The Complexity of Obstruction-Free Implementations

Hagit Attiya
Technion
and
Rachid Guerraoui
EPFL
and
Danny Hendler
Ben-Gurion University
and
Petr Kuznetsov
TU Berlin/Deutsche Telekom Laboratories

Obstruction-free implementations of concurrent objects are optimized for the common case where there is no *step contention*, and were recently advocated as a solution to the costs associated with synchronization without locks. In this paper, we study this claim and this goes through precisely defining the notions of obstruction-freedom and step contention. We consider several classes of obstruction-free implementations, present corresponding generic object implementations, and prove lower bounds on their complexity. Viewed collectively, our results establish that the worst-case operation time complexity of obstruction-free implementations is high, even in the absence of step contention. We also show that lock-based implementations are not subject to some of the time-complexity lower bounds we present.

Categories and Subject Descriptors: D.1.3 [**Software**]: Programming Techniques—*Concurrent programming*; F.1.2 [**Theory of Computation**]: Computation by Abstract Devices—*Modes of Computation*; F.2 [**Theory of Computation**]: Analysis of Algorithms and Problem Complexity

General Terms: Algorithms, Theory

Additional Key Words and Phrases: shared memory, solo-fast implementations, perturbable objects, step contention, memory contention, lower bounds

---

This paper combines results that appeared, in preliminary form, in [Attiya et al. 2005; Attiya et al. 2006].

H. Attiya is with the Department of Computer Science, Technion; supported by the *Israel Science Foundation* (grant number 953/06).

R. Guerraoui is with the School of Computer and Communication Sciences, EPFL; part of this work was done while the author visited the Computer Science and Artificial Intelligence Laboratory, MIT.

D. Hendler is with the Department of Computer Science, Ben-Gurion university; part of this work was done while the author was a post-doctoral fellow in the Faculty of Industrial Engineering and Management, Technion; supported by the *Israel Science Foundation (grant number 1344/06).*

P. Kuznetsov is at the Technische Universität Berlin/Deutsche Telekom Laboratories, part of this work was done while the author was at the School of Computer and Communication Sciences, EPFL, and Max Planck Institute for Software Systems.

1. INTRODUCTION

Major chip manufacturers are changing their focus from improving the speed of individual processors to increasing parallel processing capabilities. With multi-core and multi-processor systems becoming commonplace, most computer programs will be concurrent, in some form or another. A major challenge for the theory of distributed computing is to devise efficient algorithms to cope with concurrent accesses to shared data objects. The operations of these objects would typically be implemented in software, out of more elementary *primitives* exported by *base objects*.

Traditional *locking*-based techniques are known to scale poorly and may induce deadlock and fault-tolerance issues. In contrast, implementations of shared objects that do not use locks require processes to coordinate without relying on mutual exclusion, thus avoiding the inherent problems of locking. The safety property typically required from these implementations is *linearizability* [Herlihy 1991; Herlihy and Wing 1990]; roughly, every operation on the object should appear to take effect instantaneously between its invocation and response.

Algorithms that avoid the use of locks are often, however, inefficient. Not surprisingly, the source of the inefficiency stems from handling contended situations. In fact, it is often argued [Herlihy et al. 2003; Herlihy et al. 2003] that, although contention should indeed be expected and handled appropriately, it is rare or can be made so through the use of appropriate mechanisms for managing contention. It is thus tempting to figure out the extent to which shared object implementations can exploit this rarity. In short, this would consist in devising shared object implementations that plan for the worst (high contention) and hope for the best (no contention). A natural way to explore this approach is to look for implementations that always preserve safety (linearizability) but have their termination depend on the absence of contention, i.e., ensuring (or expediting) termination when there is no contention. This paper studies the inherent complexity of such implementations.

In these implementations, fast termination is expected in the absence of step contention. When there is step contention, termination might not even be ensured. An operation may return control to a higher-level entity, called a *client*, which might decide if and when to retry the operation and may elect to use more sophisticated base objects. The term *obstruction-free* [Herlihy et al. 2003] has been used to coin these implementations. Informally, an implementation is called *obstruction-free*, "if it guarantees progress for every thread that *eventually executes in isolation*. Even though other threads may be in the midst of executing operations, ..." [Herlihy et al. 2003, Page 522].

Nothing was known on the inherent complexity of such implementations and, in fact, we are not aware of any attempt to precisely define them.

The first contribution of this paper is to precisely characterize obstruction-free implementations and this, itself, goes first through formally capturing the very notion of *contention*. We provide the first formal definition of the concept of *step contention*, according to which processes should only be considered contending on the implemented shared object if they concurrently apply primitive operations on its base objects. The absence of step contention allows scenarios where other processes have pending operations on the same implemented object but are not accessing the

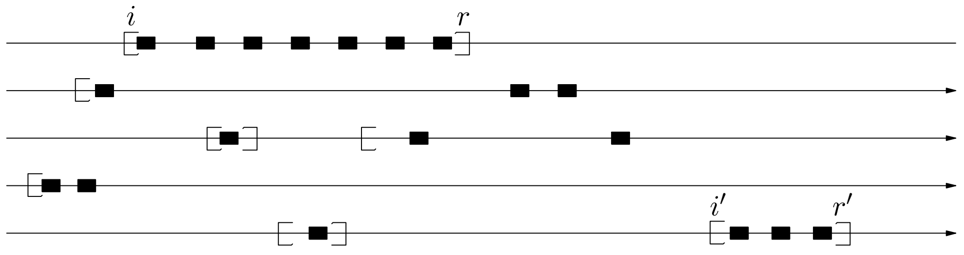

Fig. 1. Operation $[i, r]$ has interval contention 5, point contention 4, and step contention 3; operation $[i', r']$ has interval and point contention 4, and step contention 1 ($[i', r']$ is step-contention free). Square brackets denote invocations and responses, while solid intervals denote steps on base objects.

base objects of the implementation. This is fundamentally different from alternative contention metrics: *point* contention [Attiya and Fouren 2003] and *interval* contention [Afek et al. 2002], both counting also failed or swapped-out processes. (See the scenario presented in Figure 1.)

Unlike interval contention or point contention, a process encounters step contention on an implemented object only if other processes concurrently apply primitive operations on its base objects. Clearly, implementations that would guarantee termination only in the absence of interval (or point) contention might not be robust: failed (or swapped-out) processes can be used as an excuse for other processes to never terminate. An implementation might not tolerate a single process failure while being obstruction-free with respect to interval contention for instance. In short, obstruction-freedom with respect to point or interval contention do not preclude the use of locking. Our goal is to study the inherent complexity of implementations that are robust (in particular those who guarantee global progress and thus preclude locking), yet that exploit the rarity of step contention.

If a client application is not willing to wait indefinitely for operation completion, it may be beneficial for an obstruction-free implementation to return a *fail* indication to the client in the face of step contention. By doing that, the implementation enables the client to choose whether to re-invoke the same operation or to invoke another one. We show however, by reduction to the (wait-free) consensus impossibility [**?**; Loui and Abu-Amara 1987], that there is an inherent uncertainty as to whether the operation could have had an effect on the object or not (Theorem 1). When the implementation uses primitives that cannot solve consensus, e.g., reads and writes, this implies that the implementation cannot be guaranteed to return either a legal response or a *fail* indication.[1]

Several classes of obstruction-free implementations can be considered, and these differ based on how step contention is handled once detected and what base objects are available. We precisely define each class, we present corresponding generic object implementations and we give lower bounds on their complexity.

[1]The interface between the implementation and the client can be enriched so that an implementation sometimes return a special *pause* value, indicating that the client should re-invoke the same operation (see Appendix A). The notion of linearizability can be extended accordingly so as to accommodate failed operations and re-invocations of paused operations.

To measure the performance of obstruction-free implementations, we study their *step contention-free complexity*: this is the number of steps taken by a process running alone, until it returns a value. By focusing on the cost of the implementations in executions where no process encounters step contention, we capture here the complexity of obstruction-free implementations in situations where obstruction-freedom is expected to be especially beneficial.

We first study obstruction-free implementations that are restricted to use only *overwriting* or *trivial* primitives (e.g., *write*, *swap* and *read*). We present a generic obstruction-free object implementation which has a linear step contention-free complexity and uses a linear number of read/write objects (Theorem 10). We show that these complexities are asymptotically optimal, by noticing that they are subject to the lower bound of Jayanti et al. [2000] (Observation 1). This implies that obstruction-free implementations of *perturbable* objects, e.g., fetch&add, modulo-b counter (for $b \geq 2n$), b-valued compare&swap (for $b \geq n$), and LL/SC bits, using overwriting or trivial primitives must have $\Omega(n)$ step contention-free complexity and must use $\Omega(n)$ base objects.

We then study a class of obstruction-free implementations that are restricted to use simple primitives when there is no step contention, but might use more powerful primitives when there is. More specifically, these are wait-free linearizable object implementations that use only (cheap) overwriting or trivial primitives when there is no step contention, but may fall back on more powerful (and expensive) primitives like compare&swap, when step contention occurs. Such implementations are also called *solo-fast* in the sense that termination is faster when there is no step contention. We give a solo-fast implementation of consensus, which we believe is interesting in its own right, and derive a generic solo-fast implementation of any object type (Theorem 2). This generic implementation has a linear step contention-free complexity. We prove that such implementations must indeed have non-constant step and space complexity, i.e., they are also inherently expensive. Specifically, an n-process solo-fast implementation of a perturbable object, using only overwriting or trivial primitives in step-contention free executions, must have $\Omega(n)$ space complexity (Theorem 4) and $\Omega(\log n)$ step contention-free complexity (Theorem 5).

We finally consider obstruction-free implementations that use powerful primitives even when there is no step contention. We prove that, for any obstruction-free implementation of binary consensus, there exist executions where *no process is aware of step contention*, in which either some process accesses $\sqrt{n}$ distinct base objects while performing a single operation, or some process incurs $\sqrt{n}$ memory *stalls* [Dwork et al. 1997] (Theorem 6). The stalls metric captures the fact that when process p applies a nontrivial primitive to base object r right after k distinct processes other than p apply nontrivial primitives to r, then p suffers a delay of length proportional to k. This result holds also for implementations of any perturbable object (Theorem 7). Since these results are obtained in executions where none of the processes is aware of step contention, they hold for any obstruction-free implementation of these objects, regardless of how processes behave if they become aware of step contention. We also prove that, in any obstruction-free implementation of a perturbable object in which processes are not allowed to fail their operations, the number of memory stalls incurred by some process that is unaware of step contention is $\Omega(n)$ (Theorem 9). We show that lock-based implementations of perturbable objects are *not* subject to these bounds, thus establishing a separation between the worst-case operation time-complexity of obstruction-free and lock-based implementations.

To summarize, the contributions of this paper are the following. We define the notion of step contention and use it to define obstruction-free implementations (Section 2), and show that a obstruction-free consensus implementation from registers cannot return just legal values or *fail* (Section 3). We then study the complexity of different classes of obstruction-free implementations that use strong synchronization primitives only in the presence of step contention (Section 4) as well those that can use these primitives even in the absence of step contention (Section 5).

2. MODEL

We use a standard model of an asynchronous shared memory system, in which a set $\mathbf{P}$ of $n > 1$ *processes* $p_1, \ldots, p_n$ communicate through shared objects.

2.1 Objects and Implementations

Every object has a *type* that is defined by a quadruple (Q, O, R, Δ), where Q is a set of *states*, O is a set of *invocations*, R is a set of *responses*, and $\Delta \subseteq Q \times O \times Q \times R$ is a relation, known as the *sequential specification* of the type. This relation specifies all the sequences of invocation-response pairs allowed by the type.

For example, the *compare&swap* object is accessed by a *C&S*$(r_1,\, r_2,\, m)$ operation; the operation compares the value in memory location m with the content of local variable r_1, and if equal, writes the value of r_2 to m. The operation returns *true* if the compared values are equal (and the swap succeeds); otherwise, it returns *false*. The sequential specification of the *compare&swap* type includes all sequences of *C&S* operations that obey this rule. We assume that a *compare&swap* object also supports a *read* operation.

Another important example is the *consensus* object. Processes invoke a *propose* operation on the consensus object with an argument from some domain V. Processes need to agree on a single argument value. More formally, the sequential specification of consensus consists of all sequences of *propose* operations such that (1) all operations return the same value, and (2) the returned value is the argument of some *propose* operation.

To implement a (high-level) object from a set $\mathbf{B}$ of shared *base* objects, processes follow an *algorithm*, which is a collection of deterministic state machines, one for each process. The algorithm also assigns initial values to the base objects. To avoid confusion, we call operations on the base objects *primitives*, and reserve the term *operations* for the objects being implemented. We also say that an operation of an implemented object is *performed* and that a primitive is *applied to* a base object. No bound is assumed on the size of a base object (i.e., the number of distinct values the object can take).

We say that a primitive is *nontrivial* if it may change the value of the base object to which it is applied, e.g., a *write* or a *read-modify-write*, and *trivial* otherwise., e.g., a *read*. Let o be an object that supports two primitives f and f'. Following Fich et al. [1998], we say that f *overwrites* f' on o, if starting from any value v of o, applying f' and then f results in the same value as applying just f, using the same input parameters (if any) in both cases. A set of primitives is called *historyless* if all the nontrivial primitives in the set overwrite each other; we also require that each such primitive overwrite itself. A set that includes the write and swap primitives is an example of a historyless set of primitives.

In an *operation instance* $\Phi = (x, Op, p_i, args)$, process p_i performs the operation Op, with arguments $args$, on object x.

2.2 Configurations, Executions and Histories

A *configuration* specifies the value of each base object and the state of each process. In an *initial configuration*, all base objects have their initial values and all processes are in their initial states. A configuration is *quiescent* if no process is in the middle of performing an operation instance.

When receiving an *invocation* (to the high-level object), process p_i takes *steps* according to its state machine. Each step of p_i consists of some local computation and one shared memory *event*, which is a primitive applied to a base object. After each step, p_i possibly changes its local state according to its state machine and possibly returns a response on the pending high-level operation. An event is *nontrivial* if it is an application of a nontrivial primitive.

An *execution fragment* is a (possibly infinite) sequence of events, that result from interleaving the steps taken by processes, according to their state machines. An *execution* is an execution fragment that starts from an initial configuration. For any finite execution fragment α and any execution fragment α', the execution fragment $\alpha\alpha'$ is the concatenation of α and α'; in this case α' is called an *extension* of α. An execution α is *Q-free*, for a non-empty set of processes $Q \subset \mathbf{P}$, if no events in α is applied by a process in Q. If $Q = \{q\}$, we say that α is *q-free* instead of Q-free.

If the last event of an operation instance Φ has been applied in an execution α, we say that Φ *completes in* α and that Φ *returns a response in* α.

In an infinite execution, a process is *correct* if it takes an infinite number of steps or it has no pending operation; otherwise, it is *faulty*.

We denote by $\alpha|p$ the subsequence of events of execution α that are applied by process p. Two executions are *indistinguishable* to a process p, if p applies exactly the same sequence of events and gets the same responses from these events in both executions.

Every execution α induces a *history* $H(\alpha)$ that includes only the invocations and responses of the high-level operations. Each invocation or response is associated with a single process and a single object. A *local history* of process p_j in H, $H|j$, is the subsequence of H containing only events of p_j. Similarly, $H|x$ is the subsequence of H of operations on an object x.

A response *matches* an invocation if they are associated with the same process and the same object. A local history is *well-formed* if it is a sequence of matching invocation-response pairs, except perhaps for the last invocation in a finite local history. A history H is *well-formed* if every local history in H is well-formed.

A pair of consecutive invocation and matching response $[i, r]$ is called a *complete operation*, and we say that *i returns r*. An invocation i without a matching response (in a given history H) is a *pending (in H) operation*; a *completion* of a pending operation (i.e., an invocation) is the invocation followed by a matching response.

The fragment of H (or α, its corresponding execution) between the invocation i and its matching response r (if it exists) is the operation's *interval*.

2.3 Properties of Implementations

A history H is *sequential* if every invocation is immediately followed by its matching response. A sequential history H is *legal* if for every object x, $H|x$ is in the sequential specification of x.

Two different invocations i and i' on the same object x are *concurrent* in a history H, if i and i' are both pending in some finite prefix of H. This implies that their intervals overlap. We say that two operations $[i, r]$ and $[i', r']$ (or i' if i' is pending) are *non-concurrent* if their intervals are non-overlapping: Either r appears before i' in H, in which case we say that $[i, r]$ *precedes* $[i', r']$, or r' appears before i in H, in which case we say that $[i, r]$ *follows* $[i', r']$.

Definition 1. *A well-formed history H satisfies* extended linearizability *[Herlihy 1991] (see also [Attiya and Welch 2004, Chapter 10]) if there is a permutation H' containing all the complete operations and completions of a subset of the pending operations in H, such that (1) H' is legal, and (2) H' respects the order of non-concurrent operations in H. H' is the* linearization *of H.*

In accordance with the literature, we simply call this property *linearizability* below.

A process p is *active* after execution α if p is in the middle of performing some operation instance Φ, i.e., p has applied at least one event while performing Φ in α, but Φ does not complete in α. If p is not active after α, we say that p is *idle* after α.

2.4 Step contention and obstruction-freedom

The *step contention* of an execution fragment α is the number of processes that take steps in α. We say that α is *step-contention free for p* if the events of $\alpha|p$ are contiguous in α. Also, α is *step-contention free* if α is step-contention free for all processes. An operation Φ is *eventually step-contention free in α*, if either Φ completes in α or if there is a suffix of α that consists only of the events of Φ; in other words, starting from some point in α, Φ runs solo.

An implementation is called obstruction-free *"if it guarantees progress for every thread that eventually executes in isolation. Even though other threads may be in the midst of executing operations, . . . "* [Herlihy et al. 2003, Page 522]. This requirement is actually equivalent to *solo termination* [Fich et al. 1998], and it echoes the liveness correctness conditions stated for Paxos-style consensus algorithms in the context of state-machine replication [Lamport 1998]. Using our terminology, this leads to the following definition.

Definition 2. *An implementation is* obstruction-free *if every operation that is eventually step-contention free eventually returns.*

3. OBSTRUCTION-FREE IMPLEMENTATIONS USING ONLY READS AND WRITES

Obstruction-freedom is a very weak liveness condition, and it requires the operation to return only under very restricted conditions. In all other circumstances, we only

Shared variables: *register* X, initially $\perp$, and *wait-free consensus object with fails* C

```
Code for process $p_0$:

1: upon $propose(v_0)$ do
2:     $d_0 \leftarrow C.propose(v_0)$
3:     if $d_0 = \emptyset$ then
4:         $d_0 \leftarrow X$
5:     return $d_0$

Code for process $p_1$:

6: upon $propose(v_1)$ do
7:     $X \leftarrow v_1$
8:     repeat
9:         $d_1 \leftarrow C.propose(v_1)$
10:    until $d_1 \neq \emptyset$
11:    return $d_1$
```

Fig. 2. Wait-free consensus from wait-free consensus *with fails*

require that an operation's response is legal (i.e., that it does not violate the safety properties of the implementation), *if it returns a response at all.*

We extend the interface of the implementation with a special *fail* value $\emptyset$. This value can only be returned when an operation cannot return a legal response and the implementation is certain that the failed operation is not going to influence the responses of concurrent or subsequent operations. When the *client* of the implementation (i.e., the higher-level entity that invokes operations on it and receives its responses) obtains a *fail* response, it is allowed to abort its current operation and invoke any other operation instead. This enables the client to avoid waiting indefinitely for the operation to complete in high step contention scenarios.

For this purpose, we add to the set R of object responses a special *fail* value $\emptyset \notin R$. The definition of linearizability is also extended so that invocations returning *fail* are removed from the linearized history. To rule out trivial solutions, we require implementations to be *valid*: $\emptyset$ can only be returned if the corresponding operation is not step-contention free.

Note that allowing an implementation to return *fail* responses in case of step contention weakens the requirements from the implementation, and can potentially allow us to devise implementations that are impossible to devise otherwise. For instance, we could presumably implement a wait-free two-process consensus object from registers that is allowed to return $\emptyset$ in case of step contention to indicate that an operation did not take effect. We show, however, that this is impossible.

THEOREM 1. *There is no wait-free valid implementation of consensus with fails from registers.*

PROOF. Suppose by way of contradiction that such an implementation exists. We show that it is then possible to implement wait-free consensus for two processes, p_0 and p_1, using one such consensus object, denoted C, and one register X, that can be written by p_1 and read by p_0. But this contradicts [Loui and Abu-Amara 1987].

The algorithm, described in Figure 2, is asymmetric: Process p_0 invokes a *propose* operation on C. If a non-$\emptyset$ value v is returned, p_0 decides on v; otherwise, p_0 decides on the value it reads from the shared register X. Process p_1 starts by writing its value to X; then, p_1 repeatedly proposes in C, until it obtains a non-$\emptyset$ value v, and decides on it.

To see why a value returned has been proposed, note that the algorithm returns either a non-$\emptyset$ value obtained from the embedded consensus object C, or a value read from X. In the first case, from the validity property of C, the value has been proposed. Otherwise, p_0 obtains a $\emptyset$ value from C. This can only occur if p_0 encounters step contention because of the steps of p_1. In this case, it is guaranteed that p_1 has previously written its input to X, which is the value p_0 decides on.

Clearly, the processes do not return different values when they both decide on a value returned from C; the only other case is when p_0 obtains $\emptyset$ from C, in which case, p_0 decides on p_1's input (which it reads from X); in this case, since p_0's propose on C fails, C can return only p_1's input, implying that p_1 can only decide on its own value.

Finally, to show that the algorithm is wait-free, note that p_0 performs a constant number of steps, and that p_1 eventually runs in the absence of step contention, and decides. $\square$

Theorem 1 implies that any implementation of consensus from registers has executions in which it does not terminate, even if it is allowed to return $\emptyset$ when step contention is encountered. On the other hand, it is straightforward to implement an *obstruction-free* consensus object from registers (see Appendix B). By replacing wait-free consensus objects in the universal construction of Herlihy [1991] with obstruction-free ones, we obtain a generic obstruction-free implementation of any object from registers.

The generic implementation uses $O(n)$ registers and ensures that every step contention-free operation terminates in $O(n)$ steps. In fact, linear time and space complexity is asymptotically optimal for implementations of a large class of *perturbable* objects. This class, originally defined by Jayanti et al. [2000], includes counters, fetch&add and compare&swap objects; see Definition 3 in Section 4.2 below. Jayanti et al. [2000] show that any implementation of a perturbable object that satisfies the solo termination property has an execution in which a solo operation (i.e., an operation that does not observe step contention) takes $n-1$ or more steps and accesses $n-1$ or more different base objects. Since any obstruction-free implementation ensures the solo termination property, we immediately have the next observation.

Observation 1. *Let A be an obstruction-free implementation of a pertubable object using only overwriting or trivial primitives. A has an execution in which a step-contention free operation takes $n-1$ or more steps and accesses $n-1$ or more different objects.*

In Appendix A, we further extend the interface between a client and linearizable implementations with a special *pause* value. Pauses can only be returned if (1) there is step contention, and (2) the invoked operation *might* have taken effect (in a strict sense). We show that every object has an implementation from registers that provides the extended interface and always returns.

4. OBSTRUCTION-FREE IMPLEMENTATIONS THAT ARE SOLO-FAST

We call an implementation *solo-fast* if it only applies primitives from some historyless set of primitives in step-contention free executions but can apply additional primitives upon encountering step contention. In this section, we present a generic (wait-free) solo-fast implementation from registers and C&S objects, and prove complexity lower bounds for solo-fast implementations.

4.1 Generic Object Implementation

Figure 3 presents a solo-fast consensus implementation. The algorithm proceeds in rounds (lines 13–24). Starting the algorithm, every process p_i first computes in line 3 the smallest round k_i in which a value can be *fixed*, i.e., returned in line 19 (we say that p_i *joins* in round k_i). More precisely, p_i joins in minimal round k such that the registers $A_1, \ldots, A_n$ contain no values with timestamps higher than k and no *different* values with timestamp k. In every round, starting from round k_i, p_i tries to fix its current decision estimate v_i. The algorithm ensures that if no other process tries to fix concurrently any value in a higher round, or a different value in the current round, then the estimate is fixed. If p_i is not able to fix the estimate in round k (we say that p_i *loses* round k), which can only happen when there is step contention, it updates the estimate using a C&S operation and goes to round $k+1$. If no process joins in round $k+1$, then p_i fixes its estimate in round $k+1$ (the use of C&S ensures that no two processes that lost round k try to fix different values in round $k+1$). We show that no process can join in round n or later, and thus p_i fixes its estimate in round $k \leq n$. The algorithm is solo-fast, since a process does not lose a round (and thus fall back to using a C&S object) in the absence of step contention.

THEOREM 2. *There is a solo-fast consensus implementation from registers and C&S objects, which takes $O(n)$ steps in the absence of step contention.*

PROOF. The algorithm clearly implies that only a proposed value can be returned.

We say that a process p_i *reaches round* k if it reaches line 13 with $k_i = k$; p_i *participates* in an execution if it reaches some round $k \geq 1$. A process p_i *joins in round* k if k is the first round p_i reaches. We say that the result V of a collect of $A_1, \ldots, A_n$ or $B_1, \ldots, B_n$ is *k-lost* if V contains values with timestamps higher than k or two different values with timestamp k.

CLAIM 1. *For all $k \geq 2$, if no process joins in round k or later, then $A_1, \ldots, A_n$ contain no (k', v') such that $k' > k$ and no two (k, v') and (k, v'') such that $v' \neq v''$.*

PROOF. If no process joins in round k or later ($k \geq 2$), then every process that reaches round k previously completed round $k-1$ and invoked $C_{k-1}.C\&S(\perp, v_i)$ to update its current estimate (line 23). If $C_{k-1}.C\&S(\perp, v_i)$ fails, then the process adopts the value of C_{k-1} as its current estimate. Otherwise, the process keeps its estimate unchanged. Thus, all processes that reach round k have identical estimates. By the algorithm, a process can lose round k only if it reads (k', v') such that $k' > k$. Since no process joins in round k or later, this value can be written only by a process that previously lost round k. Thus, no process can lose round k and write a value (k', v') such that $k' > k$ in $A_1, \ldots, A_n$. □

CLAIM 2. *Let $k \geq 2$ be the number of processes that participate in the algorithm. No process joins in round k or later.*

PROOF. By induction on k. If there is only one participant, then it trivially joins and returns in round 1.

```
Shared variables:
    Registers $\{A_j\}, \{B_j\}, j \in \{1, 2, \ldots, n\}$, initially $\perp$
    C&S $C_1, \ldots C_{n-1}$, initially $\perp$

1:  upon propose(input_i) do
2:    $V \leftarrow$ collect $A_1, \ldots, A_n$                 // $\perp$'s are ignored in each collect
3:    $k_i \leftarrow \min\{k \geq 1 \mid \forall (k, v), (k', v') \in V : k' < k \vee (k' = k \wedge v = v')\}$
                                                               // Select a round to join
4:    if $\exists (k_i, v) \in V$ then
5:      $v_i \leftarrow v$
6:    else
7:      $V' \leftarrow$ collect $B_1, \ldots, B_n$
8:      if $V' \neq \emptyset$ then
9:        $v_i \leftarrow$ the highest timestamped value in $V'$
10:     else
11:       $v_i \leftarrow input_i$
12:   while (true) do
13:     $A_i \leftarrow (k_i, v_i)$
14:     $V \leftarrow$ collect $A_1, \ldots, A_n$
15:     if $\forall (k', v') \in V : k' < k_i \vee (k' = k_i \wedge v' = v_i)$ then
16:       $B_i \leftarrow (k_i, v_i)$
17:       $V \leftarrow$ collect $A_1, \ldots, A_n$
18:       if $\forall (k', v') \in V : k' < k_i \vee (k' = k_i \wedge v' = v_i)$ then
19:         return $v_i$
20:     $V' \leftarrow$ collect $B_1, \ldots, B_n$
21:     if $V' \neq \emptyset$ then
22:       $v_i \leftarrow$ the highest timestamped value in $V'$
23:     if $\neg C_{k_i}.C\&S(\perp, v_i)$ then $v_i \leftarrow C_{k_i}$
24:     $k_i \leftarrow k_i + 1$
```

Fig. 3. An n-process solo-fast consensus: code for process p_i

Assume now that the claim holds for all k', $2 \leq k' \leq k$, and consider any execution with $k + 1$ participants. Let p_i be the *last* process that joins in that execution. Consider a prefix α of the execution in which p_i just completed executing line 3. Let V be the result of the collect taken by p_i in line 2. Since p_i does not write in α, the other (at most k) participants cannot distinguish α from an execution with at most k participants. By the induction hypothesis, no process joins in round k or later in α. By Claim 1, V contains no (k', v') such that $k' > k$ and no two pairs (k, v') and (k, v'') such that $v' \neq v''$. By the algorithm (line 3), p_i joins in round k or earlier and the claim follows. □

Since there are at most n participants, Claim 2 implies that no process joins in round n or later. By Claim 1, in any execution, $A_1, \ldots, A_n$ contain no (k', v') such that $k' > n$ and no two (n, v') and (n, v'') such that $v' \neq v''$. It follows that any process that reaches round n will pass the "if" clauses in lines 15 and 18 and return in line 19. Thus, every process decides in round n or earlier.

Claim 3. *Assume process p_i returns v in round k. $A_1, \ldots, A_n$ contain no (k', v') such that $v' \neq v$ and $k' \geq k + 1$ and $B_1, \ldots, B_n$ contain no (k', v') such that $v' \neq v$*

and $k' \geq k$.

Proof. Assume p_i returns v in round k. By the algorithm, it has previously written (v, k) in A_i and B_i. First, we observe that, for all k, if a process p_j writes (k', v') in A_j and then (k', v'') in B_j, then $v' = v''$.

We proceed by induction on m, the number of processes that reach round $k+1$ or higher. Let $m = 0$, i.e., no process reaches any round $k' \geq k+1$. Trivially, no value with timestamp $k' \geq k+1$ can be written in $A_1, \ldots, A_n$ and $B_1, \ldots, B_n$. To obtain a contradiction, assume that some process p_j writes (k, v') with $v' \neq v$ in B_j. Thus, p_j previously wrote (k, v') in A_j, and then p_j did not read (k, v) in A_i (otherwise, p_j would not pass the "if" clause in line 15). Hence, p_i has written (k, v) in A_i *after* p_j has written (k', v') in A_j. But then p_i would necessarily read (k, v') (or a value with a higher timestamp) in A_j and would not pass the"if" clause in line 15—a contradiction.

Now assume that, for some $m \geq 0$, the claim holds for every execution in which at most m processes reach round $k+1$ or higher. Consider an execution in which $m+1$ processes reach round $k+1$ or higher. Let p_j be any process to reach round $k+1$ or higher in that execution. By the algorithm p_j collects $A_1, \ldots, A_n$ and joins in the minimal round k' such that the result of the collect operation is not k'-lost, i.e., p_j found no values with timestaps higher than k' and no different values with timestamps k'. When process p_j joins, at most m processes reached round $k+1$ or higher. We consider the two possible cases:

(1) p_j joins in round $k+1$ or higher.
By the algorithm, before joining, p_j collects $A_1, \ldots, A_n$ in line 2 and then collects $B_1, \ldots, B_n$ in line 7. Let V and V' be the results of these two collects. Since p_j joins in round $k+1$ or higher, V is k-lost. But since p_i returns v in round k, it previously wrote (k, v) in B_i and then collected $A_1, \ldots, A_n$, so that the result V'' of the last collect is not k-lost. Thus, p_j read B_i in its collect of $B_1, \ldots, B_n$ *after* p_i wrote (k, v) in it. Otherwise, since the timestamps of values written in the same register grow monotoncally, V'' would also be k-lost, and p_i would not be able to return in round k. Hence, V' contains (k, v).
Further, by the induction hypothesis, V contains no value $v' \neq v$ timestamped with $k+1$ or higher and V' contains no value $v' \neq v$ timestamped with k or higher. Thus, if p_j finds a value with timestamp $k' \geq k+1$ in V, then this value is v, and p_i adopts v as its estimate (line 5). Otherwise, p_j adopts the value with the highest timestamp in V' (line 9), and, since $(k, v) \in V'$ and no value different from v has timestamp $k+1$ or higher in V', p_j adopts v.
Finally, in round k', p_j writes (k', v) in A_j.

(2) p_j participates in round k.
Thus, in round k, p_j collects $A_1, \ldots, A_n$ so that the result of the collect operation was k-lost (line 14 or line 17), and then collects $B_1, \ldots, B_n$ in line 20. Let V and V' be the results of these two collects. Again, since p_i returns v in round k, p_i wrote (k, v) in B_i *before* p_j read B_i in its collect of $B_1, \ldots, B_n$. Hence, V' contains (k, v). By the hypothesis, when p_j completes the second collect operation, at most m processes reached round $k+1$ or higher, and, thus, V contains no value $v' \neq v$ timestamped with $k' \geq k+1$, and V' contains no value $v' \neq v$ timestamped with $k' \geq k$. By the algorithm, p_j adopts the highest

timestamped value in V' (if any) in line 22. Thus, any process that accesses the C&S object C_k previously adopted v as its estimate. If p_j fails when invoking C_k, *C&S*$(\perp, v)$, then it reads v from C_k. In both cases, when p_j begins round $k+1$, $v_j = v$.

Finally, in round $k+1$, p_j writes $(k+1, v)$ in A_j.

In both cases, when p_j reaches its first round higher than k, it cannot write a value different from v , implying the inductive claim. □

Let k be the first round in which some process p_i decides some value v. By the algorithm, if a process p_j decides v' in a round k', then it previously wrote (k', v') in B_j (line 16). By Claim 3, no process p_j writes (k', v') such that $v' \neq v$ and $k' \geq k$. Thus, no process decides $v' \neq v$, implying that processes do not decide on different values.

Finally, we need to show that the implementation is solo-fast. Assume that a process p_i joins in round k. The only reason for p_i to lose that round is to observe a value timestamped with $k' > k$ or two different values timestamped with k in $A_1, \ldots, A_n$ (line 15 or 18). But p_i previously observed the opposite in line 3 and adopted the value timestamped with k found in $A_1, \ldots, A_n$ (if any) in line 5. Thus, p_i can lose round k only when some other process p_j concurrently writes (k', v') in A_j such that $k' > k \vee (k' = k \wedge v' \neq v)$, i.e, when there is step contention. Clearly, if p_i does not lose round k, then it takes a linear number of read and write operations.

Theorem 2 and the fact that the universal construction of Herlihy [1991] uses only reads and writes, in addition to consensus objects, immediately imply:

Corollary 3. *Every sequential type has a solo-fast implementation from registers and C&S objects.*

4.2 Time and Space Lower Bounds

In this section, we prove time and space lower bounds for solo-fast implementations, showing that the non-constant step contention-free complexity and space complexity of our implementations are unavoidable.

One may be tempted to think that the linear lower bounds on the space and step complexity of solo-fast implementations follow from the results of Jayanti et al. [2000]. This is not true due to the following reason: their results are obtained by constructing executions in which processes continue executing their operations, using reads and writes only, even after they encounter step contention. Contrary to that, solo-fast implementations are allowed to apply stronger synchronization primitives in the face of step contention.

Our lower bounds hold for implementations of perturbable objects, a wide class of objects defined next; the following definition is equivalent to Definition 3.1 of Jayanti et al. [2000], when restricted to consider only deterministic implementations.

Definition 3. *An object $\mathcal{O}$ is* perturbable *if there is an operation instance op_n by process p_n, such that for any p_n-free execution $\alpha\lambda$ where no process applies more than a single event in λ and for some process $p_l \neq p_n$ that applies no event in λ, there is an extension of α, γ, consisting of events by p_l, such that p_n returns* *different responses when performing op_n by itself after $\alpha\lambda$ and after $\alpha\gamma\lambda$. We say that op_n* witnesses the perturbation *of $\mathcal{O}$.*

The following technical definition is required for our proofs. Note that if a process is active in the configuration resulting from a finite execution α, then the process has exactly one *enabled* event, i.e., the event the process is about to apply in the configuration. If a process is idle but has begun a new operation-instance, then the first event of that operation-instance is enabled; otherwise, it has no enabled event.

Definition 4. *A base object o is* covered after *an execution α if the set F of all the primitives applied to o in α is historyless, and there is a process p_n that has, after α, an enabled event e about to apply a nontrivial primitive from F to o. We also say that e* covers *o* after *α.*
An execution α is k-covering *if:*

—*α is step-contention free,*

—*there exists a set of processes $\{p_{j_1}, \ldots, p_{j_k}\}$ that does not contain process p_n, such that all the events of α are applied by processes in this set and each of the processes in the set has an enabled nontrivial event that covers a distinct base object after α.*

We call the set $\{p_{j_1}, \ldots, p_{j_k}\}$ a covering set *of α.*

The second condition in Definition 4 implies that if an implementation has a k-covering execution, then its space complexity is at least k. We now prove a linear lower bound on the space complexity of any obstruction-free solo-fast implementation.

Theorem 4. *Let A be an n-process solo-fast obstruction-free implementation of a perturbable object $\mathcal{O}$. The space complexity of A is at least $n-1$.*

Proof. Since A is solo-fast, there exists a historyless set of primitives S such that any process p can apply only primitives from S in executions that are step-contention free for p.

Let op_n be the operation instance that witnesses the perturbation of $\mathcal{O}$. We prove the theorem by showing that A has an $(n-1)$-covering execution.

The proof goes by induction. The empty execution is vacuously a 0-covering execution. Assume that α_i, for $i < n-1$, is an i-covering execution with covering set $\{p_{j_1}, \ldots, p_{j_i}\}$. Let λ_i be the execution fragment that consists of the nontrivial events by processes $p_{j_1} \ldots p_{j_i}$ that are enabled after α_i, arranged in some arbitrary order.

By Definition 3, there is an execution fragment γ by some process $p_{j_{i+1}} \notin \{p_n, p_{j_1}, \ldots, p_{j_i}\}$ such that op_n returns different responses after executions $\alpha_i\lambda_i$ and $\alpha_i\gamma\lambda_i$. We claim that γ contains a nontrivial event that accesses a base object not covered after α_i. Assume otherwise to obtain a contradiction. Since all events in executions $\alpha_i\lambda_i$ and $\alpha_i\gamma\lambda_i$ apply primitives from a historyless set, every nontrivial primitive applied to a base object in γ is overwritten by some event in λ_i. Thus, the values of all base objects are the same after $\alpha_i\lambda_i$ and after $\alpha_i\gamma\lambda_i$. This implies that op_n must return the same response after both $\alpha_i\lambda_i$ and $\alpha_i\gamma\lambda_i$, which is a contradiction.

We extend α_i by letting $p_{j_{i+1}}$ execute the shortest prefix of γ at the end of which it has an enabled nontrivial event about to access an object o not covered after α_i. We denote this prefix of γ by γ'. We define α_{i+1} to be $\alpha_i\gamma'$. Thus, at the end of α_{i+1}, $p_{j_{i+1}}$ has an enabled nontrivial event that accesses o. As none of the processes $p_{j_1}, \ldots p_{j_i}$ apply events in γ', we have that α_{i+1} is a step-contention free execution, after which processes $p_{j_1}, \ldots p_{j_{i+1}}$ have enabled events that cover distinct objects. Hence α_{i+1} is an $(i+1)$-covering execution. It follows that A has an $(n-1)$-covering execution. □

Next we prove a logarithmic lower bound on the step contention-free complexity of solo-fast implementations of perturbable objects. As the proof is quite involved, we first provide an informal description of its technique and structure.

Our goal is to construct a scenario in which process p_n has to access a large number of base objects as it runs solo while performing an operation. To that end, our proof constructs longer and longer r-covering executions. The construction proceeds in phases. After each phase r of the construction, we consider the path that p_n will take *if* it runs solo after we 'unfreeze' the pending covering events (but we don't actually unfreeze these events). We denote this path by π_r. Note that some of the objects along this path may already be covered after phase r.

To construct phase $r+1$, we deploy a 'free' process, $p_{j_{r+1}}$, and let it run solo. As processes can only apply primitives from a historyless set, and as the implemented object is perturbable, we know that $p_{j_{r+1}}$ will eventually be about to write to an uncovered object, O, along π_r. This, however, may have the undesirable effect (from the perspective of an adversary) of making π_{r+1} shorter than π_r: p_n may read the information written by $p_{j_{r+1}}$ to O (if we unfreeze its pending covering event) and not access some other objects farther along π_r!

Note, however, that objects that are part of π_r will be absent from π_{r+1} only if *O precedes them in π_r*. Thus the set of objects along π_{r+1} that are covered (after phase $r+1$) is 'closer', in a sense, to the beginning of the path. It follows that if there are many phases r such that $|\pi_r|$ decreases, then one of the paths π_r must be 'long'.

To capture this intuition, we define Ψ, a monotonically-increasing progress function of the phase numbers. Ψ_r is a $(\log n)$-digit binary number defined as follows. Bit 0 (the most significant bit) of Ψ_r is 1 if and only if the first object in π_r is covered; bit 1 of Ψ_r is 1 if and only if the second object in π_r exists and is covered, and so on. Note that we do not need to consider paths that are longer than $\log_2 n$. If such a path exists, the lower bound clearly holds.

As mentioned before, to construct phase $r+1$, we deploy a free process, $p_{j_{r+1}}$, and let it run solo until it is about to write to an uncovered object, O, along π_r. In terms of Ψ, this implies that the covering event of $p_{j_{r+1}}$ might flip some of the digits of Ψ_r from 1 to 0. But O corresponds to a more significant digit, and this digit is flipped from 0 to 1, hence $\Psi_{r+1} > \Psi_r$ must hold. As we have $n-1$ processes to deploy, Ψ_r must increase $n-1$ times and eventually it equals $n-1$. When it does, the length of π_r is *exactly* $\log_2 n$. The formal proof follows.

Theorem 5. *Let A be an n-process solo-fast obstruction-free implementation of a perturbable object $\mathcal{O}$. A has a step-contention free execution in which a process accesses at least* $\log_2 n$ *distinct base objects in the course of performing a single* *operation instance.*

Proof. Since A is solo-fast, there exists a historyless set of primitives S such that any process p can apply only primitives from S in executions that are step-contention free for p.

If there is an execution in which a process accesses more than $\log_2 n$ distinct base objects in the course of performing a single operation instance in a step-contention free manner then we are done. Assume otherwise. We construct a step-contention free execution in which a process accesses *exactly* $\log_2 n$ distinct base objects in the course of performing a single operation instance.

The construction proceeds in at most n phases. In phase $r \geq 0$, we construct an execution $\alpha_r\delta_r\phi_r$ with the following structure:

—α_r is an r-covering execution with a covering set $p_{j_1}, \ldots, p_{j_r}$,

—in δ_r, each of the processes $p_{j_1}, \ldots, p_{j_r}$ applies a nontrivial event to an object that is covered after α_r, and

—in ϕ_r, process p_n runs solo after $\alpha_r\delta_r$ until it completes the operation instance op_n.

Let $C(\alpha_r)$ denote the set of base objects that are covered after α_r. Let $\pi_r = O_r^1 \ldots O_r^{i_r}$ denote the sequence of all distinct base objects accessed by p_n in ϕ_r (after $\alpha_r\delta_r$) indexed according to the order in which they are first accessed by p_n. Also let S_{π_r} denote the set of these base objects.

In execution $\alpha_r\delta_r\phi_r$, p_n accesses i_r distinct base objects. Thus, it suffices for the proof to construct such an execution with $i_r = \log_2 n$. For $j \in \{1, \ldots, i_r\}$, we let b_r^j be the indicator variable whose value is 1 if $O_r^j \in C(\alpha_r)$ and 0 otherwise. We associate an integral progress parameter, Ψ_r, with each phase $r \geq 0$, defined as follows:

$$\Psi_r = \sum_{j=1}^{i_r} b_r^j \cdot 2^{\log_2 n - j} \ . \tag{1}$$

As we assume that $i_r \leq \log_2 n$ for all r, Ψ_r can be viewed as a $\log_2 n$-digit number in base 2 whose j'th most significant bit is 1 if the j'th object in π_r exists and is in $C(\alpha_r)$, or 0 otherwise. This implies that the number of base objects in π_r that are covered after α_r equals the number of 1-bits in Ψ_r.

We now describe our construction. Let α_0 and δ_0 denote the empty execution; let ϕ_0 denote the solo execution that results when p_n performs the operation instance op_n starting from an initial configuration, and let i_0 denote the number of distinct objects accessed in ϕ_0. Since $C(\alpha_0) = \emptyset$, we have $\Psi_0 = 0$. Suppose that, for some r, $0 \leq r < n-1$, we have constructed $\alpha_r\delta_r\phi_r$ and $\Psi_r < n-1$.

As $\mathcal{O}$ is perturbable with operation instance op_n witnessing that, there is an execution fragment γ_{r+1} by some process $p_{j_{r+1}} \notin \{p_n, p_{j_1}, \ldots, p_{j_r}\}$ such that op_n returns different responses to p_n after executions $\alpha_r\delta_r$ and $\alpha_r\gamma_{r+1}\delta_r$. We claim that, in γ_{r+1}, $p_{j_{r+1}}$ applies a nontrivial event to an object in $S_{\pi_r} \setminus C(\alpha_r)$. Assume that γ_{r+1} contains no nontrivial events to objects in $S_{\pi_r} \setminus C(\alpha_i)$ to obtain a contradiction. As $\alpha_r\gamma_{r+1}$ is step-contention free, all the events of γ_{r+1} either access base objects not in S_{π_r} or are overwritten by the events of δ_r. It follows that $\alpha_r\gamma_{r+1}\delta_r\phi_r$ is also an execution of A and that $\alpha_r\delta_r\phi_r$ and $\alpha_r\gamma_{r+1}\delta_r\phi_r$ are indistinguishable to p_n.

This implies that op_n must return the same responses after both executions, which is a contradiction.

Let γ'_{r+1} be the shortest prefix of γ_{r+1} after which $p_{j_{r+1}}$ has an enabled event, e, about to apply a nontrivial event to a base object $O_r^k \in S_{\pi_r} \setminus C(\alpha_r)$. Define $\alpha_{r+1} = \alpha_r\gamma'_{r+1}$, $\delta_{r+1} = \delta_r e$ and let ϕ_{r+1} denote the execution fragment in which p_n applies events by itself after $\alpha_{r+1}\delta_{r+1}$ as it performs the operation instance op_n to completion. It is easily verified that α_{r+1} is an $(r+1)$-covering execution and that $C(\alpha_{r+1}) = C(\alpha_r) \cup O_r^k$.

We claim that $\Psi_{r+1} > \Psi_r$ holds. As $O_r^k \notin C(\alpha_r)$, we have $b_r^k = 0$. As the values of objects $O_r^1 \cdots O_r^{k-1}$ are the same after $\alpha_r\delta_r$ and $\alpha_{r+1}\delta_{r+1}$, it follows that $b_r^j = b_{r+1}^j$ for $j \in \{1, \ldots, k-1\}$. This implies, in turn, that $O_r^k = O_{r+1}^k$. As $O_{r+1}^k \in C(\alpha_{r+1})$, we have $b_{r+1}^k = 1$. We get:

$$\begin{aligned}
\Psi_{r+1} &= \textstyle\sum_{j=1}^{i_{r+1}} b_{r+1}^j \cdot 2^{\log_2 n - j} \\
&= \textstyle\sum_{j=1}^{k-1} b_{r+1}^j \cdot 2^{\log_2 n - j} + 2^{\log_2 n - k} + \\
&\quad \textstyle\sum_{j=k+1}^{i_{r+1}} b_{r+1}^j \cdot 2^{\log_2 n - j} \\
&= \textstyle\sum_{j=1}^{k-1} b_r^j \cdot 2^{\log_2 n - j} + 2^{\log_2 n - k} + \\
&\quad \textstyle\sum_{j=k+1}^{i_{r+1}} b_{r+1}^j \cdot 2^{\log_2 n - j} \\
&\geq \textstyle\sum_{j=1}^{k-1} b_r^j \cdot 2^{\log_2 n - j} + 2^{\log_2 n - k} \\
&> \textstyle\sum_{j=1}^{k-1} b_r^j \cdot 2^{\log_2 n - j} + \sum_{j=k+1}^{i_r} b_r^j \cdot 2^{\log_2 n - j} \\
&= \Psi_r.
\end{aligned}$$

By definition, we have $0 \leq \Psi_r \leq n-1$ for all r. Furthermore, just a single process joins the execution in each phase. As we have shown that Ψ is monotonically increasing with r, this implies that we eventually reach a phase r^* with $\Psi_{r^*} = n-1$, i.e., $i_{r^*} = \log_2 n$. $\Box$

5. OBSTRUCTION-FREE IMPLEMENTATIONS USING ARBITRARY PRIMITIVES

In the previous sections, we considered obstruction-free implementations that can only apply synchronization primitives from a restricted set—either in all executions or just in step-contention free executions; the metric that we used counted the worst-case number of steps made by a process in step-contention free executions. However, the number of steps performed is not the only factor that contributes to the time complexity of concurrent objects. In practice, the performance of concurrent objects is influenced by the extent to which multiple processes access widely-shared memory locations simultaneously.

In 1993, Dwork et al. [1997] introduced a formal model to capture the phenomenon of memory contention in shared memory machines. Their model takes into consideration both the number of steps taken by a process and the number of stalls it incurs as a result of memory contention with other processes. In this section, we investigate obstruction-free implementations that can use *arbitrary* primitives even in step contention free executions. We analyze the worst-case step- and stall-complexities of these implementations. The definition of stalls that we use is similar to that used by Hendler and Shavit [2008] and Fich et al. [2005]. It is stricter than that used by Dwork et al. [1997] since we only count memory stalls caused by contention in writing, whereas Dwork et al. [1997] also count memory stalls due to contention in reading. This definition captures the fact that when multiple processes apply nontrivial primitive operations simultaneously to the same base object, these operations are being serialized.

Definition 5. *Let e be an event applied by a process p to base object r, as it performs an operation instance Φ in execution α. Let $\alpha = \alpha_0 e_1 \cdots e_k e \alpha_1$, where α_0 and α_1 are execution fragments and $e_1 \cdots e_k$ is a maximal sequence of $k \geq 1$ consecutive nontrivial events, by distinct processes other than p, that access r. Then we say that Φ* incurs k memory stalls in α on account of e. *The* number of memory stalls incurred by Φ in α *is the sum of memory stalls Φ incurs in α over all the events of Φ in α.*

Let p be a process and consider the set of executions $\mathcal{E}_p$ that are indistinguishable to p from an execution that is step-contention free to p. This includes all the executions that are step-contention free to p. From obstruction freedom, p must make progress in any execution of $\mathcal{E}_p$. Let $\alpha \in \mathcal{E}_p$ be an execution and let e be an event of p that is enabled after α. We say that e is *issued while p is unaware of step contention*. It might be that p becomes aware of step contention when it receives the response of e. Nevertheless, the delay incurred by p until it becomes aware of step contention includes the delay it incurs on account of e.

In a similar manner, we let $\mathcal{E}$ denote the set of executions that are indistinguishable to *all* processes from a step-contention free execution. Let $\alpha \in \mathcal{E}$ be an execution and let e be an event that is enabled after α. We say that e is *issued while no process is aware of step contention.*

In the proofs that follow we consider the worst-case time complexity incurred by processes on account of the events they issue while being unaware of step contention.

5.1 Binary Consensus and Perturbable Objects

In this section we consider obstruction-free implementations of binary consensus and perturbable objects. We prove a tradeoff between the worst case operation step-complexity and stall-complexity of these implementations. More specifically, we prove that each such implementation has an execution in which no process is aware of step contention, such that either a process accesses at least $\sqrt{n}$ distinct base objects in the course of performing a single operation or some process incurs at least $\sqrt{n}$ stalls on account of a single event. This bound holds for all obstruction-free implementation of these objects, regardless of how processes behave when they encounter step contention. It implies that, for these implementations, a process can be made to incur a delay of length $\Omega(\sqrt{n})$ before *any process* becomes aware of step contention.

A *one-shot binary consensus* object restricts the argument to the *propose* operation to be in the domain $\{0, 1\}$; each process calls *propose* at most once.

Theorem 6. *Let A be an n-process obstruction-free implementation of binary consensus. There is an execution α of A in which no process is aware of step contention and a process p such that either p accesses at least $\sqrt{n}$ distinct base objects in α or it incurs at least $\sqrt{n}$ stalls on account of an event it applies in α.*

Proof. Consider executions of A in which processes $p_1, \ldots, p_{n-1}$ invoke *propose* with input 0 and process p_n invokes *propose* with input 1. Let ϕ be the execution in which, starting from the initial configuration, p_n performs its *propose* instance to completion. Let B denote the set of base objects that are accessed in ϕ. If $|B| \geq \sqrt{n}$ then we are done.

Otherwise, we construct a p_n-free execution at the end of which there is a subset of processes $S \subset \{p_1, \cdots, p_{n-1}\}$ of size exactly $\sqrt{n}$, all the processes of which have enabled nontrivial events about to access the same object in B. The execution is constructed inductively in at most $n-1$ phases. We denote the execution constructed in phases $1, \cdots, i$ by α_i. Our construction maintains the following invariants for all $i \leq n-1$:

—α_i is step-contention free,

—all the events of α_i are applied by processes in $\{p_1, \ldots, p_i\}$,

—α_i does not contain any nontrivial event applied to an object in B, and

—each of the processes $p_1, \cdots, p_i$ covers a base object in B after α_i.

We let α_0 denote the empty execution. It is easily verified that the above invariants are vacuously met by α_0. Assume we have constructed α_i, for $i < n-1$, and that the number of enabled nontrivial events about to access any single object in B at the end of α_i is less than $\sqrt{n}$. We now describe the construction of α_{i+1}. We let process p_{i+1} perform its instance of *propose* by itself after α_i until it either has an enabled nontrivial event about to access an object in B, or its *propose* instance completes.

We show that the latter cannot occur, by way of contradiction. The sequential specification of the consensus object imply that p_n's instance of *propose* returns 1 in ϕ. Let σ_{i+1} be the execution in which p_{i+1} performs its *propose* instance after α_i until it completes. As $\alpha_i\sigma_{i+1}$ is p_n-free, the *propose* operation of p_{i+1} returns 0 in $\alpha_i\sigma_{i+1}$.

By the induction hypothesis on α_i, and as we assume that no nontrivial event was applied to an object in B in σ_{i+1}, $\alpha_i\sigma_{i+1}\phi$ is an execution that is indistinguishable from ϕ to p_n. It follows that the responses of the instances of *propose* by p_{i+1} and p_n in $\alpha_i\sigma_{i+1}\phi$ are 0 and 1, respectively. This contradicts the requirement that all processes decide on the same value.

Thus, at the end of α_{i+1}, process p_{i+1} has an enabled nontrivial event about to access a base object in B. By the induction hypothesis on α_i, we get that at the end of α_{i+1}, each of $p_1, \cdots, p_{i+1}$ has an enabled nontrivial event about to access an object in B, and that α_{i+1} is a step-contention free execution that contains no nontrivial event applied to an object in B.

As $|B| < \sqrt{n}$, there is a phase j, $j \leq n-1$, such that after α_j there exist at least $\sqrt{n}$ processes, all of which have enabled nontrivial events about to access the same object $o \in B$. Let α be some ordering of these events. Also let β be the longest prefix of ϕ that does not access o, and let e be p_n's enabled event after β. Then p_n incurs at least $\sqrt{n}$ memory stalls in $\alpha_j\beta\alpha e$. To conclude the proof, we note that $\alpha_j\beta$ is step-contention free and that each of the events in αe is by a different process. Thus all the events of $\alpha_j\beta\alpha e$ are issued while no process is aware of step contention. □

The proof of the following theorem follows the lines of that of Theorem 6.

THEOREM 7. *Let A be an n-process obstruction-free implementation of a perturbable object. There is an execution α of A in which no process is aware of step contention and a process p such that either p accesses at least $\sqrt{n}$ distinct base objects in α or it incurs at least $\sqrt{n}$ stalls on account of an event it applies in α.*

PROOF. Let op_n be the operation instance that witnesses the perturbation of $\mathcal{O}$. Let ϕ be the execution of A in which, starting from the initial configuration, p_n performs op_n until it completes it. Let B denote the set of base objects that are accessed in ϕ. If $|B| \geq \sqrt{n}$ then we are done. Assume otherwise.

We construct a p_n-free execution at the end of which there is a subset of processes $S \subset \{p_1, \cdots, p_{n-1}\}$ of size exactly $\sqrt{n}$, all the processes of which have enabled nontrivial events about to access the same object in B. The execution is constructed inductively in at most $n-1$ phases. We denote the execution constructed in phases $1, \ldots, i$ by α_i. Our construction maintains the following invariants for all $i \leq n-1$:

—α_i is step-contention free,

—α_i does not contain any nontrivial event applied to an object in B, and

—there exists a set of processes $\{p_{j_1}, \ldots, p_{j_i}\}$ that does not contain p_n, such that
  —each of these processes has an enabled nontrivial event about to access a base object in B after α, and
  —none of the events of α_i are applied by processes not in $\{p_{j_1}, \ldots, p_{j_i}\}$.

We let α_0 denote the empty execution. It is easily verified that the above invariants are vacuously met by α_0. Assume we have constructed α_i, for $i < n-1$, and that the number of enabled nontrivial events about to access any single object in B at the end of α_i is less than $\sqrt{n}$. We now describe the construction of α_{i+1}. By the induction hypothesis on α_i, no process has applied a nontrivial event in α_i to an object in B.

Let δ denote the execution fragment that consists of the events by $\{p_{j_1}, \ldots, p_{j_i}\}$ that are enabled after α_i. As $\mathcal{O}$ is perturbable with operation instance op_n witnessing that, there is an execution fragment γ by some process $p_{j_{i+1}} \notin \{p_n, p_{j_1}, \ldots, p_{j_i}\}$ such that op_n returns different responses to p_n after executions $\alpha_i\delta$ and $\alpha_i\gamma\delta$. We claim that $p_{j_{i+1}}$ applies in γ a nontrivial event to an object in B. Assume otherwise to obtain a contradiction. Then from the induction hypothesis and our assumption, $\alpha_i\gamma\delta\phi$ is an execution that is indistinguishable to p_n from $\alpha_i\delta\phi$. It follows that the responses of op_n are the same in $\alpha_i\gamma\delta\phi$ and $\alpha_i\delta\phi$. This is a contradiction.

Let γ' be the shortest prefix of γ after which $p_{j_{i+1}}$ has an enabled nontrivial event about to access a base object in B. We let α_{i+1} be $\alpha_i\gamma'$. Thus, from the induction hypothesis applied to α_i, we get that at the end of α_{i+1} each of $p_{j_1}, \cdots, p_{j_{i+1}}$ has an enabled nontrivial event about to access an object in B, and that α_{i+1} is a step-contention free execution that contains no nontrivial event applied to an object in B.

As $|B| < \sqrt{n}$, there is a phase k, $k \leq n-1$, such that after α_k there exist at least $\sqrt{n}$ processes, all of which have enabled nontrivial events about to access the same object $o \in B$.

Let α be some ordering of these events. Also let β be the longest prefix of ϕ that does not access o, and let e be p_n's enabled event after β. Then p_n incurs at least $\sqrt{n}$ memory stalls in $\alpha_k\beta\alpha e$ on account of e. To conclude the proof, we note that

$\alpha_k\beta\alpha e$ and $\alpha_k\alpha\beta e$ are indistinguishable to p_n and that $\alpha_k\alpha\beta e$ is step-contention free for p_n. Thus all the events of $\alpha_k\beta\alpha e$ are issued while no process is aware of step contention. □

Theorems 6 and 7 establish the existence of executions in which *a single operation* incurs a tradeoff between step-complexity and stall-complexity. It is easy to construct executions in which the tradeoffs of Theorems 6 and 7 hold for *multiple operations*. This can be done by simply extending the executions constructed by these theorems until a quiescent configuration is reached and then repeating the construction.

5.2 A Linear Lower Bound for Non-Failing Implementations

We say that an obstruction-free implementation is *non-failing* if processes are not allowed to fail their operations when they become aware of step contention. For such implementations we obtain a lower bound that is stronger than the one obtained in Section 5.1.

Fich et al. [2005] prove a lower bound of $n-1$ on the worst-case number of stalls incurred by a process as it performs a single operation instance. This bound holds for non-failing obstruction-free implementations of objects in a class $\mathcal{G}$, that includes counter and single-writer snapshot objects. It can be shown that the same lower bound holds for any perturbable object. In the following, we prove that this bound holds in an execution in which all the events of the process whose operation instance incurs the linear complexity are issued while it is not aware of step contention.

The following definition of *k-stall-execution* is taken from [Fich et al. 2005] with minor terminology adaptation.

Definition 6. *An execution $\alpha\sigma_1\cdots\sigma_i$ is a k-*stall execution for process *p if*

—*α is p-free,*

—*there are distinct base objects $O_1,\ldots,O_i$ and disjoint sets of processes $S_1,\ldots,S_i$ whose union does not include p and has size k such that, for $j=1,\ldots,i$,*

—*each process in S_j has an enabled nontrivial event about to access O_j after α, and*

—*in σ_j, process p applies events by itself until it is first about to apply an event to O_j, then each of the processes in S_j applies an event that accesses O_j, and, finally, p applies an event that accesses O_j,*

—*all processes not in $S_1\cup\cdots\cup S_i$ are idle after α,*

—*p starts at most one operation instance in $\sigma_1\cdots\sigma_i$, and*

—*in every $(\{p\}\cup S_1\cup\cdots\cup S_i)$-free extension of α, no process applies a nontrivial event to any base object accessed in $\sigma_1\cdots\sigma_i$.*

In a k-stall execution for process p, p incurs k stalls, since it incurs $|S_j|$ stalls when it applies its first event to O_j, for $j=1,\ldots i$. The results of Fich et al. [2005] are obtained by proving that non-failing obstruction-free implementations of objects such as those mentioned above have $n-1$ stall executions for any process. Our contribution lies in the following technical lemma. It shows that a process p is not aware of step contention in a k-stall execution for p.

LEMMA 8. *Let α be a k-stall execution for process p. Then all of p's events in α are issued while p is unaware of step contention.*

PROOF. Let $\alpha\sigma_1 \ldots \sigma_i$ be a k-stall execution for process p for some $k > 0$. For $j = 1, \ldots, i$, let S_j and O_j be as in Definition 6. For an execution σ, let $\sigma|\overline{p}$ be the subsequence of events in σ that are applied by processes other than p.

We prove that the sequence of events $\theta = \alpha(\sigma_1|\overline{p}) \cdots (\sigma_i|\overline{p})\ (\sigma_1|p) \cdots (\sigma_i|p)$ is an execution and that it is indistinguishable from $\alpha\sigma_1 \ldots \sigma_i$ to all processes. Since θ is step-contention free for p, this will establish that all of p's events in $\alpha\sigma_1 \ldots \sigma_i$ are issued while p is unaware of step contention.

The proof goes by double induction. For $l = 0, \ldots, i$, let $\alpha_l = \alpha\sigma_0 \ldots \sigma_l$. The outer induction is on the executions α_l. We prove that, for $l = 0, \ldots, i$, $\theta_l = \alpha(\sigma_1|\overline{p}) \cdots (\sigma_l|\overline{p})(\sigma_1|p) \cdots (\sigma_l|p)$ is an execution that is indistinguishable to all processes from α_l. The claim holds vacuously for $l = 0$. For $l < i$, assume that θ_l is an execution that is indistinguishable from α_l to all processes.

Consider the sequence of events $\sigma_{l+1}|\overline{p}$. By Definition 6, these events are enabled at the end of α_l. Consequently, from outer induction hypothesis, they are also enabled at the end of θ_l. As all the events of $(\sigma_1|p) \cdots (\sigma_l|p)$ are applied by p, the events of $\sigma_{l+1}|\overline{p}$ are enabled at the end of $\alpha(\sigma_1|\overline{p}) \cdots (\sigma_l|\overline{p})$. Additionally, as each of the events of $\sigma_{l+1}|\overline{p}$ is applied by a distinct process in S_{l+1}, $\alpha(\sigma_1|\overline{p}) \cdots (\sigma_{l+1}|\overline{p})$ is an execution.

By the induction hypothesis, all processes in $S_1 \cup \cdots \cup S_l$ apply the same events and get the same responses in $\alpha(\sigma_1|\overline{p}) \cdots (\sigma_l|\overline{p})$ and α_l. As all the events of $\sigma_{l+1}|\overline{p}$ access O_{l+1} and none of the events of $\sigma_1 \cdots \sigma_l$ accesses O_{l+1}, it follows that all processes in $S_1 \cup \cdots \cup S_{l+1}$ apply the same events and get the same responses in α_{l+1} and in $\alpha(\sigma_1|\overline{p}) \cdots (\sigma_{l+1}|\overline{p})$, and hence also in θ_{l+1}.

We next show that θ_{l+1} is an execution and that p gets the same responses from the events it applies in it as in α_{l+1}. We show this by inner induction on the number of events, m, applied by p in $(\sigma_1|p) \cdots (\sigma_{l+1}|p)$.

The claim is obvious for $m = 0$. Assume that $(\sigma_1|p) \cdots (\sigma_{l+1}|p)$ consists of $m > 0$ events and that the claim holds for the first $m - 1$ events. Let e be the m'th event. Two cases exist. If e accesses a base object $O \notin \{O_1, \ldots, O_{l+1}\}$, then, from Definition 6, O is not accessed in $\alpha\sigma_1 \cdots \sigma_{l+1}$ by any process other than p. Thus, from the inner induction hypothesis, O has the same value when e accesses it in both α_l and θ_l. Otherwise, suppose that $O = O_j$ for some $j \in \{1, \ldots, l+1\}$. The subsequence of events that precede e in accessing O_j is $(\sigma_j|\overline{p})$ in both α_{l+1} and θ_{l+1}. Consequently, from inner and outer induction hypotheses, O has the same value when accessed by e in both α_{l+1} and θ_{l+1}. It follows that, in both cases, e returns the same response in α_l. Hence also p applies the same events, and gets the same responses from these events, in both α_l and θ_l.

As all processes apply the same events, and get the same responses from these events in both α_l and θ_l, and as α_l is an execution, it follows that θ_l is also an execution. This concludes the proof of the lemma. □

The proof of Theorem 6 of Fich et al. [2005] can be used to establish that any non-failing obstruction-free n-process implementation of a perturbable object has an $(n-1)$-stall execution for any process that shares the implementation. Combining that with Lemma 8 gives the following.

THEOREM 9. *Let A be an n-process non-failing obstruction-free implementation of a perturbable object. Then for any process p there is an execution α of A such that p incurs in α at least $n-1$ stalls on account of events that it issues while it is unaware of step contention, as it performs a single operation instance.*

Theorem 9 constructs an execution in which a single operation incurs a linear number of stalls. An executions in which this complexity is incurred by *multiple operations* can be constructed by simply extending the execution constructed by the theorem until a quiescent configuration is reached and then repeating the construction.

5.3 Lock-Based Implementations

The time-complexity results we presented in this section establish that, for any obstruction-free implementation of a large class of objects, a single operation may incur a delay of length $\Omega(\sqrt{n})$ in an execution in which no process is aware of step contention, regardless of how processes proceed when they do become aware of step contention. Moreover, if processes are not allowed to fail their operations even in the face of step contention, then any process can be made to incur a delay of at least $n-1$ stalls on account of an event it issues while not being aware of step contention.

A natural question that arises is whether lock-based implementations of such objects exist that are not subject to these lower bounds. In other words, is there a separation in terms of worst-case operation time complexity between obstruction-free and lock-based implementations? We now show that this is indeed the case. Consider an implementation of a perturbable or binary consensus object that is guarded by a lock; that is, only the process that holds the lock can invoke an operation on the object. The lock can be implemented by using a binary tournament-tree implementation of mutual exclusion, employing read and write operations, such as the tournament tree algorithm due to Peterson and Fischer [1977]. The operations of capturing the lock and releasing it access $O(\log n)$ distinct base objects and incur $O(\log n)$ stalls. Hence there are lock-based implementations that do not incur the complexities of Theorems 6, 7 and 9.

One may argue that this complexity separation is artificial, since processes may perform an unbounded number of steps while they wait for a lock. This is only true, however, if processes busy-wait. If processes signal each other by using semaphores [Dijkstra 2002] or condition variables [Lea 1999] instead of employing busy-waiting, then they can indeed avoid the high worst-case complexities incurred by obstruction-free implementations.[2]

6. DISCUSSION

This paper studies obstruction-free implementations by defining their specification, presenting generic object implementations, and proving lower bounds on their complexity. Our lower bounds concentrate on the cost in uncontended executions (which

[2]Whether or not semaphores should be used instead of busy-waiting in practice obviously depends on system parameters, such as the actual level of contention for the lock and the context-switch time.

are argued to be the most frequent in practice); some of them do not restrict the behavior of the processes in contended situations where processes might be using locks, randomization or other expensive mechanisms. By measuring the complexity of the weakest form of implementations that do not use locks known to date, our results capture the seemingly inherent cost of avoiding locking.

We have established tight bounds on the cost of *generic* obstruction-free implementations. There might be more efficient obstruction-free solutions for specific problems. For obstruction-free consensus, for example, an $\Omega(\sqrt{n})$ lower bound on the number of registers (or historyless objects) was proved by Fich et al. [1998]. There is however a gap since the upper bound is $O(n)$; moreover, these results do not bound the *step contention-free complexity* of obstruction-free consensus. It would be interesting to close the gap between these upper bounds and the logarithmic lower bound we prove for the step contention-free complexity of any generic solo-fast implementation.

Luchangco et al. [2003] presented a generic object implementation that uses only constant number of reads and writes when an operation runs in the absence of point contention. Unfortunately, in their implementation this also means lack of pending operations; moreover, an operation invoked after a prefix with point contention may have to apply C&S primitives, even if it does not encounter any point contention itself. In this paper, we gave a generic solo-fast implementation. Our solo-fast implementation performs $O(n)$ reads and writes, even in the absence of step contention. By employing adaptive collect [Afek et al. 1999; Attiya and Fouren 2003], the step complexity can be made to depend only on the point contention, so it is constant when there is no point contention. By employing adaptive collect for unbounded concurrency [Gafni et al. 2001], it can be made independent of the number of processes.

Aguilera et al. [2006] proposed the concept of *abortable* objects that are, at first glance, similar to our obstruction-free objects. Abortable objects are *live*: operations on them return either matching responses or a predefined $\perp$ value. To prevent trivial implementations and ensure some level of progress in face of failures, they satisfy an additional *non-triviality* requirement that $\perp$ can only be returned in the presence of *interval* contention, and in the eventual absence of step contention, repeated invocations may return $\perp$ only finitely many times. As a result, a suspended operation may prevent a concurrent operation from making progress only for a bounded period of time. Aguilera et al. [2006] argue that if $\perp$ is allowed to be output only in the presence of *step* contention, then the resulting implementations may not preserve correctness under *composition*: by replacing base object of a correct (linearizable, live and non-trivial) implementation with its correct implementation from some finer base objects, we may not obtain a correct implementation from the finer base objects. As we show in the appendix, composability can be achieved at the expense of a slightly stronger definition of obstruction-freedom based on step contention. We propose a definition of a live object that is only allowed to return $\perp$ in the presence of step contention with an additional guarantee that an operation may only take effect within the interval of one of its occurrences. We also provide a generic implementation of such objects from read-write objects.

ACKNOWLEDGMENTS

We would like to thank Partha Dutta, Ron Levy, and Eric Ruppert for important discussions on the topic of this paper, and Eshcar Hillel and the referees for helpful comments on an earlier draft.

REFERENCES

AFEK, Y., STUPP, G., AND TOUITOU, D. 1999. Long-lived adaptive collect with applications. In *Proceedings of the 40th Annual Symposium on Foundations of Computer Science (FOCS)*. IEEE Computer Society, Washington, DC, USA, 262–272.

AFEK, Y., STUPP, G., AND TOUITOU, D. 2002. Long-lived adaptive splitter and applications. *Distributed Computing 15,* 2, 67–86.

AGUILERA, M. K. AND FRØLUND, S. 2003. Strict linearizability and the power of aborting. Tech. Rep. HPL-2003-241, HP Laboratories Palo Alto. Dec.

AGUILERA, M. K., FROLUND, S., HADZILACOS, V., HORN, S. L., AND TOUEG, S. 2006. Abortable shared objects. In *Proceedings of the 20th International Symposium on Distributed Computing (DISC)*. Springer-Verlag, Berlin / Heidelberg, 534–536.

ASPNES, J. AND HERLIHY, M. 1990. Fast randomized consensus using shared memory. *J. Algorithms 11,* 3, 441–461.

ATTIYA, H. AND FOUREN, A. 2003. Algorithms adapting to point contention. *J. ACM 50,* 4, 444–468.

ATTIYA, H., GUERRAOUI, R., HENLDER, D., AND KOUZNETSOV, P. 2006. Synchronizing without locks is inherently expensive. In *Proceedings of the 25th Annual ACM Symposium on Principles of Distributed Computing (PODC)*. ACM, New York, NY, USA, 300 – 307.

ATTIYA, H., GUERRAOUI, R., AND KOUZNETSOV, P. 2005. Computing with reads and writes in the absence of step contention. In *Proceedings of the 19th International Symposium on Distributed Computing (DISC)*. Springer-Verlag, Berlin / Heidelberg, 122–136.

ATTIYA, H. AND WELCH, J. L. 2004. *Distributed Computing: Fundamentals, Simulations and Advanced Topics*, 2nd ed. John Wiley & Sons, Hoboken, New Jersey.

BOICHAT, R., DUTTA, P., FRØLUND, S., AND GUERRAOUI, R. 2003. Deconstructing Paxos. *ACM SIGACT News Distributed Computing Column 34,* 1 (March), 47 – 67. Revised version of EPFL Technical Report 200106, January 2001.

DIJKSTRA, E. W. 2002. Cooperating sequential processes. In *The origin of concurrent programming: from semaphores to remote procedure calls.* Springer-Verlag New York, Inc., New York, NY, USA, 65–138.

DWORK, C., HERLIHY, M., AND WAARTS, O. 1997. Contention in shared memory algorithms. *Journal of the ACM 44,* 6, 779–805.

FICH, F., HERLIHY, M., AND SHAVIT, N. 1998. On the space complexity of randomized synchronization. *J. ACM 45,* 5, 843–862.

FICH, F. E., HENDLER, D., AND SHAVIT, N. 2005. Linear lower bounds on real-world implementations of concurrent objects. In *Proceedings of the 46th Annual Symposium on Foundations of Computer Science (FOCS)*. IEEE Computer Society, Washington, DC, USA, 165–173.

GAFNI, E., MERRITT, M., AND TAUBENFELD, G. 2001. The concurrency hierarchy, and algorithms for unbounded concurrency. In *Proceedings of the 20th Annual ACM Symposium on Principles of Distributed Computing (PODC)*. ACM, New York, NY, USA, 161–169.

HENDLER, D. AND SHAVIT, N. 2008. Solo-valency and the cost of coordination. *Distributed Computing 21,* 1, 43–54.

HERLIHY, M. 1991. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems 13,* 1 (January), 124–149.

HERLIHY, M., LUCHANGCO, V., AND MOIR, M. 2003. Obstruction-free synchronization: Double-ended queues as an example. In *Proceedings of the 23rd IEEE International Conference on Distributed Computing Systems (ICDCS)*. IEEE Computer Society, Washington, DC, USA, 522–529.

Herlihy, M., Luchangco, V., Moir, M., and Scherer III, W. N. 2003. Software transactional memory for dynamic-sized data structures. In *Proceedings of the 22nd Annual ACM Symposium on Principles of Distributed Computing (PODC)*. ACM, New York, NY, USA, 92–101.

Herlihy, M. and Wing, J. M. 1990. Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems 12,* 3 (June), 463–492.

Jayanti, P., Tan, K., and Toueg, S. 2000. Time and space lower bounds for nonblocking implementations. *SIAM Journal on Computing 30,* 2, 438–456.

Lamport, L. 1998. The part-time parliament. *ACM Transactions on Computer Systems 16,* 2 (May), 133–169.

Lea, D. 1999. *Concurrent Programming in Java(TM): Design Principles and Patterns*, 2nd ed. Addison-Wesley, Boston, MA, USA.

Loui, M. C. and Abu-Amara, H. H. 1987. *Memory Requirements for Agreement Among Unreliable Asynchronous Processes.* JAI Press, Greenwich, Conn., 163–183.

Luchangco, V., Moir, M., and Shavit, N. 2003. On the uncontended complexity of consensus. In *Proceedings of the 17th International Symposium on Distributed Computing (DISC)*. Springer-Verlag, Berlin / Heidelberg, 45–59.

Peterson, G. L. and Fischer, M. J. 1977. Economical solutions for the critical section problem in a distributed system (extended abstract). In *Proceedings of the 9th annual ACM symposium on Theory of computing (STOC)*. ACM, New York, NY, USA, 91–97.

Yang, J., Neiger, G., and Gafni, E. 1998. Structured derivations of consensus algorithms for failure detectors. In *Proceedings of the 17th Annual ACM Symposium on Principles of Distributed Computing (PODC)*. ACM, New York, NY, USA, 297–306.

A. EXTENDED INTERFACE OF OBSTRUCTION-FREE OBJECTS

In this section, we explore the possibility of enriching the interface of an obstruction-free implementation with an additional *pause* repsonse $\perp$. As we observed (Theorem 1), a consensus implementation from read-write registers cannot be made wait-free even if operations are allowed to return failure response in the face of step contention. Thus allowing an implementation to return a fail indication in the face of step contention does not guarantee wait-freedom in general. This may be inconvenient if we want every invocation to eventually return control to the client.

We therefore consider an extended interface between clients and implementations that guarantees that every invocation eventually returns. In the extended interface, each operation should either return a legal response for the invoked operation, or, if there is step contention, one of two special values $\perp$ and $\emptyset$. If the implementation is certain that the operation did not have an effect, $\emptyset$ is returned, indicating that the client is free to invoke any operation it wishes. Otherwise, a special *pause* value $\perp$ is returned, and the client must re-invoke the same operation until a non-$\perp$ response is received. To make sure that the implementation does not always fall back to outputting $\perp$, we require that $\perp$ can only be returned if the invoked operation can take effect in *some* extension of the current execution in which that operation is not re-invoked.

A.1 Definitions

We extend R, the set of responses of an object, with two special values: a *pause* value $\perp \notin R$ and a *fail* value $\emptyset \notin R$. The definition of a *well-formed* local history is extended to require that if an invocation i is followed by the response $\perp$, then the subsequent event, if one exists, is i.

The definition of linearizability is further extended so that invocations returning *fail* are removed from the linearized history, while at most one of identical invocations returning *pause* is considered as one operation.

Specifically, consider an execution α; the corresponding *history* is obtained as follows. For every invocation i in α that is not followed by a response in R, we insert in α a response $\perp$ immediately after the last event of the process that invokes i in α. Now the history H is defined as a subsequence of α that consists of invocation and response of high-level operations (including $\emptyset$ and $\perp$ responses). A *local* history of α is a subsequence of H consisting of all events of some process p. We only consider executions α that induce well-formed local histories.

Let $\bar{H}$ be any local history of α. Let i be an invocation in $\bar{H}$ on an object x. A fragment of the form i, r in $\bar{H}$, where $r \in R$, is called an *occurrence of* i (returning r). Since an invocation occurrence may return $\perp$ and be re-invoked later, there can be a number of occurrences of i in a history. Consider the longest fragment of the form i or $i, \perp, i, \ldots, \perp, i$ in $\bar{H}$. Recall that we completed every pending invocation in α with $\perp$, so each such fragment in $\bar{H}$ is followed by a matching response $r \in R$. If $r \notin \{\perp, \emptyset\}$, we call i, r or $i, \perp, i \ldots, \perp, i, r$ a *complete operation.* If $r = \emptyset$, we call $i, \emptyset$ or $i, \perp, i \ldots, \perp, i, \emptyset$ a *failed operation.* If $r = \perp$, we call $i, \perp$ or $i, \perp, i, \ldots, \perp, i$ or $i, \perp, i \ldots, \perp, i, \perp$ a *pending operation.* Since $\bar{H}$ is well-formed, a pending operation is a suffix of $\bar{H}$ ($\perp$ cannot be followed by an invocation other than i).

Now we say that a well-formed history H is *linearizable* if there is a *sequential* history H' containing occurrences of all the complete operations in H and a subset of occurrences of the pending operations in H, with at most one occurrence per operation, in which each $\perp$ is replaced with a matching response $r \in R - \{\emptyset, \perp\}$, such that H' is legal and it respects the order of non-concurrent operation occurrences in H. When taken in the context of the extended notions of complete and pending operations, a history is linearizable if we can assign *linearization points* to all complete operations and a subset of pending operations so that the linearization point assigned to an operation may only belong the interval of one of the operation's occurrences.

An implementation is *linearizable* if it produces only linearizable histories. An implementation is *live* if every invocation occurrence i returns in a finite number of its own steps (although a value in $\{\perp, \emptyset\}$ can be returned). A linearizable implementation is *valid* if (1) an invocation occurrence returns $\perp$ only when it is not step-contention free, (2) an invocation occurrence i returns $\emptyset$ only when the corresponding *operation* (the longest fragment of the form $i, \emptyset$ or $i, \perp, i, \ldots, \perp, i, \emptyset$ in the local history) is not step-contention free, and (3) if an invocation i performed by p_i returns $\perp$ at the end of an execution α, then i is nontrivial and there exists an execution α' that agrees with α on the state of p_i, i. e., $\alpha|i = \alpha'|i$, and i appears in some linearization of $H(\alpha')$. The last property provides a non-triviality criterion on the use of pause responses: an invocation made by a process p_i is allowed to return $\perp$ only if the corresponding operation takes effect in some indistinguishable (for p_i) execution. It is immediate that any live and valid implementation is obstruction-free.

Note that our definition ensures that in any execution of a linearizable implementation, every operation takes effect (if it does) within one of its occurrences, and, in particular, before it stops taking steps in that execution. As a result, if a process fails (stops taking steps) before completing its operation's occurrence, then the operation takes effect (if it does) *before* the process fails, making the implementations *strictly linearizable* [Aguilera and Frølund 2003].

A.2 Generic Object Implementation

This section presents a generic obstruction-free (live and valid) implementation of any type. Our generic construction extends the universal nonblocking implementation of Herlihy [1991] to handle pause and fail responses in the presence of step contention. By the validity requirement, an operation invoked by a process p_i is only allowed to return $\perp$ if its invocation takes effect in *some* indistinguishable (for p_i) execution. This makes our extension of Herlihy's construction nontrivial: we need to carefully account for the cases in which $\perp$ cannot be returned.

As in Herlihy's construction, our implementation is built from consensus objects, which in our case are obstruction-free.[3]

The implementation (presented in Figure 4) follows a deterministic sequential implementation of an object type T. An object is represented as a linked list; an element of the list represents an operation applied to the object. The list of operations clearly determines the list of corresponding responses. A process makes an invocation by appending a new element to the end of the list. The algorithm assumes a function $\mathit{response}(\mathit{invs}, \mathit{inv})$ that returns the response matching the invocation *inv* in a sequential execution of invocations from list *invs* (under the condition that $\mathit{inv} \in \mathit{invs}$).

We assume that invocations corresponding to different operations are uniquely identifiable. The algorithm uses the following shared variables:

– n atomic single-writer, multi-reader registers $L_1, \ldots, L_n$. Process p_i stores in L_i its last view of the object state in the form of a linked list of operations that p_i observed to have been applied on the object.
– $C[\,]$ is an unbounded array of obstruction-free (linearizable, live and valid) consensus objects. The array is used to agree on the order in which invocations are put into the linked list of operations $L_1, \ldots, L_n$.

Note that we do not assume that lines of the pseudo-code presented in Figure 4 are executed atomically. Informally, the algorithm works as follows. When a process p_i executes an invocation *inv*, it identifies the longest list L_j (line 2). Let $k = |L_j|$ be the number of operations on the object witnessed by p_j. If *inv* is already in L_j, the response associated with *inv* in L_j is returned (line 5). This ensures that *inv* takes effect at most once, even if repeated several times. If it is not the first instance of *inv*, and $k > |L_i|$ (i.e., *inv* did not "win" any OF Consensus to which it was proposed), then p_i returns $\emptyset$ (line 9). Otherwise, p_i proposes *inv* to $C[k+1]$ (line 10). If $C[k+1]$ returns $\perp$ (which can happen only if step contention is detected), then p_i returns $\perp$ (line 14). If the propose operation (a) returns $\emptyset$, or (b) returns a non-*inv* response and the current invocation occurrenceis not the first occurrence of *inv* (i.e., if *repeated* is *true*), then p_i returns $\emptyset$ (line 19). The condition

[3]For completeness, a simple linearizable, live and valid implementation of consensus, derived from Paxos-style consensus algorithms, appears in Appendix B.2.

repeated is *true* in (b) is needed since, otherwise, a non-$\{inv, \bot, \emptyset\}$ invocation could have been returned by $C[k]$ in the absence of step contention.

If $C[k+1]$ returns *inv*, then p_i returns the response associated with *inv* (line 23). Otherwise, the procedure is repeated, now at position $k+2$. If $C[k+2]$ returns a non-$\{inv, \bot\}$ response, then p_i returns $\emptyset$ (line 36). The second consensus operation ensures that the implementation is valid, namely, that (1) $\bot$ is never returned if the corresponding invocation instance is step-contention free, (2) $\emptyset$ is never returned if the corresponding *operation* is step-contention free, and (3) if $\emptyset$ is never returned if i is nontrivial and there exists an execution α' that agrees with α on the state of p_i, i.e., $\alpha|i = \alpha'|i$, and i appears in some linearization of $H(\alpha')$.

This algorithm implies the next theorem:

THEOREM 10. *Every sequential type T has a linearizable, live and valid implementation from registers.*

PROOF. We immediately observe that the implementation in Figure 4 is live: it contains no blocking statements and the underlying obstruction-free consensus objects are live. We prove now that it is valid and linearizable.

Let α be any well-formed finite execution of the implementation in Figure 4. First, we complete each pending invocation i in α with a $\bot$ response put immediately after the last step of the process that invoked i. Now every invocation i in α is followed by a matching response.

We say that an invocation *inv* is *fixed at index* k in α, and we write *fixed*(α, inv, k), if some propose operation on $C[k]$ returns *inv* in α (in line 10 or 24). Since each $C[k]$ is a consensus object, at most one invocation can be fixed at every index. We say that index k is *decided in* α if there is an invocation i_k such that *fixed*(α, i_k, k). Let k^* be the highest decided index in α. Since a process only accesses an OF consensus object $C[k+1]$ if some process previously obtained a non-$\{\bot, \emptyset\}$ from $C[k]$, we have the following property.

. **P1** Each $k = 1, \ldots, k^*$ is decided.

Since each $C[k]$ is a consensus implementation, it is allowed to return at most one non-$\{\bot, \emptyset\}$ value in α. Thus, for each $k = 1, \ldots, k^*$, we can define r_k as the response returned by i_k in a sequential order of invocations $i_1, i_2, \ldots, i_k$ (according to the sequential specification of T). Hence, we have the following property:

. **P2** For each $k = 1, \ldots, k^*$, if i_k returns a non-$\{\bot, \emptyset\}$ response, then the response is r_k.

Now we show that α includes no occurrence i_k, r $(k = 1, \ldots, k^*)$ where $r = \emptyset$. Indeed, i_k returns $\emptyset$ (i) in lines 9, 19 or 19 only if i_k is not fixed in any OF consensus object to which it was proposed so far (verified in lines 3 and A.2, or line A.2 or line A.2, respectively), (ii)in lines 16 or 30 only if i_k is trivial and object $C[k]$ or $C[k+1]$ returns $\bot$ (lines A.2 and A.2 or A.2 and A.2, respectively). Thus, given P2, a fixed invocation can only return $\bot$ or r_k in α.

Recall that each $C[k]$ is linearizable. For each $k = 1, \ldots, k^*$, some invocation $C[k]$.*propose*(i_k) *takes effect*, i.e., $C[k]$.*propose*(i_k) is included in every linearization of the execution on $C[k]$. Thus, for each $k = 1, \ldots, k^*$, we can determine o_k be the

```
Shared variables:
    Register $L_1, \ldots, L_n \leftarrow \emptyset, \ldots, \emptyset$
    Obstruction-free (live and valid) consensus objects $C[\,]$
Local variables:
    $repeated \leftarrow false$;  $dec \leftarrow \bot$; $invs \leftarrow \emptyset$;

1:  upon Invoking $inv$ do
2:      $invs \leftarrow longest(\{L_1, \ldots, L_n\})$          // Select the longest invocation list
3:      if $inv \in invs$ then
4:          $repeated \leftarrow false$
5:          return $response(invs, inv)$              // Return if $inv$ is already completed
6:      $k \leftarrow |invs|$
7:      if $(k > |L_i|)$ and $repeated$ then
8:          $repeated \leftarrow false$
9:          return $\emptyset$                                              // Fail the operation
10:     $dec \leftarrow C[k+1].propose(inv)$                   // The 1st consensus operation
11:     if $dec = \bot$ then
12:         $repeated \leftarrow true$
13:         if $inv$ is nontrivial then
14:             return $\bot$
15:         else
16:             return $\emptyset$
17:     if $(dec = \emptyset)$ or $(dec \neq inv$ and $repeated)$ then
18:         $repeated \leftarrow false$
19:         return $\emptyset$                                              // Fail the operation
20:     $invs \leftarrow invs \cdot dec$; $L_i \leftarrow invs$                                  // Update $L_i$
21:     if $dec = inv$ then
22:         $repeated \leftarrow false$
23:         return $response(invs, inv)$                       // Return if $inv$ is decided
24:     $dec \leftarrow C[k+2].propose(inv)$                   // The 2nd consensus operation
25:     if $dec = \bot$ then
26:         $repeated \leftarrow true$
27:         if $inv$ is nontrivial then
28:             return $\bot$
29:         else
30:             return $\emptyset$
31:     if $dec \neq \emptyset$ then
32:         $invs \leftarrow invs \cdot dec$; $L_i \leftarrow invs$
33:     if $dec = inv$ then
34:         $repeated \leftarrow false$
35:         return $response(invs, inv)$                       // Return if $inv$ is decided
36:     return $\emptyset$                                       // Fail if $inv$ is ignored twice
```

Fig. 4. An obstruction-free implementation of T: code for process p_i

occurrence i_k, r which includes the invocation $C[k].propose(i_k)$ that takes effect in α.

Now we construct a sequential history $H' = o'_1, o'_2, \ldots, o'_{k^*}$ where for each $k = 1, \ldots, k^*$, $o'_k = i_k, r_k$. Note that H' contains no invocation that returned $\emptyset$ in α, every invocation that returned $r \notin \{\perp, \emptyset\}$, and a subset of invocations that returned $\perp$.

By construction H' is legal. To show that α is linearizable, it sufficient to prove that H' preserves the order of non-concurrent occurrences $o_1, \ldots, o_k$ in α.

Indeed, suppose that o_ℓ precedes o_m in α. Since the indexes of o_ks in H' are monotonically increasing it is enough to show that $\ell < k$. Suppose, by contradiction, that $ell \geq m$. Since all i_ks ($k = 1, \ldots, k^*$) are distinct,$ell > m$. By definition, o_ℓ and o_m contain the linearized invocations $C[\ell].propose(i_\ell)$ and $C[m].propose(i_m)$, respectively. By the algorithm (lines 2 and 20), a process p_j may only access $C[\ell]$ if all $k < \ell$ are already fixed, i.e., the process previously made sure that for some $L_j \in \{L_1, \ldots, L_n\}$, $|L_j| = \ell - 1$. But since $C[m]$ is linearizable, it could not return r_m *before* it was linearized, i.e., before the beginning of the occurrence o_m. Thus, m is not fixed in the shortest prefix of α that includes the invocation of o_ℓ—a contradiction.

Finally, we prove that our implementation is valid. Assume that *inv* returns a value in $\{\perp, \emptyset\}$. We have the following cases:

(1) $\perp$ is returned in line 14 or line 28. Thus, a propose operation on $C[k+1]$ or $C[k+2]$ returned $\perp$. This can only happen if the current occurrence of *inv* is not step contention-free, nontrivial and there was a concurrent propose operation on $C[k+1]$ or $C[k+2]$. Since each $C[k']$, $k' \in \mathbb{N}$, is valid, there is an execution α' such that p_i cannot distinguish the prefix of α up to returning $\perp$ and α', and $\mathit{fixed}(\alpha', inv, k+1)$ or $\mathit{fixed}(\alpha', inv, k+2)$. Thus, *inv* takes effect in α' and α' is indistinguishable from the shortest prefix of α in which *inv* returns $\perp$.

(2) $\emptyset$ is returned in line 9. This can only happen when there were at least one prior occurrence of *inv* and all previous occurrences of *inv* returned $\perp$. Thus, the corresponding *operation* is not step contention-free.
Note that if p_i reaches line 10 while *repeated* is true, then *inv* is not fixed at any $k' \leq k$ and the last propose(*inv*) invoked by p_i on $C[k+1]$ returned $\perp$.

(3) $\emptyset$ is returned in line 16 or line 30. This can only happen if *inv* is trivial. Also, similarly to case (2) above, the current invocation of of *inv* is not step-contention free.

(4) $\emptyset$ is returned in line 19. That is, either $C[k+1]$ returned $\emptyset$, or *inv* was not fixed at any $k' \leq k+1$. Thus, *inv* did not take effect, and the corresponding operation was not step-contention free.

(5) $\emptyset$ is returned in line 36. That is, *inv* is not fixed at any index less than $k+2$ and $C[k+2]$ returned either $\emptyset$ or a value $dec \notin \{\perp, \emptyset, inv\}$. In the former case, since the corresponding propose(*inv*) operation is allowed to return $\emptyset$ only if it is not step-contention free and it did not take effect, the current instance of *inv* is not step-contention free and it did not take effect either.
Now assume that $C[k+2]$ returned $dec \notin \{\perp, \emptyset, inv\}$ and let p_j be the process that previously proposed *dec* to $C[k+2]$. By the algorithm, before proposing,

p_j has made sure that for some $p_l \in \Pi$, $|L_l| = k+1$ (lines 2, 6 and 20). But the longest list in $\{L_1, \ldots, L_n\}$ seen by p_i in the beginning of the current instance of *inv* had length k (line 6). Thus, p_l took steps in the interval of the current instance of *inv*, i.e., the instance is not step-contention free. Moreover, *inv* was not fixed at any $k' \leq k+2$, and thus did not take effect.

Thus, the implementation is linearizable, live, and valid. □

B. CONSENSUS IMPLEMENTATIONS

In this section, we give two simple implementations of obstruction-free consensus in the read-write shared memory model: one for the standard interface presented in Section 3, and one for the extended interface introduced in Appendix A.

B.1 Obstruction-Free Consensus

As suggested by Herlihy et al. [2003], an obstruction-free consensus algorithm can be derived by "de-randomizing" a randomized consensus algorithm [Aspnes and Herlihy 1990]. Figure 5 depicts a simple implementation of consensus from registers that is allowed to block or return fail in case of step contention. For simplicity, we assume that processes propose distinct values. (This is the way consensus is used in our paper.)

Our implementation is based on the *adopt-commit* protocol of Yang et al. [1998]. The one-shot *adopt-commit* abstraction accepts a value $v \in V$ as an argument and returns a special *abort* value ($abort \notin V$), or a pair (c, v'), where c is a boolean and $v' \in V$. If (c, v') is returned and $c = false$, we say that v' is *adopted.* If (c, v') is returned and $c = true$, we say that v' is *committed.* The abstraction guarantees that (i) every adopted or committed value is an input value of some process; (ii) no two processes commit on or adopt different values, and (iii) if all inputs are the same, then no process aborts or adopts a value (every process that returns must commit). Also, it is easy to observe that the adopt-commit protocol proposed by Yang et al. [1998] guarantees an additional property: if a process proposes a value v and obtains *abort*, then no other process can adopt or commit v.

In the algorithm, every process executes repeated instances of the *adopt-commit* protocol with monotonically increasing indexes, accepting all adopted values as its decision estimates, until some value is committed or *abort* is returned. Then the process returns the committed value or $\emptyset$ (in case an instance of adopt-commit returns *abort*).

Obviously in a step contention-free execution, eventually at most one value will be proposed to some instance of the adopt-commit protocol, and this value will be committed. On the other hand, if a value committed in some instance of adopt-commit, then no process can adopt or commit a different value. If *abort* is returned in instance k of adopt-commit, then the proposed value was not committed in any instance $k' < k$ and cannot be adopted in instance k.

B.2 Obstruction-Free Consensus with Extended Interface

Here we present a linearizable, live and valid consensus implementation. Interestingly, our algorithm (Figure 6) translates the long-lived ◇Register implementation [Boichat et al. 2003], from message-passing to read-write shared memory.

```
upon propose($v$) do
   $v_i := v$
   while ($true$) do
      $r = adopt\text{-}commit(v_i)$
      if $r = (true, v')$ then
         return $v'$
      else if $r = (false, v')$ then
         $v_i := v'$
      else return $\emptyset$
```

Fig. 5. Obstruction-free consensus: code for process p_i

Every process p_i maintains a current estimate of the decision value of the consensus, denoted by v_i and initialized to $\perp$, and two counters r_i and w_i, both initialized to 0. The counter r_i denotes the round number adopted by the current operation, and w_i denotes the last round number in which p_i "announced" its estimate v_i. Each process p_i is designated a single-writer multi-reader register R_i, initialized to $(0, 0, \perp)$ that is written by p_i and read by all processes. A boolean *repeated*, initialized to *false*, indicates whether this is not the first occurrence of the current operation.

Roughly, the algorithm can be decomposed into two phases. In the first phase, process p_i chooses the highest unique round number, "registers" the round (writes in R_i, line 5), and collects the shared memory. In the second phase, p_i adopts the value announced in the highest round (or its own proposal if no value is announced so far) as v_i, "announces" it with the current round number, and collects the shared memory again. If p_i observes that some process registered a higher round (which can occur only when there is step contention) and a value different from v_i is announced in a higher round (which can only happen if the value proposed by p_i cannot be decided), then p_i returns $\emptyset$ (line 16). If step contention is detected and v_i is still the highest announced value, then p_i returns $\perp$ (line 19). Otherwise, p_i returns v_i. These two phases (register and announce) ensure that once p_i returns v_i, no process will ever announce a value different from v_i in a higher round. As a result, no two processes can return different non-$\{\perp, \emptyset\}$ values, no value in $\{\perp, \emptyset\}$ can be returned in the absence of step contention, $\emptyset$ is never returned if the corresponding operation took effect, and $\perp$ is never returned if there is an indistinguishable execution in which the operation did take effect.

Theorem 11. *There is a linearizable, live and valid consensus implementation from registers.*

Proof. The implementation in Figure 6 is clearly live, since there are no cycles or wait statements in the code.

Let α be any finite well-formed execution of the algorithm in which some non-$\{\perp, \emptyset\}$ value v is returned by some process. (If there is no such value, the execution is trivially linearizable.) To prove linearizability of the implementation, it is sufficient to show that (i) every returned non-$\{\perp, \emptyset\}$ value is a previously proposed value, and (ii) no two processes return different non-$\{\perp, \emptyset\}$ values.

We observe that (i) follows directly from the algorithm.

To prove (ii), we first show that if a process p_i returns a non-$\{\perp, \emptyset\}$ value v in a

```
Shared variables:
    $Register\ R_1, \ldots, R_n \leftarrow (0, 0, \perp), \ldots, (0, 0, \perp)$
Local variables:
    $r_i \leftarrow 0;\ w_i \leftarrow 0;\ v_i \leftarrow \perp;\ repeated \leftarrow false$

1:  upon propose($v$) do
2:     $regSet \leftarrow \{R_1, \ldots, R_n\}$
3:     $mr \leftarrow \max\{r : (r, *, *) \in regSet\}$              // Adopt the highest round number
4:     $r_i \leftarrow$ the smallest integer s.t. $((r_i\ mod\ n = i)$ and $(r_i > mr))$
5:     $R_i \leftarrow (r_i, w_i, v_i)$                                      // Register the round number
6:     $regSet \leftarrow \{R_1, \ldots, R_n\}$
7:     choose $v'$ s.t. $(*, mw, v') \in regSet$ and $mw = \max\{w : (*, w, *) \in regSet\}$
                                                    // Choose the "highest" announced value
8:     if $(v' \neq \perp)$ then $v_i \leftarrow v'$ else $v_i \leftarrow v$
9:     $w_i \leftarrow r_i$
10:    $R_i \leftarrow (r_i, w_i, v_i)$                                  // Announce the current estimate
11:    $repeated \leftarrow true$
12:    $regSet \leftarrow \{R_1, \ldots, R_n\}$
13:    if$(\exists (r, *, *) \in regSet$ s.t. $r > r_i)$ then
14:       if $(\exists (*, w, val) \in regSet$ s.t. $w > r_i$ and $val \neq \perp$ and $val \neq v_i)$ then
15:          $repeated \leftarrow false$
16:          return $\emptyset$                  // Fail if a different value is announced in a higher round
17:       else
18:          $repeated \leftarrow true$
19:          return $\perp$                                                          // Pause
20:    else
21:       $repeated \leftarrow false$
22:       return $v_i$                                                    // Decide on $v_i$
```

Fig. 6. Consensus implementation with extended interface: code for process p_i

round r, then no process can announce a value different from v in a higher round. Indeed, let p_j be the first process to announce a value v' in a round $r' > r$. We immediately observe that p_j registered r' (line 5) *after* p_i announced v in round r (otherwise, p_i would see that a round higher than r is registered in line 13 and return $\perp$ or $\emptyset$). Thus, when p_j collects the shared memory in round r' (line 6), v is the value announced in the highest round number. Thus, $v = v'$. Since before returning a non-$\{\perp, \emptyset\}$ value, every process announces the value, we have (ii).

We conclude by showing that the implementation is valid (recall the definition in Appendix A.1). Responses in $\{\perp, \emptyset\}$ are returned only if step contention is detected (line 13). Specifically, if $\emptyset$ is returned (line 16), then p_i made sure that its current estimate is not the value announced in the highest round (line 14), so the estimate cannot be decided. That is, the propose(v) operation of p_i did not take effect in the execution. If $\perp$ is returned (line 19), then no process has previously announced a value different from v_i in a higher round. Thus, every process returns v_i in the extension of the current execution in which no process writes a value different from v_i in $\{R_j\}$. $\square$

Received January 2007; revised February 2008; accepted February 2009