiprocessor synchronization

Noname manuscript No.
(will be inserted by the editor)

A Complexity-Based Classification for Multiprocessor Synchronization

Faith Ellen · Rati Gelashvili · Nir Shavit · Leqi Zhu

Abstract For many years, Herlihy's elegant computability-based Consensus Hierarchy has been our best explanation of the relative power of various objects. Since real multiprocessors allow the different instructions they support to be applied to any memory location, it makes sense to consider combining the instructions supported by different objects, rather than considering collections of different objects. Surprisingly, this causes Herlihy's computability-based hierarchy to collapse.

In this paper, we suggest an alternative: a complexity-based classification of the relative power of sets of multiprocessor synchronization instructions, captured by the minimum number of memory locations of unbounded size that are needed to solve obstruction-free consensus when using different sets of instructions.

1 Introduction

An *object* is defined by a domain of values, together with a set of instructions that can be performed on these values. Herlihy's Consensus Hierarchy [Her91] assigns a *consensus number* to each object, namely, the number of processes for which there is a wait-free binary consensus algorithm using only instances of this object and read-write registers. It is simple, elegant and, for many years, has been our best explanation of synchronization power.

Robustness says that, using any collection of objects with consensus numbers at most k, it is not possible to solve wait-free consensus for more than k processes [Jay93]. The implication is that modern machines need to provide objects with infinite consensus number. Otherwise, they will not be universal, that is, they cannot be used to implement all objects or solve all tasks in a wait-free (or non-blocking) manner for any number of processes [Her91, Tau06, Ray12, HS12]. The Consensus Hierarchy is known to be robust when restricted to deterministic one-shot objects [HR00] or deterministic read-modify-write and readable objects [Rup00]. There are ingenious non-deterministic constructions that prove it is not robust [Sch97, LH00]. However, it is unknown whether the Consensus Hierarchy is robust for all deterministic objects.

Since real multiprocessors allow the instructions they support to be applied to arbitrary memory locations, it makes sense to combine the instructions supported by different objects, rather than just having a collection of different objects. For example, consider two simple instructions that can be performed on natural numbers:

- *fetch-and-add*(2), which returns the number stored in a memory location and increases its value by 2, and
- *test-and-set*(), which returns the number stored in a memory location and sets it to 1 if it contained 0. (This definition of *test-and-set* generalizes the domain of the standard definition from $\{0, 1\}$ to $\mathbb{N}$.)

Support is gratefully acknowledged from the Natural Science and Engineering Research Council of Canada, the National Science Foundation under grants CCF-1217921, CCF-1301926, and IIS-1447786, the Department of Energy under grant ER26116/DE-SC0008923, and Oracle and Intel corporations. The authors would like to thank Michael Coulombe, Dan Alistarh, Yehuda Afek, Eli Gafni and Philipp Woelfel for helpful conversations and feedback.

Faith Ellen
University of Toronto E-mail: faith@cs.toronto.edu

Rati Gelashvili
University of Toronto E-mail: gelash@cs.toronto.edu

Nir Shavit
MIT E-mail: shanir@csail.mit.edu

Leqi Zhu
University of Toronto E-mail: lezhu@cs.toronto.edu

Objects that support only one of these instructions have consensus number 2 [Her91]. Moreover, by robustness, using both these deterministic read-modify-write objects, it is still not possible to solve wait-free consensus for 3 or more processes. However, with one memory location that supports both instructions, it is possible to solve wait-free binary consensus for any number of processes: The memory location is initialized to 0. Processes with input 0 perform *fetch-and-add*(2), while processes with input 1 perform *test-and-set*(). If the value returned is odd, the process decides 1. If the value 0 was returned from *test-and-set*(), the process also decides 1. Otherwise, the process decides 0.

Another example considers three instructions defined on the set of integers:

– *read*(), which returns the number stored in a memory location,
– *decrement*(), which decrements the number stored in a memory location and returns nothing, and
– *multiply*(x), which multiplies the number stored in a memory location by x and returns nothing.

A similar situation arises: Objects that support only *decrement*() and *read*() or only *multiply*(x) and *read*() have consensus number 1 and, together, they still cannot be used to solve wait-free consensus for 2 or more processes. Again, using a memory location that supports all three instructions, it is possible to solve wait-free binary consensus for any number of processes: The memory location is initialized to 1. Processes with input 0 perform *decrement*(), while processes with input 1 perform *multiply*(n). The second operation by each process is *read*(). If the value returned is positive, then the process decides 1. Otherwise, the process decides 0.

Randomized wait-free binary consensus among any number of processes can be solved using only read-write registers, which support only *read*() and *write*(x) and have consensus number 1. Thus Herlihy's Consensus Hierarchy collapses for randomized computation.

Historyless objects support only *trivial operations*, such as *read*(), which never change the value of an object, and *historyless operations*, such as *write*(x), *swap*(x), and *test-and-set*(), which always change an object to a predetermined value. They have consensus number at most 2. Ellen, Herlihy, and Shavit [FHS98] proved that $\Omega(\sqrt{n})$ instances of historyless objects are necessary to solve randomized wait-free consensus among n processes. They noted that, in contrast, only one instance of an object supporting *read*(), *increment*(), and *decrement*() (which has consensus number 1), one instance of an object supporting *fetch-and-add*(x) (which has consensus number 2), or one instance of an object supporting *compare-and-swap*(x, y) (which has infinite consensus number) suffices for solving this problem. Their lower bound implies that, in a system of n processes, $\Omega(\sqrt{n})$ instances of historyless objects are needed for a randomized wait-free implementation of an object that supports *compare-and-swap*(x, y) or an object that supports *fetch-and-add*(x). Jayanti, Tan, and Toueg [JTT96] improved this result by showing that at least $n-1$ instances of historyless objects or resettable consensus objects are needed.

Ellen, Herlihy, and Shavit [FHS98] suggested that another way to classify the power of an object is by the number of instances of the object needed to solve randomized wait-free consensus among n processes. Instead, in this paper, we decided to consider a classification of objects based on the number of instances of that object needed to solve *obstruction-free consensus* among n processes, i.e. there exists a deterministic algorithm such that, from every reachable configuration C, each process will decide if it is given sufficiently many consecutive steps. Obstruction-freedom is a natural progress condition and is the guarantee provided by some hardware transactions [Int12].

Although obstruction-freedom is a simpler property than randomized wait-freedom, they are closely related. In fact, any (deterministic) obstruction-free algorithm can be transformed into a randomized wait-free algorithm that uses the same number of memory locations (against an oblivious adversary) [GHHW13]. Obstruction-free algorithms can also be transformed into wait-free algorithms in the unknown-bound semi-synchronous model [FLMS05].

The lower bound by Ellen, Herlihy, and Shavit [FHS98] was actually proved for consensus algorithms that satisfy *nondeterministic solo-termination*, which is a less restrictive property than either randomized wait-freedom or obstruction freedom. Hence, it implies the same lower bound for obstruction-free and randomized wait-free algorithms. Nondeterministic solo-termination requires that, for every process p and every reachable configuration C, there exists a solo execution by p from C in which p decides.

In a recent paper [EGZ18], Ellen, Gelashvili and Zhu proved that if there is a nondeterministic solo-terminating algorithm for a task using m read-write registers, then there is an obstruction-free algorithm for that task using the same number of read-write registers. Thus, any lower bound on the number of read-write registers used by obstruction-free consensus algorithms is a lower bound on the number of read-write registers used by nondeterministic solo-terminating consensus algorithms and, hence, randomized wait-free consensus algorithms. In the full version of that paper, the proof is extended to algorithms using any deterministic readable objects.

Since multiprocessors always provide *read* and *write*, objects supporting at least these operations are most interesting to study. Note that *fetch-and-add*(0), *fetch-and-multipy*(1), and *compare-and-swap*(x, x), for any value of x, all provide the same functionality as *read*. However, to understand the classification better, we also classify some objects that do not support *write*.

Many well-known operations, for example *read*, *write*, *fetch-and-add*, and *compare-and-swap*, have infinite domains. Although it is reasonable to consider objects whose domain sizes are constant or a function of the number of processes, we have chosen to restrict attention to objects with a countably infinite domain. By encoding the values in the domain as nonnegative integers, one could assume that the domain is $\mathbb{N}$. The lower bounds proved for these objects also apply to objects supporting the same set of instructions, but with smaller domains. In some cases, the upper bounds hold for smaller domains. When these upper bounds match the lower bounds, this shows that, for these sets of instructions, domain size is not important. In any case, we believe it is important to understand the relative power of sets of instructions separately from their power when the information they can convey is limited by the domain size.

1.1 Our Results

We use *n-consensus* to denote the problem of solving obstruction-free n-valued consensus among $n \geq 2$ processes. Let $\mathcal{SP}(\mathcal{I}, n)$ denote the minimum number of instances of an object with a countably infinite domain and that supports the instruction set $\mathcal{I}$ for which it is possible to solve n-consensus. This is a function from the set of integers 2 and larger to the set consisting of the positive integers and ∞. Equivalently, $\mathcal{SP}(\mathcal{I}, n)$ is the minimum number of memory locations of unbounded size that are needed to solve *n-consensus* using only the instructions in $\mathcal{I}$. For various sets of instructions, $\mathcal{I}$, we provide lower and upper bounds on $\mathcal{SP}(\mathcal{I}, n)$. The results are summarized in Table 1.

We begin, in Section 2, by defining our model. In Section 3, we consider the instructions

– *multiply*(x), which multiplies the natural number stored in a memory location by the natural number x and returns nothing,
– *add*(x), which adds the natural number x to the natural number stored in a memory location and returns nothing, and
– *set-bit*(x), which sets bit x of a memory location to 1 and returns nothing.

We show that using *read*$()$ and one of these instructions, it is possible to solve n-consensus with a single memory location. The idea is to show that these instruction sets can implement n counters in the memory location. We can then use a racing counters algorithm that is similar to a consensus algorithm by Aspnes and Herlihy [AH90].

Next, we consider a *max-register* [AAC09]. This is a memory location that is initially 0 and supports the instructions

– *read-max*$()$, which reads the natural number stored in the memory location, and
– *write-max*(x), which stores the positive integer x in the memory location, provided it contains a value less than x, and returns nothing.

In other words, *read-max*$()$ returns the largest $x \in \mathbb{Z}^+$ such that *write-max*(x) has been performed on the memory location. It returns 0 if no *write-max* instruction has been performed on the memory location. We prove that two max-registers are necessary and sufficient for solving n-consensus in Section 4.

In Section 5, we prove that using *read*$()$, *write*(x), and *fetch-and-increment*$()$ and only one memory location, it is not possible to solve n-consensus, for $n \geq 3$. We also present an algorithm for solving n-consensus using these instructions and $O(\log n)$ memory locations.

An ℓ*−buffer* is a memory location that supports buffered read and buffered write instructions, with a buffer of length $\ell \geq 1$. Specifically, an ℓ*-buffer-read* instruction returns the sequence of inputs, x, to the ℓ most recent ℓ*-buffer-write*(x) instructions applied to the memory location, in order from least recent to most recent. If the number of ℓ*-buffer-write* instructions previously applied to the memory location is $\ell' < \ell$, then the first $\ell - \ell'$ elements of this sequence are $\perp$. A 1-buffer is simply a read-write register. Independently and concurrently with our work, Perrin, Mostefaoui, and Jard [Per16, PMJ16] defined the same object, but called it a window stream of size ℓ. Perrin [Per16] proved that it has consensus number ℓ.

A *history* object supports two operations, *get-history*$()$ and *append*(x), where *get-history*$()$ returns the sequence of all values appended to it by prior *append* operations, in order. It has infinite consensus number [Dav04, DBF05]. A history object is the same as an $\ell-$buffer with $\ell = \infty$.

We consider ℓ-buffers in Section 6. We show how to solve n-consensus using $\lceil \frac{n}{\ell} \rceil$ many ℓ-buffers. We also extend Zhu's $n - 1$ lower bound [Zhu16] to prove that

Set of Instructions $\mathcal{I}$	lower bound on $\mathcal{SP}(\mathcal{I}, n)$	upper bound on $\mathcal{SP}(\mathcal{I}, n)$
$\{read(), test\text{-}and\text{-}set()\}$, $\{read(), write(1)\}$	∞	∞
$\{read(), write(1), write(0)\}$	n	$O(n \log n)$
$\{read(), write(x)\}$	n	n
$\{read(), test\text{-}and\text{-}set(), reset()\}$	$\Omega(\sqrt{n})$	$O(n \log n)$
$\{read(), swap(x)\}$	$\Omega(\sqrt{n})$	$n-1$
$\{\ell\text{-}buffer\text{-}read(), \ell\text{-}buffer\text{-}write(x)\}$	$\lceil (n-1)/\ell \rceil$	$\lceil n/\ell \rceil$
$\{read(), write(x), increment()\}$ $\{read(), write(x), fetch\text{-}and\text{-}increment()\}$	2	$O(\log n)$
$\{read\text{-}max(), write\text{-}max(x)\}$	2	2
$\{compare\text{-}and\text{-}swap(x, y)\}$ $\{read(), set\text{-}bit(x)\}$ $\{read(), add(x)\}$, $\{read(), multiply(x)\}$ $\{fetch\text{-}and\text{-}add(x)\}$, $\{fetch\text{-}and\text{-}multiply(x)\}$	1	1

Table 1 Classification of Objects with a Countably Infinite Domain Supporting Different Sets of Instructions.

$\lceil \frac{n-1}{\ell} \rceil$ many ℓ-buffers are necessary to solve n-consensus. This is tight, except when $n-1$ is divisible by ℓ.

In Section 7, we consider ℓ-buffers extended to support atomic multiple assignment. This means that a process can atomically perform one *ℓ-buffer-write* to any number of different ℓ-buffers, instead of just one. Multiple assignment to read-write registers plays an important role in the Consensus Hierarchy [Her91]: using multiple assignment to $m \geq 2$ read-write registers, it is possible to solve wait-free consensus for $2m-2$ processes, but not for $2m-1$ processes. We show that at least $\lceil \frac{n-1}{2\ell} \rceil$ different ℓ-buffers are needed to solve n-consensus, even in the presence of atomic multiple assignment. Multiple assignment can be implemented by simple transactions, so our result implies that such transactions cannot significantly reduce the number of read-write registers or ℓ-buffers needed. This result is the most technical contribution of the paper. The proof further extends the techniques of [Zhu16] via a combinatorial argument, which is of independent interest.

There are algorithms that solve n-consensus using n read-write registers [AH90, BRS15, Zhu15]. This is tight by the recent result of [EGZ18], which shows a lower bound of n read-write registers for obstruction-free binary consensus among n processes and, hence, for n-consensus. In Section 8, we present an algorithm for n-consensus using $n-1$ memory locations supporting $\{read(), swap(x)\}$. This is a modification of one of the known anonymous algorithms for n-consensus [Zhu15]. A lower bound of $\Omega(\sqrt{n})$ locations appears in [FHS98]. That lower bound also applies to memory locations that only support *read*(), *test-and-set*() and *reset*() instructions.

Finally, in Section 9, we show that an unbounded number of memory locations supporting *read*() and either *write*(*1*) or *test-and-set*() are necessary and sufficient to solve n-consensus, for $n \geq 3$. Furthermore, we show how to reduce the number of memory locations to $O(n \log n)$ when, in addition to *read*(), either *write*(0) and *write*(1) are both available, or *test-and-set*() and *reset*() are both available.

2 Preliminaries

We consider an asynchronous system of $n \geq 2$ processes, with distinct identifiers $0, 1 \ldots, n-1$, that communicate through a collection of identical memory locations, each supporting the same set, $\mathcal{I}$, of deterministic instructions.

The processes take steps at arbitrary, possibly changing, speeds and may crash at any time. Scheduling is controlled by an adversary. When allocated a step by the scheduler, a process atomically performs one instruction on one shared memory location and, based on the result, may then perform an arbitrary amount of local computation. This is a standard asynchronous shared memory model [AW04], with the restriction that all memory locations are instances of the same object.

A *configuration* consists of the state of every process and the contents of every memory location. An *execution* from a configuration C consists of an alternating sequence of steps and configurations beginning with C. A *P-only execution* is an execution in which only processes in P take steps. A *solo execution* is an execution in which only one process takes steps.

We consider the problem of solving *obstruction-free m-valued consensus* in such a system. Initially, each of the n processes has an input from $\{0, 1, \ldots, m-1\}$ and is supposed to output a value (called a *decision*), such that all decisions are the same (*agreement*) and equal to the input of one of the processes (*validity*). Once a process has decided (i.e. output its decision), the scheduler does not allocate it any further steps. A *solo-terminating execution* is a finite solo execution that ends when the process has decided. *Obstruction-freedom* means that, from each reachable configuration and for each process, there is a solo-terminating execution from that configuration by that process, i.e. if the adversarial scheduler gives the process sufficiently many consecutive steps, then the process will eventually decide a value. When $m = n$, we call this problem *n-consensus* and, when $m = 2$, we call this problem *binary consensus*. Note that lower bounds for binary consensus also apply to n-consensus.

In every reachable configuration of a consensus algorithm, each process has either decided or has one specific instruction it will perform on a particular memory location when next allocated a step by the scheduler. In this latter case, we say that the process is *poised* to perform that instruction on that memory location in the configuration.

Consider any binary consensus algorithm. We say that a set of processes $\mathcal{P}$ *can decide* $v \in \{0, 1\}$ *from* configuration C if there exists a $\mathcal{P}$-only execution from C in which v is decided. If $\mathcal{P}$ can decide both 0 and 1 from C, then $\mathcal{P}$ is *bivalent* from C. When the objects and algorithms are deterministic, from each configuration, a process only has one solo-terminating execution and, hence, can only decide one value.

The following two results are important components of a number of our lower bounds. They do not depend on what instructions are supported by the memory.

Lemma 1. *There is an initial configuration from which the set of all processes in the system is bivalent.*

Proof. Consider an initial configuration, I, with two processes p_0 and p_1, such that p_v starts with input v, for $v \in \{0, 1\}$. Observe that $\{p_v\}$ can decide v from I since, initially, I is indistinguishable to p_v from the configuration where every process starts with input v. Thus, $\{p_0, p_1\}$ is bivalent from I and, therefore, so is the set of all processes. □

Next, we show that, if a set of processes is bivalent in some configuration, then it is possible to reach a configuration from which 0 and 1 can be decided in solo executions.

Lemma 2. *Suppose $\mathcal{U}$ is a set of at least two processes that is bivalent from configuration C. Then it is possible to reach, via a $\mathcal{U}$-only execution from C, a configuration, C', such that, for all $v \in \{0, 1\}$, there is a process $q_v \in \mathcal{U}$ that decides v in its solo-terminating execution from C'.*

Proof. Suppose, for a contradiction, that for every configuration C' reachable via a $\mathcal{U}$-only execution from C, every process in $\mathcal{U}$ decides the same value in its solo-terminating execution from C'. In particular, every process in $\mathcal{U}$ decides the same value, v, in its solo-terminating execution from C. Since $\mathcal{U}$ is bivalent from C, there is a $\mathcal{U}$-only execution α from C in which $\bar{v}$ is decided. Consider the longest prefix α' of α such that some (and, hence, by assumption, every) process in $\mathcal{U}$ decides v in its solo-terminating execution from $C\alpha'$. Notice that $\alpha' \neq \alpha$ since $\bar{v}$ is decided in α. Let δ be the next step after α' in α. Since α is a $\mathcal{U}$-only execution, δ is by some process $p \in \mathcal{U}$. By definition of α', every process in $\mathcal{U}$ decides $\bar{v}$ in its solo-terminating execution from $C\alpha'\delta$. However, since p decides v in its solo-terminating execution from $C\alpha'$, it decides v in its solo-terminating execution from $C\alpha'\delta$. This is a contradiction. □

3 Arithmetic Instructions

Consider a system that supports only *read*() and either *add*(x), *multiply*(x), or *set-bit*(x). We show how to solve n-consensus using a single memory location in such a system. The idea is to show that we can simulate certain collections of objects that can solve n-consensus.

An *m-component unbounded counter* is a memory location that has m components, each containing a nonnegative integer value. It supports *increment*(v), which increments the count stored in component v by 1, and *scan*(), which returns the counts of all m components. In the next lemma, we present a racing counters algorithm.

Lemma 3. *It is possible to solve obstruction-free m-valued consensus among n processes using an m-component unbounded counter.*

Proof. The components are indexed by the m possible input values. All components initially have value 0. Each process alternates between incrementing one component and performing a *scan* of all m components. A process first increments the component indexed by its input value. After performing a *scan*, if it observes that the count stored in component v is at least n larger than the counts stored in all other components, it returns the value v. Otherwise, it increments one of the components containing the largest count (breaking ties arbitrarily).

If some process returns the value v, then each other process will increment some component at most once before next performing a *scan*. In each of those *scan*s, the count stored in component v will still be larger than the counts stored in all other components. From then on, these processes will keep incrementing component v. Eventually, the count in component v will be at least n larger than the counts in all other components, and these processes will return v, ensuring agreement.

Obstruction-freedom follows because a process running on its own will continue to increment the same component, which will eventually be n larger than the counts in all other components. □

In this algorithm, the counts stored in the components may grow arbitrarily large. The next lemma shows that it is possible to avoid this problem, provided the memory location also supports *decrement*(v), which decrements the count stored in component v by 1. More formally, an *m-component b-bounded counter* object has m components, where each component stores a count in $\{0, 1, \ldots, b\}$. It supports *increment*(v) and *decrement*(v), for each component v, along with a *scan*() operation, which returns the count stored in every component. If a process ever attempts to increment a component that has count b or decrement a component that has count 0, the object breaks (and every subsequent instruction a returns $\perp$).

Lemma 4. *It is possible to solve obstruction-free m-valued consensus among n processes using an m-component $(3n-1)$-bounded counter.*

Proof. The construction in the proof of Lemma 3 is modified slightly by sometimes changing what a process does when it wants to increment component v: If some other component has count at least n, it decrements one component (excluding v) that stores the largest count, instead of incrementing component v. If all other components have count less than n, it still increments component v.

A component with count 0 is never decremented. This is because, after the last time some process observed that it had count at least n, each process will decrement the component at most once before performing a *scan*(). Similarly, no component ever becomes larger than $3n-1$: After the last time some process observed that some component v had count less than $2n$, each process will increment component v at most once before performing a *scan*(). If the count in component v is at least $2n$, then either the counts in all other components are less than n, in which case the process decides without incrementing component v or the process decrements some other component, instead of incrementing component v. □

In the following theorem, we show how to simulate unbounded and bounded counters.

Theorem 5. *It is possible to solve n-consensus using a single memory location that supports only read() and either multiply(x), add(x), or set-bit(x).*

Proof. We first give an obstruction-free implementation of an n-component unbounded counter using a single memory location of unbounded size that supports *read*() and *multiply*(x). By Lemma 3, this is sufficient for solving n-consensus. The location is initialized with value 1. For each $v \in \{0, \ldots, n-1\}$, let p_v be the $(v+1)$'st prime number. A process increments component v by performing *multiply*(p_v). A *read*() instruction returns the value y currently stored in the memory location. This provides a *scan* of all components: the value of component v is the exponent of p_v in the prime decomposition of y.

A similar construction does not work using only *read*() and *add*(x) instructions. For example, suppose one component is incremented by calling *add*(a) and another component is incremented by calling *add*(a'). Then incrementing the first component a' times or incrementing the second component a times results in the memory location having the same value, $a \cdot a'$.

However, we can use a single memory location that stores an $O(n \log n)$ bit natural number and supports $\{$*read*(), *add*$(x)\}$ to implement an n-component $(3n-1)$-bounded counter. By Lemma 4, this is sufficient for solving consensus. We view the value stored in the location as a number written in base $3n$ and interpret the i'th least significant digit of this number as the count of component $i-1$. The location is initialized with the value 0. To increment and decrement component i, a process performs *add*$((3n)^i)$ and *add*$(-(3n)^i)$, respectively. From the result of a *read*(), a process can obtain a *scan* of all n components.

Finally, in systems supporting *read*() and *set-bit*(x), we can implement an n-component unbounded counter by viewing the memory location as being partitioned into a countably infinite number of blocks, each consisting of n^2 bits. The $(in+v)$'th bit of its b'th block is 1 if and only if process i has incremented component v at least b times. Initially all bits are 0. Each process locally stores the number of times it has incremented each component. To increment component v, process i sets the $(in+v)$'th bit in block $b+1$ to 1, where b is the number of times it has previously incremented component v. It is possible to determine the count stored in each component via a single *read*(): The count stored in component v is simply the sum of the number of times each process has incremented component v, which is the number of 1's in bit positions congruent to v modulo n. □

4 Max-Registers

We show that two max-registers are necessary and sufficient for solving n-consensus.

Theorem 6. *It is not possible to solve obstruction-free binary consensus for $n \geq 2$ processes using a single max-register.*

Proof. Suppose there is an obstruction-free algorithm solving binary consensus using one max-register. Let C be an initial configuration where a process p has input 0 and another process q has input 1. Consider a solo-terminating execution α of p from C and a solo-terminating execution β of q from C. We show how to interleave these two executions so that the resulting execution is indistinguishable to both processes from their respective solo executions. Hence, both values will be returned, contradicting agreement.

To build the interleaved execution, run both processes until they are first poised to perform *write-max*. Suppose p is poised to perform *write-max*(a) and q is poised to perform *write-max*(b). If $a \leq b$, let p take steps until it is next poised to perform *write-max*(a'), with $a' > b$, or until the end of α, if it performs no such *write-max* operations. Otherwise, let q take steps until it is next poised to perform *write-max*(b'), with $b' > a$, or until the end of β. Repeat this until one of the processes reaches the end of its execution and then let the other process finish. □

Theorem 7. *It is possible to solve n-consensus for any number of processes using only two max-registers.*

Proof. We describe an algorithm for n-consensus using two max-registers, m_1 and m_2. Consider the lexicographic ordering $\prec$ on the set $S = \mathbb{N} \times \{0, \ldots, n-1\}$. Let y be a fixed prime that is larger than n. Note that, for $(r, x), (r', x') \in S$, $(r, x) \prec (r', x')$ if and only if $(x+1)y^r < (x'+1)y^{r'}$. Thus, by identifying $(r, x) \in S$ with $(x+1)y^r$, we may assume that m_1 and m_2 are max-registers defined on S with respect to the lexicographic ordering $\prec$.

Since no operations decrease the value in a max-register, it is possible to implement an obstruction-free *scan* operation on m_1 and m_2 using the double collect algorithm [AAD+93]: A process repeatedly collects the values in both locations (performing *read-max*() on each location to obtain its value) until it observes two consecutive collects with the same values.

Initially, both m_1 and m_2 have value $(0, 0)$. Each process alternately performs *write-max* on one component and takes a *scan* of both components. It begins by performing *write-max*$(0, x')$ to m_1, where $x' \in \{0, \ldots, n-1\}$ is its input value. If m_1 has value $(r+1, x)$ and m_2 has value (r, x) in the *scan*, then it decides x and terminates. If both m_1 and m_2 have value (r, x) in the *scan*, then it performs *write-max*$(r+1, x)$ to m_1. Otherwise, it performs *write-max* to m_2 with the value of m_1 in the *scan*.

To obtain a contradiction, suppose that there is an execution in which some process p decides value x and another process q decides value $x' \neq x$. Immediately before its decision, p performed a *scan* where m_1 had value $(r+1, x)$ and m_2 had value (r, x), for some $r \geq 0$. Similarly, immediately before its decision, q performed a *scan* where m_1 had value $(r'+1, x')$ and m_2 had value (r', x'), for some $r' \geq 0$. Without loss of generality, we may assume that q's *scan* occurs after p's *scan*. In particular, m_2 had value (r, x) before it had value (r', x'). So, from the specification of a max-register, $(r, x) \preceq (r', x')$. Since $x' \neq x$, it follows that $(r, x) \prec (r', x')$.

We show inductively, for $j = r', \ldots, 0$, that some process performed a *scan* in which both m_1 and m_2 had value (j, x'). By assumption, q performed a *scan* where m_1 had value $(r'+1, x')$. So, some process performed *write-max*$(r'+1, x')$ on m_1. From the algorithm, this process performed a *scan* where m_1 and m_2 both had value (r', x'). Now suppose that $0 < j \leq r'$ and some process performed a *scan* in which both m_1 and m_2 had value (j, x'). So, some process performed *write-max*(j, x') on m_1. From the algorithm, this process performed a *scan* where m_1 and m_2 both had value $(j-1, x')$.

Consider the smallest value of j such that $(r, x) \prec (j, x')$. Note that $(r, x) \prec (r', x)$, so $j \leq r'$. Hence, some process performed a *scan* in which both m_1 and m_2 had value (j, x'). Since $(r, x) \prec (j, x')$, this *scan* occurred after the *scan* by p, in which m_2 had value (r, x). But m_1 had value (j, x') in this *scan* and m_1 had value $(r+1, x)$ in p's *scan*, so $(r+1, x) \preceq (j, x')$. Since $x \neq x'$, it follows that $(r+1, x) \prec (j, x')$. Hence $j \geq 1$ and $(r, x) \prec (j-1, x')$. This contradicts the choice of j. Thus, the algorithm satisfies agreement.

Suppose some process decides value x. Then it performed a *scan* where m_1 had value $(r+1, x)$ and m_2 had value (r, x) for some $r \geq 0$. Consider the smallest r' such that some process performed *write-max*(r', x) to m_1. If $r' > 0$, then both m_1 and m_2 had value $(r'-1, x)$ in its preceding *scan*. But this means that some process performed *write-max*$(r'-1, x)$ to m_1, contradicting the definition of r'. Thus, $r' = 0$. Since a process only performs *write-max*$(0, x)$ to m_1 if it has input x, the algorithm satisfies validity.

Suppose m_1 had value (r, x) and m_2 had value in the first *scan* of a solo execution by some process. If $r = r' + 1$ and $x = x'$, then the process terminates immediately. If $r = r'$ and $x = x'$, then it performs *write-max*$(r+1, x)$ to m_1 and terminates immediately after its next *scan*. Otherwise, the process performs *write-max*(r, x) to m_2 and terminates after two more *scans*. Thus, each process terminates after performing *scan* at most 3 times in a solo execution. Hence, the algorithm is obstruction-free. □

5 Increment

Consider a system that supports only *read*(), *write*(x), and *fetch-and-increment*(). We prove that it is not possible to solve binary consensus and, hence, n-consensus using a single memory location. We also consider a weaker system that supports only *read*(), *write*(x), and *increment*() and provide an algorithm for n-consensus using $O(\log n)$ memory locations.

Theorem 8. *It is not possible to solve obstruction-free binary consensus for $n \geq 2$ processes using a single memory location that supports only read(), write(x), and fetch-and-increment().*

Proof. Suppose there is a binary consensus algorithm for two processes, p and q, using only one memory location. Let α be a solo-terminating execution by p starting from any initial configuration in which p has input 0 and let α' be the longest prefix of α that does not contain a *write*. Similarly, let β be a solo-terminating execution by p starting from any initial configuration in which p has input 1 and let β' be the longest prefix of β that does not contain a *write*. Without loss of generality, suppose that at least as many *fetch-and-increment*() instructions are performed in β' as in α'. Let C be the configuration that results from executing α' starting from the initial configuration in which p has input 0 and the other process, q has input 1.

Consider the shortest prefix β'' of β' in which p performs the same number of *fetch-and-increment*() instructions as it performs in α'. Let C' be the configuration that results from executing β'' starting from the initial configuration in which both p and q have input 1. Then q must decide 1 in its solo-terminating execution γ starting from configuration C'. However, C and C' are indistinguishable to process q, so it must decide 1 in γ starting from configuration C. If p has decided in configuration C, then it has decided 0, since q takes no steps in α'. Then both 0 and 1 are decided in execution $\alpha'\gamma$ starting from the initial configuration in which p has input 0 and q has input 1. This violates agreement. Thus, p cannot have decided in configuration C.

Therefore, p is poised to perform a *write* in configuration C. Let α'' be the remainder of α, so $\alpha = \alpha'\alpha''$. Since there is only one memory location, the configurations resulting from performing this *write* starting from C and $C\gamma$ are indistinguishable to p. Thus, p also decides 0 starting from $C\gamma$. But in this execution, both 0 and 1 are decided, violating agreement. □

The following well-known construction converts any algorithm for solving binary consensus to an algorithm for solving n-consensus [HS12].

Lemma 9. *Consider a system that supports a set of instructions that includes read() and write(x). If it is possible to solve obstruction-free binary consensus among n processes using only c memory locations, then it is possible to solve n-consensus using only $(c+2) \cdot \lceil \log_2 n \rceil - 2$ locations.*

Proof. The processes agree bit-by-bit in $\lceil \log_2 n \rceil$ asynchronous rounds, each using $c + 2$ locations. A process starts in the first round with its input value as its value for round 1. In round i, if the i'th bit of its value is 0, a process p writes its value in a designated location associated with bit 0 for the round. Otherwise, it writes its value in a designated location associated with bit 1 for round i. Next, it performs the obstruction-free binary consensus algorithm using c locations to agree on the i'th bit, v_i, of the output. If the i'th bit of p's value differs from the decided bit, v_i, then some other process proposed v_i. Before doing so, that other process wrote its value to the designated location associated with bit v_i for round i. Then process p can read a value from this designated location and adopt it for the next round. This ensures that the values used for round $i+1$ are all input values and they all agree in their first i bits. By the end, all processes have agreed on $\lceil \log_2 n \rceil$ bits, i.e. on one of the at most n different input values.

We can save two locations because the last round does not require designated locations associated with 0 and 1. □

We can implement a 2-component unbounded counter, defined in Section 3, using two locations that support *read*() and *increment*(). The values in the two locations never decrease. Therefore, as in the proof of Theorem 7, an obstruction-free *scan*() operation that returns the values of both counters can be performed using the double collect algorithm [AAD+93]. By Lemma 3, n processes can solve obstruction-free binary consensus using a 2-component unbounded counter. The next result then follows from Lemma 9.

Theorem 10. *It is possible to solve n-consensus using $O(\log n)$ memory locations that support only read(), write(x), and increment().*

6 Buffers

First, we show that a single ℓ-buffer can be used to simulate a history object that can be updated by at most ℓ processes. This will allow us to simulate an obstruction-free variant of Aspnes and Herlihy's algorithm for n-consensus [AH90] and, hence, solve n-consensus, using only $\lceil n/\ell \rceil$ ℓ-buffers. Then we prove that $\lceil (n-1)/\ell \rceil$

ℓ-buffers are necessary. This matches the upper bound whenever $n-1$ is not a multiple of ℓ.

6.1 Simulations Using Buffers

We begin by showing how to simulate a history object that supports arbitrarily many readers and at most ℓ different appenders, using a single ℓ-buffer. When $\ell = 1$, this is straightforward, since a history object that supports only 1 appender can be simulated using a single-writer register to which the appender writes the sequence of all values it has previously appended together with the new value it wants to append.

Lemma 11. *For $\ell \geq 2$, a single ℓ-buffer can simulate a history object on which at most ℓ different processes can perform append(x) and any number of processes can perform get-history$()$.*

Proof. Without loss of generality, assume that no value is appended to the history object H more than once. This can be achieved by having a process include its process identifier and a sequence number along with the value that it wants to append.

In our implementation, the ℓ-buffer B is initially $\bot$. Each value written to B is of the form $(\mathbf{h}, x)$, where $\mathbf{h}$ is an arbitrarily long finite history of appended values and x is a single appended value.

To implement *append*(x) on H, a process obtains a history, $\mathbf{h}$, by performing *get-history*$()$ on H and then performs ℓ*-buffer-write*$(\mathbf{h}, x)$ on B. The operation is linearized at this ℓ*-buffer-write* step.

To implement *get-history*$()$ on H, a process simply performs an ℓ*-buffer-read* of B to obtain a vector $(a_1, \ldots, a_\ell)$, where a_ℓ is the most recently written value. The operation is linearized at this ℓ*-buffer-read*. We describe how the return value of the *get-history*$()$ operation is computed.

We prove that each *get-history*$()$ operation, G, on H returns the sequence of inputs to all *append* operations on H that were linearized before it, in order from least recent to most recent. Let R be the ℓ*-buffer-read* step performed by G and let $(a_1, \ldots, a_\ell)$ be the vector returned by R.

Note that $(a_1, \ldots, a_\ell) = (\bot, \ldots, \bot)$ if and only if no ℓ*-buffer-write* steps were performed before R i.e. if and only if no *append* operations are linearized before G. In this case, the empty sequence is returned by the *get-history*$()$ operation, as required.

Now suppose that $k \geq 1$ ℓ*-buffer-write* steps were performed on B before R, i.e. k *append* operations were linearized before G. Inductively assume that each *get-history*$()$ operation which has fewer than k *append* operations linearized before it returns the sequence of inputs to those *append* operations.

If $a_i \neq \bot$, then $a_i = (\mathbf{h}_i, x_i)$ was the input to an ℓ*-buffer-write* step W_i on B performed before R. This step was performed during an *append*(x_i) operation, A_i, whose *get-history*$()$ operation, G_i, returned the history $\mathbf{h}_i$ of appended values. Let R_i be the ℓ*-buffer-read* step performed by G_i. Since R_i occurred before W_i, which occurred before R, fewer than k ℓ*-buffer-write* steps occurred before R_i. Hence, fewer than k *append* operations are linearized before G_i. By the induction hypothesis, $\mathbf{h}_i$ is the sequence of inputs to the *append* operations linearized before G_i.

If $k < \ell$, then $a_1 = \cdots = a_{\ell-k} = \bot$. In this case, G returns the sequence $x_{\ell-k+1}, \ldots, x_\ell$. Since each *append* operation is linearized at its ℓ*-buffer-write* step and $x_{\ell-k+1}, \ldots, x_\ell$ are the inputs to these k *append* operations, in order from least recent to most recent, G returns the sequence of inputs to the *append* operations linearized before it.

So, suppose that $k \geq \ell$. Let $\mathbf{h}_m$ be the longest history amongst $\mathbf{h}_1, \ldots, \mathbf{h}_\ell$. If $\mathbf{h}_m$ contains x_1, then G returns $\mathbf{h}', x_1, \ldots, x_\ell$, where $\mathbf{h}'$ is the prefix of $\mathbf{h}_m$ up to, but not including, x_1. By definition, $a_1, \ldots, a_\ell$ are the inputs to the last ℓ*-buffer-write* operations prior to R, so $x_1, \ldots, x_\ell$ are the last ℓ values appended to H prior to G. Since $\mathbf{h}_m$ contains x_1, it also contains all values appended to H prior to x_1. It follows that $\mathbf{h}' \cdot (x_1, \ldots, x_\ell)$ is the sequence of inputs to the *append* operations linearized before G.

Now suppose that $\mathbf{h}_m$ does not contain x_1. Then none of $\mathbf{h}_1, \ldots, \mathbf{h}_\ell$ contain x_1. Hence $G_1, \ldots, G_\ell$ were linearized before A_1 and $R_1, \ldots, R_\ell$ were performed prior to W_1. Since step W_1 occurred before $W_2, \ldots, W_\ell$, the operations $A_1, \ldots, A_\ell$ are all concurrent with one another. This is illustated in Figure 1. Therefore $A_1, \ldots, A_\ell$ are performed by different processes. Only ℓ different processes can perform *append* operations on H, so no other *append* operations on H are linearized between R_m and W_1. Therefore, $\mathbf{h}_m$ contains all values appended to H prior to x_1. It follows that $\mathbf{h}_m \cdot (x_1, \ldots, x_\ell)$ is the sequence of inputs to the *append* operations linearized before G. □

This lemma allows us to simulate *any* object that supports at most ℓ updating processes using only a single ℓ-buffer. This is because the state of an object is determined by the history of the non-trivial operations performed on it. In particular, we can simulate an array of ℓ single-writer registers using a single ℓ-buffer.

Lemma 12. *A single ℓ-buffer can simulate ℓ single-writer registers.*

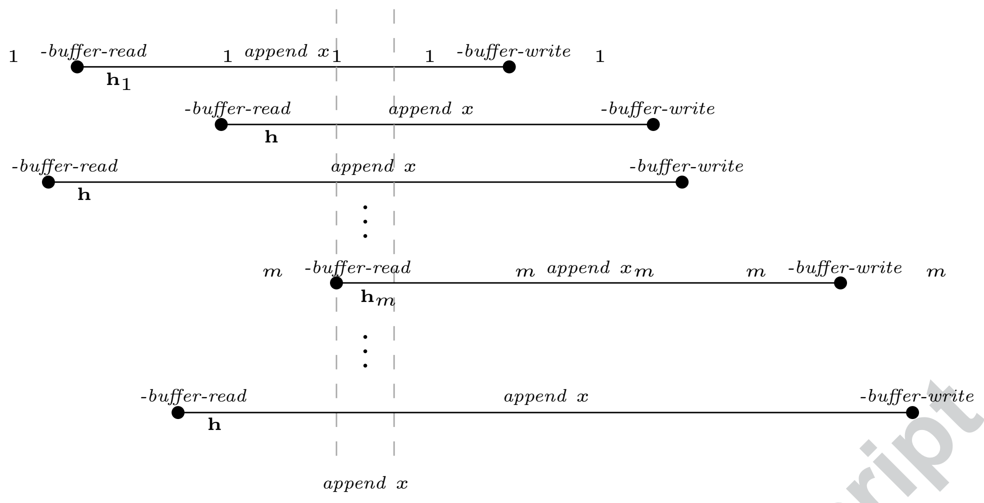

Fig. 1 When $\mathbf{h}_m$ does not contain x_1, there are ℓ concurrent *append* operations.

Proof. Suppose that register R_i is owned by process p_i, for $1 \leq i \leq \ell$. By Lemma 11, it is possible to simulate a history object H that can be updated by ℓ processes and read by any number of processes. To write value x to R_i, process p_i appends (i, x) to H. To read R_i, a process reads H and finds the value of the most recent write to R_i. This is the second component of the last pair in the history whose first component is i. $\square$

Thus, we can use $\lceil \frac{n}{\ell} \rceil$ ℓ-buffers to simulate n single-writer registers. An n-component unbounded counter shared by n processes can be implemented in an obstruction-free way from n single-writer registers. Each process records the number of times it has incremented each component in its single-writer register. As in the proof of Theorem 7, an obstruction-free *scan*() can be performed using the double collect algorithm [AAD+93]. Hence, by Lemma 3 we get the following result.

Theorem 13. *It is possible to solve n-consensus using only* $\lceil n/\ell \rceil$ *ℓ-buffers.*

6.2 A Lower Bound

In this section, we prove a lower bound on the number of ℓ-buffers necessary for solving obstruction-free binary consensus among $n \geq 2$ processes.

In any configuration, memory location r is *covered* by process p if p is poised to perform *ℓ-buffer-write* on r. A memory location is *k-covered* by a set of processes $\mathcal{P}$ in a configuration if there are exactly k processes in $\mathcal{P}$ that cover it. A configuration is *at most k-covered* by $\mathcal{P}$, if every process in $\mathcal{P}$ covers some memory location and no memory location is k'-covered by $\mathcal{P}$, for any $k' > k$.

Let C be a configuration and let $\mathcal{Q}$ be a set of processes, each of which is poised to perform *ℓ-buffer-write* in C. A *block write* by $\mathcal{Q}$ *from* C is an execution, starting from C, in which each process in $\mathcal{Q}$ takes exactly one step. If a block write is performed that includes ℓ different *ℓ-buffer-write* instructions to the same memory location, and then some process performs *ℓ-buffer-read* on that location, the process gets the same result regardless of the value of that location in C.

To obtain the lower bound, we extend the proof of the $n-1$ lower bound on the number of registers required for solving n-process consensus [Zhu16]. We also borrow intuition about reserving executions from the $\Omega(n)$ lower bound for anonymous consensus [Gel15]. The following auxiliary lemma is largely unchanged from [Zhu16]. The main difference is that we only perform block writes on ℓ-buffers that are ℓ-covered by $\mathcal{P}$.

Lemma 14. *Let C be a configuration and let $\mathcal{Q}$ be a set of processes that is bivalent from C. Suppose C is at most ℓ-covered by a set of processes $\mathcal{R}$, where $\mathcal{R} \cap \mathcal{Q} = \emptyset$. Let L be the set of locations that are ℓ-covered by $\mathcal{R}$ in C. Let β be a block write from C by the set of $\ell \cdot |L|$ processes in $\mathcal{R}$ that cover L. Then there exists a $\mathcal{Q}$-only execution ξ from C such that $\mathcal{R} \cup \mathcal{Q}$ is bivalent from $C\xi\beta$ and, in configuration $C\xi$, some process in $\mathcal{Q}$ covers a location not in L.*

Proof. Suppose process $p \in \mathcal{R}$ can decide value $v \in \{0, 1\}$ from configuration $C\beta$. Since Q is bivalent from C, there is a $\mathcal{Q}$-only execution, ζ, from C in which $\bar{v}$ is decided. Let ξ be the longest prefix of ζ such that p can decide v from $C\xi\beta$. Let δ be the next step by $q \in \mathcal{Q}$ in ζ after ξ.

If δ is an *ℓ-buffer-read* or is an *ℓ-buffer-write* to a location in L, then $C\xi\beta$ and $C\xi\delta\beta$ are indistinguishable to p. This is impossible, since p can decide v from $C\xi\beta$, but cannot decide v from $C\xi\delta\beta$. Thus, δ is an ℓ-*buffer-write* to a location not in L, in configuration $C\xi$, q covers a location not in L, and $C\xi\beta\delta$ is indistinguishable from $C\xi\delta\beta$ to process p. Therefore, by definition of ξ, p cannot decide v from $C\xi\beta\delta$, so it can decide $\bar{v}$ from $C\xi\beta\delta$. Since p can decide v from $C\xi\beta$, it follows This implies that $\{p,q\}\subseteq \mathcal{R}\cup\mathcal{Q}$ is bivalent from $C\xi\beta$. □

From a configuration that is at most ℓ-covered by a set of processes $\mathcal{R}$, we show how to reach another configuration that is at most ℓ-covered by $\mathcal{R}$ and in which another process $z \notin \mathcal{R}$ covers a location that is not ℓ-covered by $\mathcal{R}$. This is similar to the induction used by Zhu [Zhu16].

Lemma 15. *Let C be a configuration and let $\mathcal{P}$ be a set of $n \geq 2$ processes. If $\mathcal{P}$ is bivalent from C, then there is a $\mathcal{P}$-only execution α starting from C and a set $\mathcal{Q} \subseteq \mathcal{P}$ of two processes such that $\mathcal{Q}$ is bivalent from $C\alpha$ and $C\alpha$ is at most ℓ-covered by the remaining processes $\mathcal{P}-\mathcal{Q}$.*

Proof. By induction on n. The base case is when $n = 2$. Let $\mathcal{Q} = \mathcal{P}$ and let α be the empty execution. Since $\mathcal{P}-\mathcal{Q} = \emptyset$, the claim holds.

Now let $n > 2$ and suppose the claim holds for $n-1$. By Lemma 2, there exist a $\mathcal{P}$-only execution γ starting from C and a set $\mathcal{Q} \subset \mathcal{P}$ of two processes that is bivalent from $D = C\gamma$. Pick any process $z \in \mathcal{P}-\mathcal{Q}$. Then $\mathcal{P}-\{z\}$ is bivalent from D because $\mathcal{Q}$ is bivalent from D.

We construct a sequence of configurations $D_0, D_1, \ldots$ reachable from D such that, for all $i \geq 0$, the following properties hold:

1. there exists a set of two processes $\mathcal{Q}_i \subseteq \mathcal{P}-\{z\}$ such that $\mathcal{Q}_i$ is bivalent from D_i,
2. D_i is at most ℓ-covered by the remaining processes $\mathcal{R}_i = (\mathcal{P}-\{z\})-\mathcal{Q}_i$, and
3. if L_i is the set of locations that are ℓ-covered by $\mathcal{R}_i$ in D_i, then D_{i+1} is reachable from D_i by a $(\mathcal{P}-\{z\})$-only execution α_i which contains a block write β_i to L_i by $\ell\cdot|L_i|$ processes in $\mathcal{R}_i$.

By the induction hypothesis applied to D and $\mathcal{P}-\{z\}$, there is a $(\mathcal{P}-\{z\})$-only execution η starting from D and a set $\mathcal{Q}_0 \subseteq (\mathcal{P}-\{z\})$ of two processes such that $\mathcal{Q}_0$ is bivalent from $D_0 = D\eta$ and D_0 is at most ℓ-covered by $\mathcal{R}_0 = (\mathcal{P}-\{z\})-\mathcal{Q}_0$.

Now suppose that D_i is a configuration reachable from D and $\mathcal{Q}_i$ and $\mathcal{R}_i$ are sets of processes that satisfy all three conditions.

By Lemma 14 applied to configuration D_i, there is a $\mathcal{Q}_i$-only execution ξ_i such that $\mathcal{R}_i\cup\mathcal{Q}_i = \mathcal{P}-\{z\}$ is bivalent from $D_i\xi_i\beta_i$, where β_i is a block write to L_i by $\ell\cdot|L_i|$ processes in $\mathcal{R}_i$. Applying the induction hypothesis to $D_i\xi_i\beta_i$ and $\mathcal{P}-\{z\}$, we get a $(\mathcal{P}-\{z\})$-only execution ψ_i leading to a configuration $D_{i+1} = D_i\xi_i\beta_i\psi_i$, in which there is a set, $\mathcal{Q}_{i+1}$, of two processes such that $\mathcal{Q}_{i+1}$ is bivalent from D_{i+1}. Additionally, D_{i+1} is at most ℓ-covered by the set of remaining processes $\mathcal{R}_{i+1} = (\mathcal{P}-\{z\})-\mathcal{Q}_{i+1}$. Note that the execution $\alpha_i = \xi_i\beta_i\psi_i$ contains the block write β_i to L_i by $\ell\cdot|L_i|$ processes in $\mathcal{R}_i$.

Since there are only finitely many locations, there exist $0 \leq i < j$ such that $L_i = L_j$. Next, we insert steps of z that cannot be detected by any process in $\mathcal{P}-\{z\}$. Consider any $\{z\}$-only execution ζ from $D_i\xi_i$ that decides a value $v \in \{0,1\}$. If ζ does not contain any ℓ-*buffer-write* to locations outside L_i, then $D_i\xi_i\zeta\beta_i$ is indistinguishable from $D_i\xi_i\beta_i$ to processes in $\mathcal{P}-\{z\}$. Since $D_i\xi_i\beta_i$ is bivalent for $\mathcal{P}-\{z\}$, there exists a $\mathcal{P}-\{z\}$ execution from $D_i\xi_i\beta_i$ and, hence, from $D_i\xi_i\zeta\beta_i$ that decides $\bar{v}$, contradicting agreement. Thus ζ contains an ℓ-*buffer-write* to a location outside L_i. Let ζ' be the longest prefix of ζ that does not contain an ℓ-*buffer-write* to a location outside L_i. Then, in $D_i\xi_i\zeta'$, z is poised to perform an ℓ-*buffer-write* to a location outside $L_i = L_j$.

Configuration $D_i\xi_i\zeta'\beta_i$ is indistinguishable from $D_i\xi_i\beta_i$ to $\mathcal{P}-\{z\}$, so the $(\mathcal{P}-\{z\})$-only execution $\psi_i\alpha_{i+1}\cdots\alpha_{j-1}$ can be applied from $D_i\xi_i\zeta'\beta_i$. Let $\alpha = \gamma\eta\alpha_0\cdots\alpha_{i-1}\xi_i\zeta'\beta_i\psi_i\alpha_{i+1}\cdots\alpha_{j-1}$. Every process in $\mathcal{P}-\{z\}$ is in the same state in $C\alpha$ as it is in D_j. In particular, $\mathcal{Q}_j \subseteq \mathcal{P}-\{z\}$ is bivalent from D_j and, hence, from $C\alpha$. Every location is at most ℓ-covered by $\mathcal{R}_j = (P-\{z\})-\mathcal{Q}_j$ in D_j and, hence, in $C\alpha$. Moreover, since z takes no steps after $D_i\xi_i\zeta'$, z covers a location not in L_j in configuration $C\alpha$. Therefore, every location is at most ℓ-covered by $\mathcal{R}_j\cup\{z\} = \mathcal{P}-\mathcal{Q}_j$ in $C\alpha$. □

Finally, we can prove the main theorem.

Theorem 16. *Consider a memory consisting of ℓ-buffers. Then any obstruction-free binary consensus algorithm for n processes uses at least $\lceil (n-1)/\ell\rceil$ locations.*

Proof. Consider any obstruction-free binary consensus algorithm for n processes. By Lemma 1, there exists an initial configuration from which the set of all n processes, $\mathcal{P}$, is bivalent. Lemma 15 implies that there is a configuration, C, reachable from this initial configuration and a set, $\mathcal{Q} \subseteq \mathcal{P}$, of two processes such that $\mathcal{Q}$ is bivalent from C and C is at most ℓ-covered by the remaining processes $\mathcal{R} = \mathcal{P}-\mathcal{Q}$. By the pigeonhole principle, $\mathcal{R}$ covers at least $\lceil (n-2)/\ell\rceil \geq \lceil (n-1)/\ell\rceil - 1$ different locations.

Suppose that $\mathcal{R}$ covers exactly $\lceil (n-2)/\ell\rceil$ different locations and $\lceil (n-2)/\ell\rceil < \lceil (n-1)/\ell\rceil$. Then $n-2$

is a multiple of ℓ and every location covered by $\mathcal{R}$ is, in fact, ℓ-covered by $\mathcal{R}$. Since $\mathcal{Q}$ is bivalent from C, Lemma 14 implies that there is a $\mathcal{Q}$-only execution ξ such that some process in $\mathcal{Q}$ covers a location that is not covered by $\mathcal{R}$. Hence, there are at least $\lceil (n-2)/\ell \rceil + 1 = \lceil (n-1)/\ell \rceil$ locations. □

The lower bound in Theorem 16 can be extended to a *heterogeneous setting*, where the lengths of the buffers are not necessarily the same. To do so, we extend the definition of a configuration C being at most ℓ-covered by a set of processes $\mathcal{P}$. Instead, we require that the number of processes in $\mathcal{P}$ covering each buffer is at most the length of that buffer. Then we consider block writes to a set of locations containing ℓ different *ℓ-buffer-write* operations to each ℓ-buffer in the set. The general result is that, for any algorithm which solves consensus for n processes and satisfies nondeterministic solo termination, the sum of the lengths of all buffers must be at least $n-1$.

The lower bound also applies to systems in which the return value of every non-trivial instruction on a memory location does not depend on the value of that location and the return value of any trivial instruction is a function of the sequence of the preceding ℓ non-trivial instructions performed on the location. This is because such instructions can be implemented by *ℓ-buffer-read* and *ℓ-buffer-write* instructions. We record each invocation of a non-trivial instruction using *ℓ-buffer-write*. The return values of these instructions can be determined locally. To implement a trivial instruction, we perform *ℓ-buffer-read*, which returns a sequence containing the description of the last ℓ non-trivial instructions performed on the location. This is sufficient to determine the correct return value.

7 Multiple Assignment

In this section, we explore whether multiple assignment can improve the space complexity of solving obstruction-free consensus. A motivation for this question is that obstruction-free multiple assignment can be easily implemented using a simple transaction.

We prove a lower bound that is similar to the lower bound in Section 6.2. Suppose the *ℓ-buffer-read*() and *ℓ-buffer-write*(x) instructions are supported on every memory location in a system and, for any subset of locations, a process can atomically perform one *ℓ-buffer-write* instruction per location. Then $\lceil n/2\ell \rceil$ locations are necessary for n processes to solve binary consensus. As in Section 6.2, this result can be further generalized to a heterogeneous setting.

The main technical difficulty is proving an analogue of Lemma 14. In the absence of multiple assignment, if β is a block write to a set of ℓ-covered locations, L, and δ is an *ℓ-buffer-write* to a location not in L, then β and δ commute (in the sense that the configurations resulting from performing $\beta\delta$ and $\delta\beta$ are indistinguishable to all processes). However, a multiple assignment δ can atomically perform *ℓ-buffer-write* to many locations, including locations in L. Thus, it may be possible for processes to distinguish between $\beta\delta$ and $\delta\beta$. Using a careful combinatorial argument, we construct two blocks of multiple assignments, β_1 and β_2, such that, in each block, *ℓ-buffer-write* is performed at least ℓ times on each location in L and is not performed on any location outside of L. Given this, we can show that $\beta_1\delta\beta_2$ and $\delta\beta_1\beta_2$ are indistinguishable to all processes. This is enough to prove an analogue of Lemma 14.

First, we define a notion of covering for this setting. In configuration C, process p *covers* location r if p is poised to perform a multiple assignment that includes an *ℓ-buffer-write* to r. The next definition is key to our proof. Suppose that, in some configuration C, each process in $\mathcal{P}$ is poised to perform a multiple assignment. A *k-packing* of $\mathcal{P}$ in C is a function π mapping each process in $\mathcal{P}$ to some memory location it covers such that no location r has more than k processes mapped to it (i.e., $|\pi^{-1}(r)| \leq k$). When $\pi(p) = r$ we say that π *packs* p in location r. A k-packing may not always exist or there may be many k-packings, depending on the configuration, the set of processes, and the value of k. A location r is *fully k-packed* by $\mathcal{P}$ in configuration C, if there is a k-packing of $\mathcal{P}$ in C and all k-packings of $\mathcal{P}$ in C pack exactly k processes in r.

Suppose that, in some configuration, there are two k-packings of the same set of processes, but the first packs more processes in some location r than the second. We show there is a location r' in which the first packing packs fewer processes than the second and there is a k-packing which, as compared to the first packing, packs one less process in location r, one more process in location r', and the same number of processes in all other locations. The proof relies on existence of a certain Eulerian path in a multigraph that we build to represent these two k-packings.

Lemma 17. *Suppose g and h are two k-packings of the same set of processes $\mathcal{P}$ in some configuration C and r_1 is a location such that $|g^{-1}(r_1)| > |h^{-1}(r_1)|$ (i.e., g packs more processes in r_1 than h does). Then, there exists a sequence of locations, $r_1, r_2, \ldots, r_t$, and a sequence of distinct processes, $p_1, p_2, \ldots, p_{t-1}$, such that $|h^{-1}(r_t)| > |g^{-1}(r_t)|$ (i.e., h packs more processes in r_t than g), and $g(p_i) = r_i$ and $h(p_i) = r_{i+1}$ for $1 \leq i \leq t-1$. Moreover, for $1 \leq j < t$, there exists a*

k-packing g' such that g' packs one less process than g in r_j, g' packs one more process than g in r_t, g' packs the same number of processes as g in all other locations, and $g'(q) = g(q)$ for all $q \notin \{p_j, \ldots, p_{t-1}\}$.

Proof. Consider a multigraph with one node for each memory location in the system and one directed edge from node $g(p)$ to node $h(p)$ labelled by p, for each process $p \in \mathcal{P}$. The in-degree of any node v is $|h^{-1}(v)|$, which is the number of processes that are packed into memory location v by h, and the out-degree of node v is $|g^{-1}(v)|$, which is the number of processes that are packed in v by g.

Now, consider any maximal Eulerian path in this multigraph starting from the node r_1. This path consists of a sequence of distinct edges, but may visit the same node multiple times. Let $r_1, \ldots, r_t$ be the sequence of nodes visited and let p_i be the labels of the traversed edges, in order. Then $g(p_i) = r_i$ and $h(p_i) = r_{i+1}$ for $1 \leq i \leq t-1$. The edges in the path are all different and each is labelled by a different process, so the path has length at most $|\mathcal{P}|$. By maximality, the last node in the sequence must have more incoming edges than outgoing edges, so $|h^{-1}(r_t)| > |g^{-1}(r_t)|$.

Let $1 \leq j < t$. We construct g' from g by re-packing each process p_i from r_i to r_{i+1} for all $j \leq i < t$. Then $g'(p_i) = r_{i+1}$ for $j \leq i < t$ and $g'(p) = g(p)$ for all other processes p. Notice that p_i covers r_{i+1}, since $h(p_i) = r_{i+1}$ and h is a k-packing. As compared to g, g' packs one less process in r_j, one more process in r_t, and the same number of processes in every other location. Since h is a k-packing, it packs at most k processes in r_t. Because g is a k-packing that packs less processes in r_t than h, g' is also a k-packing. □

Let $\mathcal{P}$ be a set of processes, each of which is poised to perform a multiple assignment in some configuration C. A *block multi-assignment* by $\mathcal{P}$ from C is an execution starting at C, in which each process in $\mathcal{P}$ takes exactly one step.

Consider some configuration C and a set of processes $\mathcal{R}$ such that there is a 2ℓ-packing π of $\mathcal{R}$ in C. Let L be the set of all locations that are fully 2ℓ-packed by $\mathcal{R}$ in C, so π packs exactly 2ℓ processes from $\mathcal{R}$ in each location $r \in L$. Partition the $2\ell \cdot |L|$ processes packed by π in L into two sets, $\mathcal{R}^1$ and $\mathcal{R}^2$, each containing $\ell \cdot |L|$ processes, such that, for each location $r \in L$, ℓ of the processes packed in r by π belong to $\mathcal{R}^1$ and the other ℓ belong to $\mathcal{R}^2$. For $i \in \{1, 2\}$, let β_i be a block multi-assignment by $\mathcal{R}^i$.

Notice that, for any location $r \in L$, the outcome of any *ℓ-buffer-read* on r after β_i does not depend on multiple assignments that occurred prior to β_i. Moreover, we can prove the following crucial property about these block multi-assignments to fully packed locations.

Lemma 18. *Neither β_1 nor β_2 involves an ℓ-buffer-write to a location outside of L.*

Proof. Assume the contrary. Let $q \in \mathcal{R}^1 \cup \mathcal{R}^2$ be a process with $\pi(q) \in L$ such that, in C, q also covers some location $r_1 \notin L$. If $|\pi^{-1}(r_1)| < 2\ell$, then there is another 2ℓ packing of $\mathcal{R}$ in C, which is the same as π, except that it packs q in location r_1 instead of $\pi(q)$. However, this packing packs fewer than 2ℓ processes in $\pi(q) \in L$, contradicting the definition of L. Therefore $|\pi^{-1}(r_1)| = 2\ell$, i.e., π packs exactly 2ℓ processes in r_1.

Since L is the set of all fully 2ℓ-packed locations, there exists a 2ℓ-packing h, which packs strictly fewer than 2ℓ processes in $r_1 \notin L$. From Lemma 17 with $g = \pi$ and $k = 2\ell$, there are sequences of locations, $r_1, \ldots, r_t$, and processes, $p_1, \ldots, p_{t-1}$, such that $|h^{-1}(r_t)| > |\pi^{-1}(r_t)|$. Since h is a 2ℓ-packing, it packs at most 2ℓ processes in r_t and, hence, π packs strictly less than 2ℓ processes in r_t. Thus, $r_t \notin L$. We consider two cases.

First, suppose that $q \neq p_i$ for all $i = 1, \ldots, t-1$, i.e., q does not occur in the sequence $p_1, \ldots, p_{t-1}$. By the second part of Lemma 17 with $j = 1$, there is a 2ℓ-packing π' that packs less than 2ℓ processes in r_1, one more process than π in r_t, and the same number of processes as π in all other locations. In particular, π' packs exactly 2ℓ processes in each location in L, including $\pi(q)$. Moreover, $\pi'(q) = \pi(q)$, since q does not occur in the sequence $p_1, \ldots, p_{t-1}$. Consider another 2ℓ packing of $\mathcal{R}$ in C, which is the same as π', except that it packs q in location r_1 instead of location $\pi(q)$. However, this packing packs fewer than 2ℓ processes in $\pi(q) \in L$, contradicting the definition of L.

Now, suppose that $q = p_s$, for some $s \in \{1, \ldots, t-1\}$. Since $r_s = \pi(p_s) = \pi(q) \in L$, it follows that $|\pi^{-1}(r_s)| = 2\ell$. By the second part of Lemma 17 with $j = s$, there is a 2ℓ-packing that packs less than 2ℓ processes in r_s, one more process than π in r_t, and the same number of processes as π in all other locations. Since $r_s \in L$, this contradicts the definition of L.

Thus, in configuration C, every process in $\mathcal{R}^1 \cup \mathcal{R}^2$ only covers locations in L. □

With C, $\mathcal{R}$, β_1, and β_2 as defined prior to Lemma 18, we now prove a lemma that replaces Lemma 14.

Lemma 19. *Let $\mathcal{Q}$ be a set of processes disjoint from $\mathcal{R}$ that is bivalent from C. Then there exists a $\mathcal{Q}$-only execution ξ from C such that $\mathcal{R} \cup \mathcal{Q}$ is bivalent from $C\xi\beta_1$ and, in configuration $C\xi$, some process in $\mathcal{Q}$ covers a location not in L.*

Proof. Suppose process $p \in \mathcal{R}$ can decide value $v \in \{0,1\}$ from configuration $C\beta_1\beta_2$ and ζ is a $\mathcal{Q}$-only execution from C in which $\bar{v}$ is decided. Let ξ be the longest prefix of ζ such that p can decide v from $C\xi\beta_1\beta_2$. Let δ be the next step by $q \in \mathcal{Q}$ in ζ after ξ.

If δ is an ℓ-*buffer-read* or a multiple assignment involving only ℓ-*buffer-write* operations to locations in L, then $C\xi\beta_1\beta_2$ and $C\xi\delta\beta_1\beta_2$ are indistinguishable to p. Since p can decide v from $C\xi\beta_1\beta_2$, but cannot decide v from $C\xi\delta\beta_1\beta_2$, δ must be a multiple assignment that includes an ℓ-*buffer-write* to a location not in L. Thus, in configuration $C\xi$, q covers a location not in L. For each location $r \in L$, the value of r is the same in $C\xi\delta\beta_1\beta_2$ as it is in $C\xi\beta_1\delta\beta_2$ due to the block multi-assignment β_2. By Lemma 18, for each location $r \notin L$, neither β_1 nor β_2 performs an ℓ-*buffer-write* to r, so the value of r is the same in $C\xi\delta\beta_1\beta_2$ as it is in $C\xi\beta_1\delta\beta_2$. Since the state of process p is the same in configuration $C\xi\beta_1\delta\beta_2$ and $C\xi\delta\beta_1\beta_2$, these two configurations are indistinguishable to p.

By definition of ξ, p cannot decide v from $C\xi\delta\beta_1\beta_2$ and, hence, it can decide $\bar{v}$ from $C\xi\delta\beta_1\beta_2$. Therefore, p can decide $\bar{v}$ from $C\xi\beta_1\delta\beta_2$. Since p can decide v from $C\xi\beta_1\beta_2$, it follows that $\mathcal{R}\cup\mathcal{Q}$ is bivalent from $C\xi\beta_1$. □

Using these tools, we can prove the following analogue of Lemma 15.

Lemma 20. *Let C be a configuration and let $\mathcal{P}$ be a set of $n \geq 2$ processes. If $\mathcal{P}$ is bivalent from C, then there is a $\mathcal{P}$-only execution α and a set $\mathcal{Q} \subseteq \mathcal{P}$ of two processes such that $\mathcal{Q}$ is bivalent from $C\alpha$ and there exists a 2ℓ-packing π of the remaining processes $\mathcal{P} - \mathcal{Q}$ in $C\alpha$.*

Proof. By induction on n. The base case is when $n = 2$. Let $\mathcal{Q} = \mathcal{P}$ and let α be the empty execution. Since $\mathcal{P} - \mathcal{Q} = \emptyset$, the claim holds.

Now let $n > 2$ and suppose the claim holds for $n-1$. By Lemma 2, there exists a $\mathcal{P}$-only execution γ starting from C and a set $\mathcal{Q} \subset \mathcal{P}$ of two processes that is bivalent from $D = C\gamma$. Pick any process $z \in \mathcal{P}-\mathcal{Q}$. Then $\mathcal{P}-\{z\}$ is bivalent from D because $\mathcal{Q}$ is bivalent from D.

We construct a sequence of configurations $D_0, D_1, \ldots$ reachable from D, such that, for all $i \geq 0$, the following properties hold:

1. there exists a set of two processes $\mathcal{Q}_i \subseteq \mathcal{P}-\{z\}$ such that $\mathcal{Q}_i$ is bivalent from D_i,
2. there exists a 2ℓ-packing π_i of the remaining processes $\mathcal{R}_i = (\mathcal{P} - \{z\}) - \mathcal{Q}_i$ in D_i, and
3. if L_i is the set of all locations that are fully 2ℓ-packed by $\mathcal{R}_i$ in D_i, then D_{i+1} is reachable from D_i by a $(\mathcal{P} - \{z\})$-only execution α_i which contains a block multi-assignment β_i such that, for each location $r \in L_i$, there are at least ℓ multiple assignments in β_i that perform ℓ-*buffer-write* on r.

By the induction hypothesis applied to D and $\mathcal{P} - \{z\}$, there is a $(\mathcal{P} - \{z\})$-only execution η starting from D and a set $\mathcal{Q}_0 \subseteq (\mathcal{P} - \{z\})$ of two processes such that $\mathcal{Q}_0$ is bivalent from $D_0 = D\eta$ and and there exists a 2ℓ-packing π_0 of the remaining processes $\mathcal{R}_0 = (\mathcal{P} - \{z\}) - \mathcal{Q}_0$ in D_0.

Now suppose that D_i is a configuration reachable from D and $\mathcal{Q}_i$ and $\mathcal{R}_i$ are sets of processes that satisfy all three conditions.

By Lemma 19 applied to configuration D_i, there is a $\mathcal{Q}_i$-only execution ξ_i such that $\mathcal{R}_i \cup \mathcal{Q}_i = \mathcal{P}-\{z\}$ is bivalent from $D_i\xi_i\beta_i$, where β_i is a block multi-assignment in which ℓ-*buffer-write* is performed at least ℓ times on r, for each location $r \in L_i$. Applying the induction hypothesis to $D_i\xi_i\beta_i$ and $\mathcal{P} - \{z\}$, we get a $(\mathcal{P} - \{z\})$-only execution ψ_i leading to a configuration $D_{i+1} = D_i\xi_i\beta_i\psi_i$, in which there is a set, $\mathcal{Q}_{i+1}$, of two processes such that $\mathcal{Q}_{i+1}$ is bivalent from D_{i+1}. Additionally, there exists a 2ℓ-packing π_{i+1} of the remaining processes $\mathcal{R}_{i+1} = (\mathcal{P} - \{z\}) - \mathcal{Q}_{i+1}$ in D_{i+1}. Note that the execution $\alpha_i = \xi_i\beta_i\psi_i$ contains the block multi-assignment β_i.

Since there are only finitely many locations, there exists $0 \leq i < j$ such that $L_i = L_j$, i.e., the set of fully 2ℓ-packed locations by $\mathcal{R}_i$ in D_i is the same as the set of fully 2ℓ-packed locations by $\mathcal{R}_j$ in D_j. Next, we insert steps of z that cannot be detected by any process in $\mathcal{P} - \{z\}$. Consider any $\{z\}$-only execution ζ from $D_i\xi_i$ that decides a value $v \in \{0,1\}$. If ζ does not contain any ℓ-*buffer-write* to locations outside L_i, then $D_i\xi_i\zeta\beta_i$ is indistinguishable from $D_i\xi_i\beta_i$ to processes in $\mathcal{P} - \{z\}$. Since $D_i\xi_i\beta_i$ is bivalent for $\mathcal{P} - \{z\}$, there exists a $\mathcal{P}-\{z\}$ execution from $D_i\xi_i\beta_i$ and, hence, from $D_i\xi_i\zeta\beta_i$ that decides $\bar{v}$, contradicting agreement. Thus ζ contains an ℓ-*buffer-write* to a location not in L_i. Let ζ' be the longest prefix of ζ that does not contain an ℓ-*buffer-write* to a location outside L_i. Then, in $D_i\xi_i\zeta'$, z is poised to perform a multiple assignment containing an ℓ-*buffer-write* to a location outside $L_i = L_j$.

Configuration $D_i\xi_i\zeta'\beta_i$ is indistinguishable from $D_i\xi_i\beta_i$ to $\mathcal{P}-\{z\}$, so the $(\mathcal{P}-\{z\})$-only execution $\psi_i\alpha_{i+1}\cdots\alpha_{j-1}$ can be applied from $D_i\xi_i\zeta'\beta_i$. Let $\alpha = \gamma\eta\alpha_0\cdots\alpha_{i-1}\xi_i\zeta'\beta_i\psi_i\alpha_{i+1}\cdots\alpha_{j-1}$. Every process in $\mathcal{P}-\{z\}$ is in the same state in $C\alpha$ as it is in D_j. In particular, $\mathcal{Q}_j \subseteq \mathcal{P}-\{z\}$ is bivalent from D_j and, hence, from $C\alpha$. The 2ℓ-packing π_j of $\mathcal{R}_j$ in D_j is a 2ℓ-packing of $\mathcal{R}_j$ in $C\alpha$ and $L_i = L_j$ is the set of locations that are fully 2ℓ-packed by $\mathcal{R}_j$ in $C\alpha$. Since z takes no steps after $D_i\xi\zeta'$, z covers a location r not in L_j in configuration $C\alpha$. Since $r \notin L_j$, there is a 2ℓ-packing π'_j of $\mathcal{R}_j$ in $C\alpha$ which packs less than 2ℓ processes into r. Let π be the packing that packs z into location r and packs each process in $\mathcal{R}_j$ in the same location as π'_j does. Then π is a 2ℓ-packing of $\mathcal{R}_j \cup \{z\} = \mathcal{P} - \mathcal{Q}_j$ in $C\alpha$. □

We can now prove the main theorem.

Theorem 21. *Consider a memory consisting of ℓ-buffers, in which each process can atomically perform ℓ-buffer-write to any subset of the ℓ-buffers. Then any obstruction-free binary consensus algorithm for n processes uses at least $\lceil (n-1)/2\ell \rceil$ locations.*

Proof. Consider any obstruction-free binary consensus algorithm for n processes. By Lemma 1, there exists an initial configuration from which the set of all n processes, $\mathcal{P}$, is bivalent. Lemma 20 implies that there is a configuration, C, reachable from this initial configuration, a set of two processes $\mathcal{Q} \subseteq \mathcal{P}$ such that $\mathcal{Q}$ is bivalent from C, and a 2ℓ-packing π of the remaining processes $\mathcal{R} = \mathcal{P} - \mathcal{Q}$ in C. By the pigeonhole principle, $\mathcal{R}$ covers at least $\lceil (n-2)/2\ell \rceil$ different locations.

Suppose that $\mathcal{R}$ covers exactly $\lceil (n-2)/2\ell \rceil$ different locations and $\lceil (n-2)/2\ell \rceil < \lceil (n-1)/2\ell \rceil$. Then $n-2$ is a multiple of 2ℓ and every location is fully 2ℓ-packed by $\mathcal{R}$. Since $\mathcal{Q}$ is bivalent from C, Lemma 19 implies that there is a $\mathcal{Q}$-only execution ξ such that some process in $\mathcal{Q}$ covers a location that is not fully 2ℓ-packed by $\mathcal{R}$. Hence, there are at least $\lceil (n-2)/2\ell \rceil + 1 = \lceil (n-1)/2\ell \rceil$ locations. □

8 Swap and Read

The *swap*(x) instruction atomically sets the memory location to have value x and returns the value that it previously contained. In this section, we present Algorithm 1, an anonymous obstruction-free algorithm for solving n-consensus using $n-1$ shared memory locations, $X_1, \ldots, X_{n-1}$, which support *read* and *swap*. It is similar in spirit to the racing counters algorithm presented in the proof of Lemma 3, but more complicated. The main idea is that one register can be eliminated by having each process use the information its gains when it performs a *swap*.

It is possible to implement a linearizable, obstruction-free *scan* of the $n-1$ shared memory locations, by having each process include its process identifier and a strictly increasing sequence number as part of the argument of each *swap* it performs. As in the double collect algorithm [AAD+93], a process repeatedly collects the values in all the locations (using *read*) until it observes two consecutive collects with the same values. In addition to a process identifier and a sequence number (which we will henceforth ignore), each shared memory location stores a vector of n non-negative integers, all of which are initially 0.

Intuitively, the processes view the possible input values as competing to complete *laps*. Each process has a local variable, $\ell \in \mathbb{N}^n$, storing, for each value $v \in \{0, \ldots, n-1\}$, the lap $\ell_v \geq 0$ that the process thinks v is on. The process updates ℓ after performing a *scan*. Initially, these are all 0. A process with input x begins by setting ℓ_x to 1. Then it repeatedly tries to complete a lap for a value that it thinks is in the lead. When it thinks that some value has a substantial lead on all other values, it decides that value. Each process also has two other local variables, $\vec{a}$, in which it stores the result of its last *scan*$(X_1, \ldots, X_{n-1})$ and s, in which it stores the value returned by its last *swap* operation. The i'th component $\vec{a}(i)$ of $\vec{a}$ is the vector in $\mathbb{N}^n$ that it last read from X_i. Initially, s contains a vector of n zeros.

In its first step, a process performs a *scan* of all $n-1$ memory locations. Then, for each value v, it updates the lap, ℓ_v, that it thinks v is on to be the maximum among ℓ_v, the v'th component, s_v, of s, and the v'th component, $\vec{a}(i)_v$, of the vector in memory location i when the *scan* was performed, for $1 \leq i \leq n-1$. Next, it chooses its preferred value, v^* to be a value it thinks is on the largest lap (breaking ties in favour of smaller values). If there is a memory location that does not contain ℓ, the process performs *swap*(ℓ) on the first such location. Now suppose all the memory locations contain ℓ. If ℓ_{v^*} is at least 2 larger than every other component of ℓ, the process decides v^*. Otherwise, it considers v^* to have completed lap ℓ_{v^*}, it increments ℓ_{v^*}, performs *swap*(ℓ) on X_1, and repeats this sequence of steps.

Fix an execution of the algorithm. For each *scan*, S, by a process p, let $\ell(S)$ be the value of p's local variable ℓ immediately after the for loop on lines 6–8 following S. For each value $v \in \{0, \ldots, n-1\}$, let $\ell_v(S)$ denote component v of this n-component vector. Similarly, for every *swap*, U, by a process p let $\ell(U)$ be the value of p's local variable ℓ immediate before U is performed on line 21 and let $\ell_v(U)$ denote its component v.

We now prove that the algorithm is correct. We begin with some easy observations, which follow from inspection of the code.

Observation 22. *Let U be a swap by some process p and let S be any scan that p performed before U. Then $\ell_v(U) \geq \ell_v(S)$ for every value $v \in \{0, \ldots, n-1\}$.*

Observation 23. *Let U be a swap by some process p and let S be the last scan that p performed before U. If there exists a value $v \in \{0, \ldots, n-1\}$ such that $\ell_v(U) > \ell_v(S)$, then $\ell_v(U) = \ell_v(S) + 1$, $\ell_{v'}(S) \leq \ell_v(S)$ for all other values $v' \neq v$, and S returned $\ell(S)$ from each shared memory location.*

Algorithm 1 An anonymous n-consensus algorithm for a process with input value x using *swap* and *scan*.

```
1:  $\ell \leftarrow \vec{0}$
2:  $\ell_x \leftarrow 1$
3:  $s \leftarrow \vec{0}$
4:  loop
5:      $\vec{a} \leftarrow scan(X_1, \ldots, X_{n-1})$
6:      for $v \in \{0, 1, \ldots, n-1\}$ do
7:          $\ell_v \leftarrow \max(\{\ell_v, s_v\} \cup \{\vec{a}(j)_v : 1 \leq j \leq n-1\})$
8:      end for
9:      $\ell^* \leftarrow \max\{\ell_0, \ldots, \ell_{n-1}\}$
10:     $v^* \leftarrow \min\{v : \ell_v = \ell^*\}$
11:     if $\vec{a}(j) = \ell$ for all $1 \leq j \leq n-1$ then
12:         ▷ value $v^*$ has completed lap $\ell^*$
13:         if $\ell^* \geq \ell_v + 2$ for all $v \neq v^*$ then
14:             ▷ $v^*$ is at least 2 laps ahead of all other values
15:             decide $v^*$ and terminate
16:         end if
17:         $\ell_{v^*} \leftarrow \ell_{v^*} + 1$
18:         ▷ value $v^*$ is now on the next lap
19:     end if
20:     $j \leftarrow \min\{j : \vec{a}(j) \neq \ell\}$
21:     $s \leftarrow swap(X_j, \ell)$
22: end loop
```

Observation 24. *Let U be a swap by some process p that returned s. Let S be any scan that p performed before U and let S' be any scan that p performed after U. Then $\ell_v(S') \geq \max\{s_v, \ell_v(S)\}$ for every value $v \in \{0, \ldots, n-1\}$.*

The next lemma follows from these observations. It says that if there was a *scan*, S, where value v is on lap $\ell > 0$, i.e. $\ell_v(S) = \ell$, then there was a *scan* where v is on lap $\ell - 1$ and all the swap objects contained this information.

Lemma 25. *Let S be a scan and let $v \in \{0, \ldots, n-1\}$ be a value. If $\ell_v(S) > 0$, then there was a scan, S', performed prior to S such that S' returned $\ell(S')$ from each shared memory location, $\ell_v(S') = \ell_v(S) - 1$, and $\ell_{v'}(S') \leq \ell_v(S')$, for all $v' \neq v$.*

Proof. Since each memory location initially contains an n-component vector of 0's and $\ell_v(S) > 0$, there was *swap* U prior to S such that $\ell_v(U) = \ell_v(S)$. Consider the first such *swap*. Let p be the process that performed U and let S' be the last *scan* performed by p before U. By Observation 22, $\ell_v(U) \geq \ell_v(S')$. If $\ell_v(U) = \ell_v(S')$, there would have been a *swap* U' prior to S' and, hence, prior to U with $\ell_v(U') = \ell_v(S') = \ell_v(U) = \ell_v(S)$, contradicting the definition of U. Therefore $\ell_v(U) > \ell_v(S')$. By Observation 23, it follows that $\ell_v(U) = \ell_v(S') + 1$, $\ell_{v'}(S') \leq \ell_v(S')$ for all other values $v' \neq v$, and S' returned $\ell(S')$ from each shared memory location. Since $\ell_v(U) = \ell_v(S)$, it follows that $\ell_v(S') = \ell_v(U) - 1 = \ell_v(S) - 1$. □

The following lemma is key to the proof of correctness. It says that, if a process performs a *scan* S where all the components have the same value and, as a result, thinks value v has completed lap ℓ, then every process will think that v is at least on lap ℓ when it performs any *scan* after S.

Lemma 26. *Suppose S is a scan that returned $\ell(S)$ from each shared memory location. If T is a scan performed after S, then, for each $v \in \{0, \ldots, n-1\}$, $\ell_v(T) \geq \ell_v(S)$.*

Proof. Suppose, for a contradiction, that there is a *scan* T performed after S such that $\ell_v(T) < \ell_v(S)$ for some $v \in \{0, \ldots, n-1\}$. Consider the first such *scan* T.

By definition of $\ell_v(T)$, T returned a vector $\vec{a}$ such that $\vec{a}(j)_v < \ell_v(S)$ for every component $j \in \{1, \ldots, n-1\}$. Since S returned a vector whose components all contain $\ell(S)$, for each $j \in \{1, \ldots, n-1\}$, some process q_j performed a *swap* U_j on X_j between S and T such that $\vec{a}(j) = \ell(U_j)$.

Let S'_j be the last *scan* prior to U_j by q_j. By Observation 22, $\ell_v(U_j) \geq \ell_v(S'_j)$. Since $\ell_v(S) > \vec{a}(j)_v = \ell_v(U_j)$, it follows that $S \neq S'_j$. If S'_j occurred after S, then, by definition of T, $\ell_v(S'_j) \geq \ell_v(S)$. Thus S'_j occurred before S.

Since processes alternately perform *scan* and *swap*, it follows that $q_1, \ldots, q_{n-1}$ are distinct processes and none of them is the process, p, that performed S. Each of them is poised to perform *swap* to a different memory location immediately after S.

Let $j \in \{1, \ldots, n-1\}$ be arbitrary and let s be the vector returned by U_j. If $s = \vec{\ell}(S)$, then $s_v \geq \ell_v(S)$. So, suppose $s \neq \ell(S)$. Since S returned $\ell(S)$ from each shared memory location, $s = \ell(U')$ for some *swap* U' performed between S and U_j. Consider the process that performed U'. This process is not q_j, since q_j takes no steps between S and U_j. If U' is by p, then, by Observation 22, $s_v \geq \ell_v(S)$. Now suppose U' is by some process $q_i \neq q_j$. Since q_i was poised to perform swap to a different memory location immediately after S, U' occurs after U_i. Let S' be the last *scan* by q_i before U'. Then S' occurs between S and T. Hence, $\ell_v(S') \geq \ell_v(S)$, by definition of T. By Observation 22, $\ell_v(U') \geq \ell_v(S')$. Hence, $s_v = \ell_v(U') \geq \ell_v(S)$.

By Observation 24, $\ell_v(T') \geq s_v \geq \ell_v(S)$ for any *scan* T' performed by q_j after U_j. Since $\ell_v(T) < \ell_v(S)$ and T was performed after U_j, it follows that T was not performed by process q_j.

Note that p is the only process besides $q_1, \ldots, q_{n-1}$. Therefore, T was performed by p. However, by Observation 24, $\ell_v(T) \geq \ell_v(S)$. This is a contradiction. □

The previous lemma allows us to prove that once a value v^* is at a lap ℓ that is 2 laps ahead of $v \neq v^*$ and every swap object contains this information, then v will never reach lap ℓ, so v^* will always be at least one lap ahead of v.

Lemma 27. *Suppose S is a scan that returned $\ell(S)$ from each shared memory location and there is some $v^* \in \{0, \ldots, n-1\}$ such that $\ell_{v^*}(S) \geq \ell_v(S) + 2$ for all other values $v \neq v^*$. Then $\ell_v(T) \leq \ell_v(S) + 1$ for every scan T and every value $v \neq v^*$.*

Proof. Suppose, for a contradiction, that there is some *scan* T and some value $v' \neq v^*$ such that $\ell_{v'}(T) \geq \ell_{v'}(S) + 2$. Consider the first such *scan*. Since $\ell_{v'}(S) + 2 > 0$, Lemma 25 implies there was a *scan*, T', prior to T such that T' returned $\ell(T')$ from each shared memory location, $\ell_{v'}(T') = \ell_{v'}(T) - 1$, and $\ell_{v^*}(T') \leq \ell_{v'}(T')$. By definition of T and T', $\ell_{v'}(T') < \ell_{v'}(S) + 2 \leq \ell_{v'}(T) = \ell_{v'}(T') + 1$, so $\ell_{v'}(S) + 2 = \ell_{v'}(T) = \ell_{v'}(T') + 1$.

If S was performed after T', then, by Lemma 26, $\ell_v(S) \geq \ell_v(T')$ for all $v \in \{0, \ldots, n-1\}$. However, $\ell_{v'}(S) < \ell_{v'}(T')$, so S was performed before T'. Then Lemma 26 implies that $\ell_{v^*}(T') \geq \ell_{v^*}(S)$. By assumption, $\ell_{v^*}(S) \geq \ell_{v'}(S) + 2$. Hence, $\ell_{v^*}(T') \geq \ell_{v'}(S) + 2 = \ell_{v'}(T') + 1$. This contradicts the fact that $\ell_{v^*}(T') \leq \ell_{v'}(T')$. □

We can now prove that the algorithm satisfies agreement, validity, and obstruction-free termination.

Lemma 28. *No two processes decide differently.*

Proof. From the code, the last step performed by a process before deciding value v^* is a *scan*, S, such that S returns $\ell(S)$ from each shared memory location and $\ell_{v^*}(S) \geq \ell_v(S) + 2$ for all other values $v \neq v^*$. Consider the first such *scan* by any process. By Lemma 26, $\ell_{v^*}(T) \geq \ell_{v^*}(S)$ for every *scan* T performed after S. By Lemma 27, $\ell_v(T) \leq \ell_v(S) + 1$ for all $v \neq v^*$. Hence, $\ell_{v^*}(T) > \ell_v(T)$. It follows that no process ever decides $v \neq v^*$. □

Lemma 29. *If some process decides x, then some process has input x.*

Proof. Suppose that some process decides x, but it has input value $x' \neq x$. It initializes its local variable $\ell_{x'} = 1$ and, from the code, $\ell_{x'}$ never decreases. Let $\vec{a}$ be the result of its last *scan*. By line 11, $\vec{a}(j)_x = \ell_x$ for all $1 \leq j \leq n-1$ and, by line 13, $\ell_x \geq \ell_{x'} + 2 > 0$. Initially, every memory location contains a vector of n zeros, so there is a *swap*, U, such that $\ell_x(U) > 0$.

Consider the first such *swap*. Let p be the process that performed U. Prior to U, component x of every shared memory location is 0. Moreover, its local variable s contains a vector of n zeros. Hence, prior to U, p's local variable ℓ_x is not changed by line 7. By lines 2 and 9, its local variable $\ell^* \geq 1$. Thus, if p does not have input value x, then $\ell_x = 0$ immediately prior to U. Then, by definition, $\ell_x(U) = 0$, which is a contradiction. Hence p has input value x. □

Lemma 30. *Every process decides after performing at most $3n - 2$ scans in a solo execution.*

Proof. Let p be any process and consider the first *scan* S performed by p in its solo execution. Let $\ell' = \ell(S)$. After performing at most $n-1$ *swaps*, all with argument ℓ', p will perform a *scan* that returns ℓ' from every shared memory location. Let $v^* = \min\{v : \ell_v(S) \geq \ell_{v'}(S) \text{ for all } v' \neq v\}$. If $\ell'_{v^*} \geq \ell'_v + 2$ for all $v \neq v^*$, then p decides v^*. Otherwise, p performs $n-1$ *swaps*, all with argument ℓ'', where $\ell''_{v^*} = \ell'_{v^*} + 1$ and $\ell''_v = \ell'_v$, for $v \neq v^*$. Then it performs a *scan* that returns a vector whose components all contain ℓ''. If $\ell''_{v^*} \geq \ell''_v + 2$ for all $v \neq v^*$, then p decides v^*. If not, then p performs an additional $n-1$ *swaps*, all with argement ℓ''', where $\ell'''_{v^*} = \ell''_{v^*} + 1 = \ell'_{v^*} + 2$ and $\ell'''_v = \ell''_v = \ell'_v$ for $v \neq v^*$. Finally, p performs a *scan* that returns a vector whose components all contain ℓ'''. Since p performs at most $3(n-1)$ *swaps* and each *swap* is immediately followed by a *scan*, this amounts to $3n - 2$ *scans*, including the first scan, S. □

The preceding lemmas immediately yield the following theorem.

Theorem 31. *There is an anonymous, obstruction-free algorithm for solving consensus among n processes that uses only $n - 1$ memory locations supporting read and swap.*

In [FHS98], there is a proof that $\Omega(\sqrt{n})$ shared memory locations are necessary to solve obstruction-free consensus when the system only supports *swap* and *read* instructions.

9 Test-and-Set and Read

Consider a system that supports only *test-and-set*() and *read*(). It is impossible to solve 2-consensus using only 1 memory location. To see why, first note that, in a solo-terminating execution from an initial configuration, a process must decide its input value, because the other process might have the same input value. Moreover, in every such execution, a process must perform *test-and-set*() on the location. (If not, the location still has value 0 after the execution, so, if it is followed by a solo-terminating execution by the other process with the other input, agreement will be violated.) Then the memory location has value 1 after any solo-terminating execution by a process from an initial configuration, so the value decided by the other process cannot depend on what the first process decided. Hence, there is an execution in which the second process decides a different value, violating agreement.

It is possible to solve wait-free binary consensus for 2 processes using only 2 memory locations, which are both initially 0. In Algorithm 2, a process with input 0 begins by performing *test-and-set*() on memory location M_0 and, if the result is 0, it decides 0. Otherwise, it reads memory location M_1 and decides the value it read. A process with input 1 begins by performing *test-and-set*() on M_1 and, if the result is 1, it decides 1. Otherwise, it performs *test-and-set*() on memory location M_0 and decides the complement of the result.

Algorithm 2 A wait-free 2-consensus algorithm for a process with input value x

```
1: if $x = 0$ then
2:     if $test\text{-}and\text{-}set(M_0) = 0$ then
3:         decide 0
4:     else
5:         if $read(M_1) = 0$ then
6:             decide 0
7:         else decide 1
8:         end if
9:     end if
10: else
11:     if $test\text{-}and\text{-}set(M_1) = 1$ then
12:         decide 1
13:     else
14:         if $test\text{-}and\text{-}test(M_0) = 0$ then
15:             decide 1
16:         else decide 0
17:         end if
18:     end if
19: end if
```

First suppose both processes have input 0. The first process to perform *test-and-set*(M_0) gets 0 and immediately decides 0. The other process gets 1 and then reads 0 from M_1, so it also decides 0.

Now suppose both processes have input 1. The first process to perform *test-and-set*(M_1) gets 0. Then it performs *test-and-set*(M_0), gets 0, and decides 1. The second process to perform *test-and-set*(M_1) gets 1 and immediately decides 1.

Finally, suppose one process, say p_0, has input 0 and the other process, p_1, has input 1. If p_1 takes the first two steps, it performs *test-and-set*(M_1), gets 0, performs *test-and-set*(M_0), gets 0, and returns 1. Then p_0 performs *test-and-set*(M_0), gets 1, reads 1 from M_1, and returns 1. So, suppose that p_1 does not take both of the first two steps. In particular, it does not perform *test-and-set*(M_0) before p_0. Hence, when p_0 performs *test-and-set*(M_0), it gets 0 and immediately decides 0. The first step by p_1 is *test-and-set*(M_1), which returns 0, and its second step is *test-and-set*(M_0), which returns 1. Thus p_1 also decides 0.

However, if the only instructions available are *read*(), *test-and-set*, and *write*(1), then any algorithm for solving obstruction-free binary consensus among $n \geq 3$ processes must use an unbounded number of memory locations.

Lemma 32. *Let p_0, p_1, and q be different processes and let C_0 be a configuration. If $\{p_0, p_1\}$ is bivalent from C_0, then, for every $k \geq 0$, it possible to reach a configuration C_k by a $\{p_0, p_1, q\}$-only execution such that $\{p_0, p_1\}$ is bivalent from C_k and at least k memory locations have been set to 1 in C_k.*

Proof. By induction on k. The base case, $k = 0$, holds trivially for C_0. Given C_k, for some $k \geq 0$, we show how to reach C_{k+1}. By Lemma 2, it is possible to reach a configuration C'_k by a $\{p_0, p_1\}$-only execution from C_k such that p_0 and p_1 decide different values in their solo-terminating executions, γ_0 and γ_1, from C'_k. Without loss of generality, assume that p_0 decides 0 and p_1 decides 1. Let L be the set of memory locations that have been set to 1 in C'_k. By the induction hypothesis, $|L| \geq k$.

Let δ be q's solo-terminating execution from C'_k. If δ does not contain a *test-and-set*() or *write*(1) to a location outside L, then $C'_k\delta$ is indistinguishable from C'_k to $\{p_0, p_1\}$ and, hence, both γ_0 and γ_1 are applicable from $C'_k\delta$. Since some value $v \in \{0, 1\}$ is decided in δ and $\bar{v}$ is decided in $\gamma_{\bar{v}}$, this violates agreement.

So, δ contains at least one *test-and-set*() or *write*(1) to a location outside L. Let δ' be the longest prefix of δ that does not contain such an instruction and let $C = C'_k\delta'$. Then q is poised to perform a *test-and-set* or *write*(1), δ'', to a location $\ell \notin L$ at C. Since δ' does not set any new memory locations to 1, C is indistinguishable from C'_k to $\{p_0, p_1\}$ and, hence, both γ_0 and γ_1 are applicable from C.

If $\{p_0, p_1\}$ is bivalent from $C\delta''$, then $C_{k+1} = C\delta''$ satisfies the claim since $|L \cup \{\ell\}| \geq k + 1$ memory locations are set to 1 in C_{k+1}. So, without loss of generality, suppose that $\{p_0, p_1\}$ is 0-univalent from $C\delta''$. Let γ'_1 be the longest prefix of γ_1 such that p_0 decides 0 from $C\gamma'_1\delta''$. Note that $\gamma'_1 \neq \gamma_1$ since 1 is decided in γ_1. Let γ''_1 be the first step in γ_1 following γ'_1.

If γ''_1 is a *test-and-set*() or *write*(1) to a location in $L \cup \{\ell\}$ or γ''_1 is a *read*, then $C\gamma'_1\gamma''_1\delta''$ is indistinguishable from $C\gamma'_1\delta''$ to p_0. This is impossible, since p_0 decides 0 from $C\gamma'_1\delta''$ and decides 1 from $C\gamma'_1\gamma''_1\delta''$. Thus, γ''_1 is a *test-and-set*() or *write*(1) to a location outside $L \cup \{\ell\}$. It follows that $C\gamma'_1\gamma''_1\delta'' = C\gamma'_1\delta''\gamma''_1$ and, hence, p_0 decides 1 from $C'\gamma'_1\delta''\gamma''_1$.

Let $C_{k+1} = C\gamma_1'\delta''$. Since p_0 decides 0 from C_{k+1}, p_0 decides 1 from $C_{k+1}\gamma_1''$, and γ_1'' is a step by p_1, it follows that $\{p_0, p_1\}$ is bivalent from C_{k+1}. Furthermore, $|L \cup \{\ell\}| \geq k+1$ memory locations are set to 1 in C_{k+1}. Therefore, C_{k+1} satisfies the claim. □

By Lemma 1, there is an initial configuration from which the set of all processes in the system is bivalent. Then it follows from Lemma 32 that any binary consensus algorithm for $n \geq 3$ processes uses an unbounded number of locations.

Theorem 33. *For $n \geq 3$, it is not possible to solve n-consensus using a bounded number of memory locations supporting only read(), test-and-set(), and write(1).*

There is an algorithm for obstruction-free binary consensus that uses an unbounded number of shared memory locations that support only *read*() and *write*(1) [GR05]. All locations are initially 0. The idea is to simulate a counter using an unbounded number of binary registers and then to run the racing counters algorithm presented in Lemma 3. In this algorithm, there are two disjoint, unbounded tracks on which processes race, one for preference 0 and one for preference 1. Each track consists of an unbounded sequence of shared memory locations. To indicate progress, a process performs *write*(1) to the location on its preferred track from which it last read 0. Since the count on each track does not decrease, a process can perform a *scan* using the double collect algorithm [AAD+93]. It is not necessary to read all the locations in a track to determine the count it represents. It suffices to read from the location on the track from which it last read 0, continuing to read from the subsequent locations on the track until it reads another 0. A process changes its preference if it sees that the number of 1's on its preferred track is less than the number of 1's on the other track. Once a process sees that its preferred track is at least 2 ahead of the other track, it decides its current preference.

It is possible to generalize this algorithm to solve n-valued consensus by having n disjoint tracks, each consisting of an unbounded sequence of shared memory locations. Since *test-and-set*() can simulate *write*(1) by ignoring the value returned, we get the following result.

Theorem 34. *It is possible to solve n-consensus using an unbounded number of memory locations supporting only read() and either write(1) or test-and-set().*

Now, suppose we can also perform *write*(0) or *reset*() a memory location from 1 to 0. There is an existing binary consensus algorithm that uses $2n$ locations, each storing a single bit [Bow11]. Then, it is possible to solve n-consensus using $O(n \log n)$ locations by applying Lemma 9. There is a slight subtlety, since the algorithm in the proof of Lemma 9 uses two designated locations for each round, to which values in $\{0, \ldots, n-1\}$ can be written. In place of each designated location, it is possible to use a sequence of n binary locations, all initialized to 0. Instead of performing *write*(x) on the designated location, a process performs *write*(1) to the $(x+1)$'st binary location. To find one of the values that has been written to the designated location, a process *reads* the sequence of binary locations until it sees a 1.

Theorem 35. *It is possible to solve n-consensus using $O(n \log n)$ memory locations supporting only read(), either write(1) or test-and-set(), and either write(0) or reset().*

10 Conclusions and Future Work

In this paper, we classify sets of instructions based on the minimum number of instances of an object with a countably infinite domain supporting these operations that are needed to solve obstruction-free consensus. We used consensus because it is a well-studied problem that seems to capture a fundamental difficulty of multiprocessor synchronization.

One instance of a history object can be used to get a wait-free implementation of any sequentially defined object. To perform a non-trivial operation, a process *appends* it identifier, a sequence number, the name of the operations, and the values of its arguments. It can use *get-history* to obtain the sequence of all non-trivial operations that have been performed, from which it can compute the result of a trivial operation or the last nontrivial operation it appended.

Likewise, a memory location that supports *compare-and-swap* (and, hence, *read*) can be used to implement any sequentially defined object in a non-blocking (and, hence, obstruction-free) manner. The memory location stores the value of the object. To perform an operation, a process begins by reading the location. If the operation is trivial, it uses the value, v, read to decide what to return and returns it. Otherwise, it computes the value v' the object would have if its operation were to be performed when the object has value v and the result r that would be returned from the object. Then the process performs *compare-and-swap*(v, v'). If v is returned from the *compare-and-swap*, then the process returns r. If not, it tries to perform the operation again.

Consequently, it may make sense to classify objects based on the minimum number of instances of the object needed to implement a history object or an object that supports only *compare-and-swap*. Motivated by our work, researchers have used combinations of instructions to get efficient implementations of history objects [GKSW17] and queues [KW18].

As mentioned in the introduction, it is reasonable to consider objects with bounded domains, instead of just countably infinite domains. It would be interesting to study how the domain size of an object affects the number of instances of the object that are necessary for solving n-consensus or binary consensus among n processes. Another direction for future work is to explore a classification based on the expected step complexity or solo step complexity of solving n-consensus.

There are several other interesting open problems. Except for the recent results in [EGZ18], existing space lower bounds rely on a combination of covering, valency, and indistinguishability arguments. When covering processes apply *swap*(x), as opposed to *write*(x), they can observe differences between executions. Thus, to maintain indistinguishability, these processes cannot be reused. For this reason, Lemma 4 [Zhu16] cannot be directly extended to *swap*s instead of *write*s, even though Lemmas 1, 2, and 3 can be extended. We believe that getting an $\Omega(n)$ space lower bound for solving n-consensus using only *swap*(x) and *read*() would most likely require new techniques. An algorithm that uses $o(n)$ shared memory locations would be even more surprising, as the processes would have to modify the sequence of memory locations they access based on the values they receive from *swap*s. (If they don't do this, then Lemma 4 can be extended.) We are unaware of any such algorithm.

Getting an $\omega(\sqrt{n})$ space lower bound for solving n-consensus using only *test-and-set*(), *reset*() and *read*() is also interesting. With *test-and-set*(), a covering process can observe a difference between two executions, as it can with *swap*(x). However, each location can only store a single bit. This restriction could potentially help when proving a lower bound.

To prove that $\lceil \frac{n-1}{\ell} \rceil$ ℓ-buffers are necessary for solving n-consensus, we extended the technique of [Zhu16]. The $n-1$ lower bound of [Zhu16] has since been improved to n by [EGZ18]. Hence, we expect that the new simulation-based technique used there can also be extended to prove a tight space lower bound of $\lceil \frac{n}{\ell} \rceil$.

We conjecture that, for a set of instructions, $\mathcal{I}$, which contains only *read*(), *write*(x), and either *increment*() or *fetch-and-increment*(), $\mathcal{SP}(\mathcal{I}, n) \in \Theta(\log n)$. Similarly, we conjecture that $\mathcal{SP}(\mathcal{I}, n) \in \Theta(n \log n)$ for $\mathcal{I} = \{read(), write(0), write(1)\}$. Proving these conjectures is likely to require techniques that depend on the number of input values, such as in the lower bound for m-valued adopt-commit objects by Aspnes and Ellen [AE14].

We would like to understand the properties of sets of instructions that are classified by the same function. What properties enable a collection of instructions to solve n-consensus using a single memory location? Is there an interesting characterization of the sets of instructions $\mathcal{I}$ for which $\mathcal{SP}(\mathcal{I}, n)$ is constant? What combinations of sets of instructions decrease the amount of space needed to solve consensus? For example, using only *read*(), *write*(x), and either *increment*() or *decrement*(), at least two memory locations are needed to solve binary consensus. But with both *increment*() and *decrement*(), a single memory location suffices. Are there general properties governing these relationships?

References

AAC09. James Aspnes, Hagit Attiya, and Keren Censor. Max registers, counters, and monotone circuits. In *Proceedings of the 28th ACM Symposium on Principles of Distributed Computing*, (PODC), pages 36–45, 2009.

AAD+93. Yehuda Afek, Hagit Attiya, Danny Dolev, Eli Gafni, Michael Merritt, and Nir Shavit. Atomic snapshots of shared memory. *J. ACM*, 40(4):873–890, 1993.

AE14. James Aspnes and Faith Ellen. Tight bounds for adopt-commit objects. *Theory of Computing Systems*, 55(3):451–474, 2014.

AH90. James Aspnes and Maurice Herlihy. Fast randomized consensus using shared memory. *Journal of Algorithms*, 11(3):441–461, September 1990.

AW04. Hagit Attiya and Jennifer Welch. *Distributed computing: fundamentals, simulations, and advanced topics*, volume 19. John Wiley & Sons, 2004.

Bow11. Jack R. Bowman. Obstruction-free snapshot, obstruction-free consensus, and fetch-and-add modulo k. Technical Report TR2011-681, Computer Science Department, Dartmouth College, 2011. http://www.cs.dartmouth.edu/reports/TR2011-681.pdf.

BRS15. Zohir Bouzid, Michel Raynal, and Pierre Sutra. Anonymous obstruction-free (n, k)-set agreement with n-k+1 atomic read/write registers. *Distributed Computing*, pages 1–19, 2015.

Dav04. Matei David. Wait-free linearizable queue implementations. Master's thesis, University of Toronto, 2004.

DBF05. Matei David, Alex Brodsky, and Faith Ellen Fich. Restricted stack implementations. In *19th International Conference on Distributed Computing*, (DISC), page 137—151, 2005.

EGZ18. Faith Ellen, Rati Gelashvili, and Leqi Zhu. Revisionist simulations: A new approach to proving space lower bounds. In *Proceedings of the 37th ACM Symposium on Principles of Distributed Computing*, (PODC), pages 61–70, 2018.

FHS98. Faith Fich, Maurice Herlihy, and Nir Shavit. On the space complexity of randomized synchronization. *J. ACM*, 45(5):843–862, 1998.

FLMS05. Faith Ellen Fich, Victor Luchangco, Mark Moir, and Nir Shavit. Obstruction-free algorithms can be practically wait-free. In *Proceedings of* *the 19th International Conference on Distributed Computing*, (DISC), pages 78–92, 2005.

Gel15. Rati Gelashvili. On the optimal space complexity of consensus for anonymous processes. In *Proceedings of the 29th International Conference on Distributed Computing*, (DISC), pages 452–466, 2015.

GHHW13. George Giakkoupis, Maryam Helmi, Lisa Higham, and Philipp Woelfel. An $\mathcal{O}(\sqrt{n})$ space bound for obstruction-free leader election. In *Distributed Computing*, pages 46–60. Springer, 2013.

GKSW17. Rati Gelashvili, Idit Keidar, Alexander Spiegelman, and Roger Wattenhofer. Brief Announcement: Towards Reduced Instruction Sets for Synchronization. In *31st International Conference on Distributed Computing*, (DISC), pages 53:1–53:4, 2017.

GR05. Rachid Guerraoui and Eric Ruppert. What can be implemented anonymously? In *Proceedings of the 19th International Conference on Distributed Computing*, (DISC), pages 244–259, 2005.

Her91. Maurice Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(1):124–149, 1991.

HR00. Maurice Herlihy and Eric Ruppert. On the existence of booster types. In *Proceedings of the 41st Annual IEEE Symposium on Foundations of Computer Science*, (FOCS), pages 653–663, 2000.

HS12. Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming.* Elsevier, 2012.

Int12. Intel. *Transactional Synchronization in Haswell*, 2012. http://software.intel.com/en-us/blogs/2012/02/07/transactional-synchronization-in-haswell.

Jay93. Prasad Jayanti. On the robustness of herlihy's hierarchy. In *Proceedings of the 12th Annual ACM Symposium on Principles of Distributed computing*, (PODC), pages 145–157, 1993.

JTT96. Prasad Jayanti, King Tan, and Sam Toueg. Time and space lower bounds for non-blocking implementations (preliminary version). In *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing*, (PODC), pages 257–266, 1996.

KW18. Pankaj Khanchandani and Roger Wattenhofer. On the importance of synchronization primitives with low consensus numbers. In *Proceedings of the 19th International Conference on Distributed Computing and Networking*, (ICDCN), pages 18:1–18:10, 2018.

LH00. Wai-Kau Lo and Vassos Hadzilacos. All of us are smarter than any of us: Nondeterministic wait-free hierarchies are not robust. *SIAM J. Comput.*, 30(3):689–728, 2000.

Per16. Matthieu Perrin. *Spécification des objets partagés dans le systèmes répartis sans attente (Specification of shared objects in wait-free distributed systems).* PhD thesis, University of Nantes, France, 2016.

PMJ16. Matthieu Perrin, Achour Mostéfaoui, and Claude Jard. Causal consistency: beyond memory. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, (PPOPP), pages 26:1–26:12, 2016.

Ray12. Michel Raynal. *Concurrent programming: algorithms, principles, and foundations.* Springer Science & Business Media, 2012.

Rup00. Eric Ruppert. Determining consensus numbers. *SIAM J. Comput.*, 30(4):1156–1168, 2000.

Sch97. Eric Schenk. The consensus hierarchy is not robust. In *Proceedings of the 16th Annual ACM Symposium on Principles of Distributed Computing*, (PODC), page 279, 1997.

Tau06. Gadi Taubenfeld. *Synchronization algorithms and concurrent programming.* Pearson Education, 2006.

Zhu15. Leqi Zhu. Brief announcement: Tight space bounds for memoryless anonymous consensus. In *Proceedings of the 29th International Conference on Distributed Computing*, (DISC), page 665, 2015.

Zhu16. Leqi Zhu. A tight space bound for consensus. In *Proceedings of the 48th Annual ACM Symposium on Theory of Computing*, (STOC), pages 345–350, 2016.