Randomized Wait-Free Concurrent Objects (extended abstract)

Maurice Herlihy
Digital Equipment Corporation
Cambridge Research Laboratory
One Kendall Square
Cambridge MA, 02139

Digital Equipment Corporation
Cambridge Research Lab

CRL 91/5 June 3, 1991

Abstract

A *concurrent object* is a data structure shared by concurrent processes. A *wait-free* implementation of a concurrent object guarantees that every operation completes in a finite number of steps, regardless of how processes interleave. It is known, however, that if concurrent processes communicate only by applying read and write operations to a shared memory, then it is impossible to construct wait-free implementations of many simple and useful data objects. In this paper we show how to construct *randomized* wait-free implementations of long-lived concurrent objects, implementations that guarantee that every operation completes in a finite *expected* number of steps, even against a powerful adversary.

This paper will appear in the Tenth Annual ACM Symposium on Principles of Distributed Computing, August 19-21, Montreal, Canada.

1 Introduction

A *concurrent object* is a data structure shared by asynchronous concurrent processes. An implementation of a concurrent object is *wait-free* if it guarantees that every operation completes a finite number of steps, regardless of how process steps are interleaved. An implementation is *randomized* wait-free if it guarantees that every operation completes in a finite *expected* number of steps. The wait-free condition guarantees that no process can be prevented from completing an operation by variations in other processes' speeds, or by undetected halting failures. This condition rules out many conventional algorithmic techniques such as busy-waiting, conditional waiting, critical sections, or barrier synchronization, since the failure or delay of a single process can prevent the non-faulty processes from making progress.

In this paper, we propose new algorithms for constructing randomized wait-free implementations of arbitrary objects in an architecture where processes communicate by applying *read* and *write* operations to locations in shared memory. Randomization is necessary because this kind of architecture does not support wait-free solutions to many fundamental problems [3, 11, 13, 17, 22], including basic decision problems such as consensus [14], and implementations of many simple data objects such as sets, queues, or lists. We give a general algorithm for constructing a randomized implementation of any *read-modify-write* [19] operation. Our construction uses no unbounded registers, and has worst-case time and space complexity identical to that of the best randomized consensus protocols known for this model [4, 7, 27].

The principal contribution of this work is to show how techniques developed for short-lived decision problems can be adapted to long-lived data objects. Much of the work on randomized wait-free synchronization algorithms concerns decision problems, such as consensus, in which a protocol is run only once. Each process simply executes its part of the protocol and halts, and the shared data structures do not need to be reused. By contrast, practical applications such as operating systems and data bases are organized around long-lived data objects, which are inherently more difficult than decision problems. A data object has an unbounded lifetime during which each process can execute an arbitrary sequence of operations. Unlike a decision protocol, an object implementation must ensure that the size of the object's representation remains bounded even when the number of operations applied to it increases without limit. It must retain enough information to ensure that "sleepy" processes that arbitrarily suspend and resume execution can continue to progress, while discarding enough information to keep the object size bounded. A wait-free object implementation must also guard against starvation, since one operation can be "overtaken" by an arbitrary sequence of other operations, a problem that does not arise in decision protocols.

2 Preliminaries

A *concurrent system* consists of a collection of n *processes* that communicate through shared typed *objects*. Processes are *sequential* — each process applies a sequence of operations to objects, alternately issuing an invocation and then receiving the associated response. We make no fairness assumptions about processes. A process can halt, or display arbitrary variations in speed. In particular, one process cannot tell whether another has halted or is just running very slowly.

Objects are data structures in memory. Each object has a *type*, which defines a set of possible *values* and a set of primitive *operations* that provide the only means to manipulate that object. Each object has a *sequential specification* that defines how the object behaves when its operations are invoked one at a time by a single process. For example, the behavior of a queue object can be specified by requiring that *enq* insert an item in the queue, and that *deq* remove the oldest item in the queue. In a concurrent system, however, an object's operations can be invoked by concurrent processes, and it is necessary to give a meaning to interleaved operation executions. An object is *linearizable* [18] if each operation appears to take effect instantaneously at some point between the operation's invocation and response. Linearizability implies that processes appear to be interleaved at the granularity of complete operations, and that the order of non-overlapping operations is preserved.

We focus on an asynchronous multiple instruction/multiple data (MIMD) architecture in which shared memory consists of a sequence of locations, called *registers*, that can be written by (at least) one process and read by any process. Our time and space complexity measures are expressed in terms of read and write operations on registers of size $O(n)$. Polynomial-time algorithms for implementing large single-writer/multi-reader registers from small ones are well-known [10, 20, 21, 24, 25, 28].

A *read-modify-write* operation [19] is defined as follows. Let x be an object, v its current value, and f a function from values to values. The operation $RMW(x, f)$ atomically replaces the value of x with $f(v)$, and returns v (Figure 1). Although *read-modify-write* operations implemented in hardware typically work on single words of memory, software implementations such as the one given here can work on objects of arbitrary size. The class of *read-modify-write* operations is *universal*, in the sense that one can implement any operation (such as *enq* or *deq*) from a suitable *read-modify-write* operation. A second approach to universality is to focus on a particular *read-modify-write* operation, such as *compare&swap*, shown in Figure 2. Elsewhere [15], we give a detailed methodology for transforming a stylized sequential implementation of any object into a wait-free linearizable implementation in which processes synchronize using only *read*, *write*, and *compare&swap* operations.

Our construction also makes use of an *atomic snapshot scan* algorithm. Given an n-element array A, where P is the only process that writes $A[P]$, the snapshot scan makes an instantaneous (linearizable) copy of A. Atomic

```
RMW(r: register, f: function) returns(value)
   previous := r
   r := f(r)
   return previous
   end RMW
```

Figure 1: A Read-Modify-Write Operation

```
compare&swap(w: word, old, new: value)
   returns(boolean)
   if w = old
      then w := new
           return true
      else return false
      end if
   end compare&swap
```

Figure 2: The Compare&Swap Operation

snapshot scan algorithms have been proposed by Anderson [2] (bounded registers and exponential running time), Aspnes and Herlihy [6] (unbounded registers and $O(n^2)$ running time), and by Afek et al. (bounded registers and $O(n^2)$ running time). Of these, the last algorithm is the best suited for our purposes, since it is both bounded and efficient.

3 Randomized Consensus

The heart of our construction is a *binary consensus protocol*, in which each of n asynchronous processes starts with a *preference* taken from a two-element set (typically $\{0,1\}$), and runs until it chooses a *decision value* and halts. The protocol is correct if it is *consistent:* no two processes choose different decision values, *valid:* the decision value is some process's preference, and *randomized wait-free:* each process decides after a finite expected number of steps. When computing a protocol's expected number of steps, we assume that scheduling decisions are made by an *adversary* with unlimited resources and complete knowledge of the processes' protocols, their internal states, and the state of the shared memory. The adversary cannot, however, predict future coin flips.

Our technique can be applied to a variety of randomized binary consensus protocols, but to keep our discussion as concrete as possible, we focus on the simplest such protocol, due to Aspnes [4], shown in Figure 3. Here, n processes collectively undertake a one-dimensional random walk centered at the origin with absorbing barriers at $\pm 2n$. The protocol makes use of three counters: $\mathcal{P}_0$ and $\mathcal{P}_1$ are the respective number of processes that prefer 0 and 1, and $\mathcal{C}$ is the cursor for the random walk. Each process alternates between reading the three shared counters (using a single atomic scan) and updating C. Eventually the counter reaches one of the absorbing barriers, determining the decision value. While the counter is near the middle of the region, each process flips an unbiased coin to determine the direction in which to move the counter. If the counter is sufficiently close to one of the barriers, however, each process moves it deterministically toward that barrier. Each shared counter (Figure 4) is implemented as an n-element array of integers, one per process. To increment or decrement the counter, a process updates its own field. To read the counter, it atomically scans all the fields. Careful use of modular arithmetic ensures that all values remain bounded. The expected running time of this protocol, expressed in primitive reads and writes, is $O(n^4)$, and the space complexity is $O(n^2)$.

Any binary consensus protocol can be extended to a consensus protocol in which process take their preferences from an arbitrary set. Consider a $\log n$-depth binary tree where each leaf is initially associated with a process. In the first round, each process performs binary consensus with its immediate neighbor, and each process adopts the resulting decision value as its next preference. At round r, two groups of 2^{r-1} processes, each with a common preference, achieve binary consensus to decide the common preference for both groups at the next level. The protocol terminates after $\log n$ rounds of binary consensus, when all processes have a common preference. The simple inequalities

$$\sum_{i=1}^{\log n} 2^{4i} < (\sum_{i=1}^{\log n} 2^i)^4 < (2n)^4$$

imply that the asymptotic expected running time for the multi-value consensus protocol is identical to that of binary consensus:

$$O(\sum_{i=1}^{\log n} 2^{4i}) \quad = \quad O(n^4)$$

The same property holds for any polynomial-time protocol. The space complexity increases from $O(n^2)$ to $O(n^2 \log n)$.

4 Formal Model

A more complete version of our formal model appears elsewhere [17]. Formally, we model objects and processes as non-deterministic automata, a simplified form of the I/O automata of Lynch and Tuttle [23]. Each automaton has a set of *states*, sets of *input*, *output*, and *internal events*, and a *transition relation* given by a set of triples (s', e, s), where s and s' are states and e is an event. Such a triple is called a *step*, and it means that an automaton in state s' can undergo

```
shared data:
  P_0: counter with range [0...n]
  P_1: counter with range [0...n]
  C: counter with range [-3n...3n]

consensus(prefer) returns(Boolean)
   inc(P_prefer)
   loop
         p_0, p_1, c := read(P_0, P_1, C)
         select
           case c <= -2n do return 0
           case c >= 2n do return 1
           case c <= -(p_0 + p_1) or p_1 = 0 do inc(C)
           case c <= (p_0 + p_1) or p_0 = 0 do dec(C)
           otherwise do
           if flip()
                then inc(C)
                else  dec(C)
                end if
           end select
         end loop
   end consensus
```

Figure 3: A Randomized Binary Consensus Protocol

a transition to state s, and that transition is associated with the event e. If (s', e, s) is a step, we say that e is *enabled* in s'. Inputs cannot be disabled: for each input event e and each state s', there exist a state s and a step (s', e, s).

An *execution fragment* of an automaton A is a finite sequence $s_0, e_1, s_1, \ldots e_n, s_n$ or infinite sequence $s_0, e_1, s_1, \ldots$ of alternating states and events such that each (s_i, e_{i+1}, s_{i+1}) is a step of A. An *execution* is an execution fragment where s_0 is a starting state. A *history fragment* of an automaton is the subsequence of events occurring in an execution fragment, and a *history* is the subsequence occurring in an execution.

Automata can be *composed* if they share no output or internal events. A state of the composed automaton S is a tuple of component states, and a starting state is a tuple of component starting states. The set of events of S is the union of the components' sets of events. The set of output events of S is the union of the components' sets of output events; the set of internal events is the union of the components' sets of internal events; and the set of input events of S is the set of input events of S that are not output events for some component. A triple (s', e, s) is in a step of S if and only if, for all component automata A,

$[-r, r]$ is the range of the counter
m is any integer greater than $2r + 1$

```
inc(C)
   v := C[P]
   write(v + 1 mod m, C[P])
   end inc

dec(C)
   v := C[P]
   write(v − 1 mod m, C[P])
   end dec

read(C) returns(integer)
   c := scan(C)
   v := Σ_{i=0}^{n-1} c[i]
   return v' where  −r ≤ v' ≤ r
                    and v' ≡ v   (mod m)
   end read
```

Figure 4: Counter Implementation

one of the following holds: (1) e is an event of A, and the projection of the step onto A is a step of A, or (2) e is not an event of A, and A's state components are identical in s' and s. If H is a history of a composite automaton and A a component, then $H|A$ denotes the subhistory of H consisting of events of A.

Processes and objects are each modeled as automata. Operation invocations are modeled as output events of processes, and input events of objects, while operation results are input events of processes and output events of objects. To capture the notion that a process represents a single thread of control, we say that a process history is *well-formed* if it begins with an invocation and alternates matching invocations and responses. A *concurrent system* $\{P_1, \ldots, P_n; A_1, \ldots, A_m\}$ is an automaton composed from processes $P_1, \ldots, P_n$ and objects $A_1, \ldots, A_m$, where processes and objects are composed by identifying corresponding invocations and responses. A history of a concurrent system is *well-formed* if each $H|P_i$ is well-formed, and a concurrent system is *well-formed* if each of its histories is well-formed. Henceforth, we restrict our attention to well-formed concurrent systems.

An execution is *sequential* if its first event is an invocation, and it alternates matching invocations and responses. A history is sequential if it is derived from a sequential execution. (Notice that a sequential execution permits process steps to be interleaved, but at the granularity of complete operations.) If we restrict our attention to sequential histories, then the behavior of an object can be specified in a particularly simple way: by giving pre- and postconditions for each operation. We refer to such a specification as a *sequential specification.*

If H is a history, let $complete(H)$ denote the maximal subsequence of H consisting only of invocations and matching responses. Each history H induces a partial *precedence* order $\prec_H$ on its operations: $p \prec_H q$ if the response for p precedes the invocation for q. Operations unrelated by $\prec_H$ are said to be *concurrent.* If H is sequential, $\prec_H$ is a total order. A concurrent system $\{P_1, \ldots, P_n; A_1, \ldots, A_m\}$ is *linearizable* if, each history H can be extended to a well-formed history H', by adding zero or more responses, for each history H, there exists a sequential history S such that:

- For all P_i, $complete(H')|P_i = S|P_i$
- $\prec_H \subset \prec_S$

In other words, the history "appears" sequential to each individual process, and this apparent sequential interleaving respects the real-time precedence ordering of operations. Equivalently, each operation appears to take effect instantaneously at some point between its invocation and its response. A concurrent object A is *linearizable* [18] if, for every history H of every concurrent system $\{P_1, \ldots, P_n; A_1, \ldots, A_j, \ldots, A_m\}$, $H|A_j$ is linearizable. A linearizable object is thus "equivalent" to a sequential object, and its operations can also be specified by simple pre- and postconditions. We restrict our attention to linearizable concurrent systems.

An *implementation* of an object A is a concurrent system $\{F_1, \ldots, F_n; R\}$, where the F_i are called *front-ends*, and R is called the *representation.* Informally, R is the data structure that implements A, and F_i is the procedure called by process P_i to execute an operation. Each invocation of an operation of A is an input event of F_i, and each response is an output event of F_i. An implementation I_j of A_j is *correct* if the two systems are indistinguishable to the ensemble of processes: for every history H of $\{P_1, \ldots, P_n; A_1, \ldots, I_j, \ldots, A_m\}$, there exists a history H' of $\{P_1, \ldots, P_n; A_1, \ldots, A_j, \ldots, A_m\}$, such that $H|\{P_1, \ldots, P_n\} = H'|\{P_1, \ldots, P_n\}$.

An implementation is *wait-free* [17] if:

- It has no history in which an invocation of P_i remains pending across an infinite number of steps of F_i.
- If P_i has a pending invocation in a state s, then there exists a history fragment starting from s, consisting entirely of events of F_i and R, that includes the response to that invocation.

The first condition rules out unbounded busy-waiting: a front-end cannot take an infinite number of steps without responding to an invocation. The second condition rules out conditional waiting: F_i cannot block waiting for another process to make a condition true.

In this paper, the representation object R is an array of *registers* that provide linearizable *read* and *write* operations. We also use a *scan* procedure, which has a wait-free implementation using *read* and *write*, as well as a *consensus* procedure, which has a randomized wait-free implementation. For brevity, our algorithms are expressed in pseudocode, although it is straightforward to translate this notation into automata definitions.

5 The Algorithm

The processes execute a sequence of consensus protocols, called *rounds*, which determine the order in which concurrent operations are applied. We say a process *wins* round r if its preference is that round's decision value, otherwise it *loses*. A *sleepy* process is one that suspends execution in the middle of a round, and resumes after a later round has started.

Recall that each shared counter $\mathcal{C}$ used by the consensus protocol is implemented as an array of (bounded) integer values, where each integer represents that process's contribution to the counter's value. A unbounded technique that permits different rounds to share the same data structures is simply to tag each counter register with the current round number. When a process reads a register, it ignores values tagged with earlier round numbers, treating those registers as uninitialized. When a sleepy process reads a value tagged with a later round number, it quits the protocol, since information it needs may have been overwritten. The sleepy process then inspects other data structures to reconstruct whether it had won the interrupted round.

To avoid unbounded round numbers, we introduce the notion of a (bounded) *leadership graph*, a concept adapted from the *distance graph* construction of Attiya, Dolev, and Shavit [7]. A leadership graph is a graph whose vertices are connected by directed or undirected edges. A *path* in a leadership graph from vertex v_0 to vertex v_k is a sequence of vertices $v_0, \ldots, v_k$ such that for each v_i, v_{i+1}, either there is an edge directed from v_i to v_{i+1}, or there is an undirected edge between them. A *directed path* is a path that includes at least one directed edge. We say that w is *ahead* of v ($w > v$) if there is a directed path from v to w, and that v and w are *even* ($v \sim w$) if neither is ahead of the other. Informally, if w is ahead of v, then w's consensus round preceded v's. A *leader* is a vertex with no directed paths to other vertices. A *k-generation* leadership graph is one where there are k vertices associated with each process. These k vertices will be connected by directed edges, and the last of these is the process's *latest* vertex.

A vertex in a leadership graph is represented by a *round vector*, which is an n-element array of *round counters*. A round counter assumes values in the range $\{0, \ldots, 2k\}$. Value a is considered "greater" than $a-1, \ldots a-k$, and "less" than $a+1,\ldots,a+k$, where all arithmetic is modulo $2k+1$. If v and v' are vectors associated with P and Q, then there is an edge directed from v to v' if $v[Q] < v'[P]$, and there is an undirected edge if $v[Q] = v'[P]$. Our protocol uses a 2-generation leadership graph.

New round vectors are created by the *advance* operation, shown in Figure 5. This procedure takes as arguments a leadership graph $\mathcal{G}$ and a process P. It generates a new round vector for P (without modifying the graph itself). It erases any outgoing edges from P's most recent vector, adds incoming edges from any previously unrelated vectors, and leaves existing incoming edges alone.

A *recycling counter* implementation appears in Figure 5. Each element of the $\mathcal{C}$ array is now an entry consisting of an integer value tagged with one or more round vectors. A process P reads the counter by scanning $\mathcal{C}$, and reconstructing a leadership graph from the round vectors. If P's latest round vector is not a leader, then P is sleepy, and it exits the operation by signaling an exception. Otherwise, it sums and returns the contributions of the other processes whose most recent round vectors are leaders.

The *recycling consensus* protocol is constructed by replacing the counter in Figure 3 with a recycling counter. This protocol allows a single data structure to be reused for multiple consensus protocols. A sleepy process that attempts to read the counters will receive an exception. In such a case, it is convenient to define the protocol to return the distinguished value $\perp$.

When a sleepy process discovers that it is no longer a leader, it must determine whether it won its last round. We incorporate a bounded amount of history information by associating a boolean *toggle* value with each operation. Each time a process starts a new operation, it complements its toggle bit. Each preference has an additional field: an n-element boolean *horizon* array. The toggle bit of the last round won by P is kept in $p.horizon[P]$. When a sleepy process P discovers it has been overtaken, it locates the winning preference p from any later round (as described below). P won the interrupted round if and only if its own toggle bit matches $p.horizon[P]$.

A process starting a round must be able to reconstruct the object's current state. Our protocol maintains a 2-generation leadership graph $\mathcal{G}$ in which each process P keeps a *past* vector from the last round it completed, and a *present* vector from the the last round it started. Each *past* vector is tagged with either the winning preference from that round or with the distinguished value $\perp$. A *leading past vector* is one that has no directed paths leading to any other past vectors. The *latest* procedure scans $\mathcal{G}$, locates the leading past vectors, and returns any associated preference distinct from $\perp$.

The final issue concerns starvation. A naive approach is simply to use each consensus round to decide which operation is scheduled next. Such an algorithm is non-blocking, in the sense that the system as a whole continually makes progress, but it is not wait-free, since an individual process may starve if it loses every round. Instead, we use a form of *software combining* to guarantee that each operation completes in at most two rounds. Each process *announces* its intention to execute an operation, and each process collects recently announced operations, and constructs its preference by applying them in sequence.

We are now ready to present the complete protocol. We use the following data types and structures. An *invocation* is a record with two fields: a toggle bit and a function. A *preference* is a record with three fields: *p.value* is the object's new value if the process wins the consensus round, *p.response* an n-element array of values, and *p.horizon* an n-element array of toggle bits.

An *entry* is a record with the following fields:

- *past* is the round vector from the last round P completed.
- *winner* is the winning preference from that round, or $\perp$.
- *present* is the round vector from the current round. This is the vector used by the consensus protocol to detect sleepy processes (as in Figure 5).
- *counters* is a $3 \cdot \log n$-element array of integers used for the counters needed by the recycling consensus protocol's random walks.

The processes share an n-element array $\mathcal{A}$ of invocations, and an n-element array of entries $\mathcal{G}$. The entries' *past* and *present* fields define a 2-generation leadership graph. A *leading past vector* is one that has no directed paths leading to any other past vectors. We use two auxiliary procedures: *scan* takes an atomic snapshot scan of an arbitrary array, and *latest* scans $\mathcal{G}$, locates the leading past vectors, and returns an associated *winner* preference distinct from $\perp$.

The pseudo-code for $RMW(f)$, where f is a function, appears in Figure 5. P first creates a new invocation, complements the previous invocation's toggle bit, and stores the new invocation in $\mathcal{A}$. P reads its *present* vector from $\mathcal{G}$, and then calls *latest* to get the winning preference from the most recently completed round. It then calls *make-prefer* (Figure 5) to create a new preference. This procedure copies the last winning preference, and scans the $\mathcal{A}$ array. It then locates unapplied invocations by comparing the invocations' toggle bits with the corresponding bits in the preference's horizon array. It applies any new invocations, storing their responses in the new preference's *response* field, and their toggle bits in the *horizon* field. After creating the new preference, P then joins the recycling consensus protocol, returning either the winning preference or $\perp$. P then creates a new entry for its next operation, setting *past* to the current value of *present*, *winner* to the outcome of the protocol (if known) or $\perp$, *present* to the result of *advance*, and *counters* to an array of $3 \cdot \log n$ zeroes. After executing this loop twice, P calls *latest* to locate the latest winning preference, and extracts its response to its operation from the preference's *response* field.

6 Correctness

We give an explicit linearization: for any execution of the protocol, we construct an equivalent legal serial execution. Because of space limitations, some proofs are omitted or sketched.

Lemma 1 *Being even is an equivalence relation: if $u \sim v$ and $v \sim w$, then $u \sim w$.*

Proof: Every two vertices are joined by an edge, either directed or undirected, hence if $u \sim v$, then there is an undirected edge between them. Suppose $u \sim v$ and $v \sim w$, but $u < w$. Then there is a directed path from v to w constructed by joining the undirected edge from v to u to the directed path from u to v, contradicting the hypothesis that $u \sim v$. ∎

Since reasoning about round numbers is easier than reasoning about round vectors, we introduce unbounded round numbers as *auxiliary data*, variables which do not affect the protocol's control flow. We tag every round vector v in $\mathcal{G}$ with an unbounded round number $\hat{v}$ defined as follows. Initially, all round vectors have round number zero. Suppose when P calls *advance*, g is the scanned copy of $\mathcal{G}$, r is the highest round number in g, and r_P is P's highest round number in g. If v is the round vector returned by *advance*, then $\hat{v} = \max(r_P + 1, r)$.

Lemma 2 *If* advance *returns a vector with round number r, then for every integer between 0 and r, there exists a vector with that round number.*

Proof: Initially all vectors have round number 0, and each call to *advance* increases the maximum round number by at most one. ∎

If P sets its *past* vector to v, then we say that P writes $\hat{v}$ to *past*($\mathcal{G}$), and similarly for *present*($\mathcal{G}$).

Lemma 3 *If P is the first process to write r to present($\mathcal{G}$), then it simultaneously writes $r-1$ to past($\mathcal{G}$).*

Proof: We show that if P's call to *advance* returns round number r, then P's *present* vector must have round number $r-1$. By Lemma 2, if *advance* returns r, then round vectors exist with round number $r-1$. If P's *present* vector has a round number less than $r-1$, then *advance* would return $r-1$, since no higher round number appears in the graph. ∎

The next lemma follows directly from the definition of round numbers.

Lemma 4 *If $v \sim w$, then $\hat{v} = \hat{w}$.*

Let $present_P$ and $past_P$ denote P's present and past vectors. The two next lemmas follow from simple inductive arguments:

Lemma 5 $|present_P[Q] - present_Q[P]| \leq 1$

Lemma 6 $present_P[Q] - past_P[Q] \leq 1$

Lemma 7 *If $v < w$, then $\hat{v} < \hat{w}$.*

Proof: Suppose the property is violated by vectors v and w, generated by P and Q, such that $\hat{v} < \hat{w}$ but $v > w$. We first claim that one can choose v and w so that there is a directed edge from w to v (i.e., $v[Q] > w[P]$). Choose v and w with a minimal-length directed path between them. If that path has the form $w, w', \ldots v$, then either $\hat{w} < \hat{w}'$, in which case w and w' are closer, or $\hat{w} \geq \hat{w}'$, in which case w' and v are closer.

If $\hat{v} < \hat{w}$, then P's scan occurred before Q's. We will show that if P's scan occurs first, then there can be no directed edge from w to v (i.e. $w[P] \geq v[Q]$).

Suppose that v was present in $\mathcal{G}$ when Q performed its scan. If v is a present vector, then Lemma 5 implies that $w[P] = v[Q]$ or $w[P] = v[Q] + 1$. If v is a past vector, and Q's present vector is v', then, as before, $w[P] = v'[Q]$ or $w[P] = v'[Q]+1$, and Lemma 6 implies that either $w[P] = v[Q]$, $w[P] = v[Q]+1$, or $w[P] = v[Q] + 2$. In both cases, $w[P] \geq v[Q]$.

Suppose that v was not yet written to $\mathcal{G}$ when Q performed its scan. Let v' be P's present vector during Q's scan, and let w' be Q's present vector during P's scan. (Note that P's scan is earlier, and that Q can do an arbitrary number of scans between P's scan and the scan in which it constructed w. During Q's scans, however, P's vectors remain fixed.) There are three cases to consider. If $v'[Q] = w'[P]$, then $v[Q] = w'[P] + 1$ and $w[P] = v'[Q] + 1$, hence $w[P] = v[Q]$. If $v'[Q] > w'[P]$, then $v[Q] = w'[P]$ and either $w[P] = v'[Q]$ or $w[P] = v'[Q]+1$, hence $w[P] = v[Q]$ or $w[P] = v[Q]+1$. Finally, if $v'[Q] < w'[P]$, then $v[Q] = w'[Q]$ and $w[P] = w'[P]$, hence $v[Q] = w[P]$. In all three cases, $w[P] \geq v[Q]$. ∎

A round is *complete* if it has been written to *past*($\mathcal{G}$).

Lemmas 4 and 7 imply that we can view the consensus protocols as taking place in disjoint rounds, where the decision value for $r - 1$ is chosen before that of round r. A process *joins* consensus round r if it joins the consensus protocol while its *present* vector has round number r. P completes the protocol *successfully* if it returns a preference distinct from $\perp$, and *unsuccessfully* otherwise.

Lemma 8 *For each completed round r, some process joins consensus round r and completes successfully.*

Proof: Each process alternates executing the consensus protocol and writing a new vector to *present*($\mathcal{G}$). Consider the first process to write $r+1$ to *present*($\mathcal{G}$). By Lemma 3, its previous *present* vector had round number r, hence it joined consensus round r. It must have completed successfully, since it could not have encountered any vectors with higher round numbers. ∎

We now show that the *latest* procedure can always find a value to return:

Lemma 9 *Some leading* past *vector is always associated with a preference.*

Proof: The first process to write $r+1$ to *present*($\mathcal{G}$) is also the first to write r to *past*($\mathcal{G}$), and its *winner* field is not equal to $\perp$ (Lemma 8). ∎

Lemma 10 *If P successfully completes consensus round r, then the value returned by its call to* latest *is the decision value for round* $r-1$.

Proof: Lemma 7 implies that the leading *past* vector is associated with the most recent round in *past*($\mathcal{G}$). Lemma 9 implies that this round is associated with decision value distinct from $\perp$, hence *latest* returns a value. Lemma 3 implies that round $r-1$ has been written to *past*($\mathcal{G}$), so the decision value returned belongs to a round greater than or equal to $r-1$. If the latest round in *past*($\mathcal{G}$)is greater than $r-1$, then the latest round in *present*($\mathcal{G}$)is greater than r, and p could not have completed successfully. Therefore, the value returned by *latest* must be the decision value for round $r-1$. ∎

An invocation p is *announced* when it is written to $\mathcal{A}$, it is *observed* by P when P applies it to a value as part of constructing a preference, and it is *executed* if P wins that round.

Lemma 11 *Each invocation is executed at most once.*

Proof: After the invocation is executed, and before a new invocation is announced, the toggle bit in the invocation agrees with the corresponding toggle bit in future winning preferences' horizon fields, so *make-prefer* will skip that invocation. ∎

Lemma 12 *If an invocation is announced during round r, then it will be executed either in round r or* $r+1$.

Proof: Suppose p is announced during round r but not executed in that round. Let P be the process that wins round $r+1$. Since p's announcement precedes P's call to *make-prefer*, the invocation's toggle bit will disagree with the corresponding horizon bit in the preference returned by P's call to *latest*, and P will observe p. ∎

Theorem 13 *The protocol in Figure 5 implements a randomized wait-free linearizable read-modify-write operation.*

Proof: Executions of the consensus protocols occur as a sequence of rounds (Lemma 7), where each round has a unique winner (Lemma 8). Define the object's *state* at round r to be the *value* field of the winning preference for that round. The object's state at round r is constructed by applying a sequence of invocations to the object's state at round $r-1$ (Lemma 10). The resulting execution is equivalent to a sequential execution in which operations are ordered by the round number in which they were executed, and operations in the same round are ordered by in the order they were observed by that round's winner. Every invocation is executed after at most two rounds (Lemma 12), and every invocation is executed exactly once (Lemma 11). If one operation starts after another returns, then the later operation will have a higher round number, and therefore the sequential order is a valid linearization order.

Finally, the protocol is randomized wait-free, since each execution of the consensus protocol is randomized wait-free, and the encompassing protocol terminates after a fixed number of steps. ∎

7 Discussion and Related Work

We remark that the universal protocol given here can be optimized when something is known about the function f. For example, Figure 7 shows an optimized implementation of *compare&swap* [1] This implementation returns immediately if the object's state fails to match the *old* argument, and it eliminates the *response* field by having each winner return *true* and each loser return *false*. A *read* operation is even simpler: it just returns latest$(\mathcal{G}).value$.

Fischer, Lynch, and Paterson [14] show that there is no consensus protocol for two processes that communicate by asynchronous messages. Dolev, Dwork, and Stockmeyer [12] and Dwork, Lynch, and Stockmeyer [13] give a comprehensive analysis of the circumstances under which consensus can be achieved by message-passing. Ben-Or [9] proposes a randomized consensus protocol with exponential expected running time that tolerates up to $n/5$ failures, where n is the number of processes. Loui and Abu-Amara [22] give several consensus protocols and impossibility results for processes that communicate through shared registers with various read-modify-write ("test-and-set") operations. Chor, Israeli and Li [11] give two randomized consensus protocols for shared read/write registers, one for two processes, and one for three processes. These protocols, however, run against a weaker adversary than the others cited here.

The first shared-memory protocol that runs against a strong adversary is due to Abrahamson [1]. This protocol has exponential expected running time. The first polynomial-time protocol is an unbounded protocol due to Aspnes and Herlihy [5]. This protocol introduced the use of random walk, the basic technique behind all known polynomial-time protocols. Attiya, Dolev, and Shavit [7] show how to eliminate unbounded counters from the original random walk algorithm, and Saks, Shavit, and Woll [27] show how to make the protocol fast when processes run in approximate synchrony. Bar-Noy and Dolev [8] adapt the random walk protocol to a message-passing model, yielding the fastest known consensus protocol for that model.

[1] This procedure implements a slightly restricted *compare&swap* in which the *old* and *new* arguments must be distinct.

The author [17] has shown that n-process consensus is *universal* in a system of n processes: given a synchronization primitive that solves n-process consensus, one can construct a deterministic wait-free implementation of any object. Plotkin [26] also gives a universal construction using a particular *read-modify-write* primitive called a *sticky-bit*. The author [15] gives a more more time- and space-efficient universal construction using *read*, *write* and the well-known *compare&swap* instruction. If the shared memory provides only *read* and *write* operations, then Herlihy and Aspnes [6] have shown that one can construct a deterministic wait-free implementation of any object whose operations either commute with or overwrite one another. The author [16] gives a more general characterization of the objects that do have deterministic wait-free or non-blocking implementations in this model.

References

[1] K. Abrahamson. On achieving consensus using a shared memory. In *Seventh ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, pages 291–302, August 1988.

[2] J. Anderson. Composite registers. In *Proceedings of the 9th ACM Symposium on Principles of Distributed Computing*, pages 15–30, August 1990.

[3] J.H. Anderson and M.G. Gouda. The virtue of patience: Concurrent programming with and without waiting. Private Communication.

[4] J. Aspnes. Time- and space-efficient randomized consensus. In *Proceedings of the 9th ACM Symposium on Principles of Distributed Computing*, pages 325–332, August 1990.

[5] J. Aspnes and M.P. Herlihy. Fast randomized consensus using shared memory. *Journal of Algorithms*, 15(1):441–460, September 1990.

[6] J. Aspnes and M.P. Herlihy. Wait-free data structures in the asynchronous pram model. In *Proceedings of the 2nd Annual Symposium on Parallel Algorithms and Architectures*, pages 340–349, July 1990.

[7] H. Attiya, D. Dolev, and N. Shavit. Bounded polynomial randomized consensus. In *Eighth ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, pages 281–294, August 1989.

[8] A. Bar-Noy and D. Dolev. Shared memory vs. message-passing in an asynchronous distributed environment. In *Eighth ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, pages 307–318, August 1989.

[9] M. Ben-Or. Another advantage of free choice: completely asynchronous agreement protocols. In *Second ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, pages 27–30, August 1983.

[10] J.E. Burns and G.L. Peterson. Constructing multi-reader atomic values from non-atomic values. In *Proceedings of the Sixth ACM Symposium on Principles of Distributed Computing*, pages 222–231, 1987.

[11] B. Chor, A. Israeli, and M. Li. On processor coordination using asynchronous hardware. In *Proceedings of the Sixth ACM Symposium on Principles of Distributed Computing*, pages 86–97, 1987.

[12] D. Dolev, C. Dwork, and L Stockmeyer. On the minimal synchronism needed for distributed consensus. *Journal of the ACM*, 34(1):77–97, January 1987.

[13] C. Dwork, N. Lynch, and L Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM*, 35(2):228–323, April 1988.

[14] M. Fischer, N.A. Lynch, and M.S. Paterson. Impossibility of distributed commit with one faulty process. *Journal of the ACM*, 32(2), April 1985.

[15] M.P. Herlihy. A methodology for implementing highly concurrent data structures. In *Proceedings of the 2nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 197–206, March 1990.

[16] M.P. Herlihy. Impossibility results for asynchronous pram. In *Proceedings of the 3rd ACM Symposium on Parallel Architectures and Algorithms*, July 1991. to appear.

[17] M.P. Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):124–149, January 1991.

[18] M.P. Herlihy and J.M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, July 1990.

[19] C.P. Kruskal, L. Rudolph, and M. Snir. Efficient synchronization on multiprocessors with shared memory. In *Fifth ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, pages 218–228, August 1986.

[20] L. Lamport. Concurrent reading and writing. *Communications of the ACM*, 20(11):806–811, November 1977.

[21] L. Lamport. On interprocess communication, parts i and ii. *Distributed Computing*, 1:77–101, 1986.

[22] M.C. Loui and H.H. Abu-Amara. *Advances in Computing Research*, volume 4, chapter Memory Requirements for Agreement Among Unreliable Asynchronous Processes, pages 163–183. JAI Press, 1987.

[23] N.A. Lynch and M.R. Tuttle. An introduction to input/output automata. Technical Report MIT/LCS/TM-373, MIT Laboratory for Computer Science, November 1988.

[24] R. Newman-Wolfe. A protocol for wait-free, atomic, multi-reader shared variables. In *Proceedings of the Sixth ACM Symposium on Principles of Distributed Computing*, pages 232–249, 1987.

[25] G.L. Peterson. Concurrent reading while writing. *ACM Transactions on Programming Languages and Systems*, 5(1):46–55, January 1983.

[26] S.A. Plotkin. Sticky bits and universality of consensus. In *Proceedings of the Eighth ACM Symposium on Principles of Distributed Computing*, pages 159–176, 1989.

[27] M. Saks, N. Shavit, and H. Woll. Optimal time randomized consensus - making resilient algorithms fast in practice. In *Proceedings of the ACM Symposium on Discrete Algorithms*, January 1991.

[28] A.K. Singh, J.H. Anderson, and M.G. Gouda. The elusive atomic register revisited. In *Proceedings of the Sixth ACM Symposium on Principles of Distributed Computing*, pages 206–221, August 1987.

```
[−r, r] is the range of the counter
m is any integer greater than 2r + 1

inc(C)
   v := C[P]
   v.value := v.value+1   (mod m)
   C[P] := v
   end inc

dec(C)
   v := C[P]
   v.value := v.value−1   (mod m)
   C[P] := v
   end dec

read(C) returns(integer) signals(quit)
   c := scan(C)
   g := leadership graph from c
   p := P's most recent vector in g
   if p ∉ leader(g)
      then signal quit
      end if
   v := 0
   for Q in 0...n − 1 do
         q := Q's most recent vector in g
         if q ∈ leader(g)
            then v := v + c[Q].value
            end if
         end for
   return v' where  −r ≤ v' ≤ r
                    and v' ≡ v   (mod m)
   end read
```

Figure 5: Recycling Counter Implementation

```
advance(𝒢, P) returns(round-vector)
   r := new round-vector
   g := scan(𝒢)
   p := P's most recent vector in g
   for Q in 0..n − 1 do
      q := Q's most recent vector in g
      select
        case p[Q] < q[P] do r[Q] := q[P]
        case p[Q] = q[P] do r[Q] := p[Q] + 1
        case p[Q] > q[P] do r[Q] := p[Q]
        end select
      end for
   return r
   end advance
```

Figure 6: The Advance Procedure

```
RMW(f: function) returns(boolean)
   toggle := ¬ 𝒜[P].toggle
   𝒜[P] := [toggle: toggle, function: f]
   for i in 1..2 do
      last := latest(𝒢)
      prefer := make-prefer(last.winner)
      decision := consensus(prefer)
      𝒢[P] := [past: 𝒢[P].present
               winner: decision,
               present: advance(𝒢, P),
               counters: [0, ..., 0]]
      end for
   return latest(𝒢).winner.response[P]
   end RMW
```

Figure 7: Read-Modify-Write

```
make-prefer(old: preference) returns(preference)
  new := copy(old)
  a := scan(𝒜)
  for Q in 0...n − 1 do
     if a[Q].toggle ≠ old.horizon[Q]
       then new.horizon[Q] := ¬ new.horizon[Q]
            new.response[Q] := new.value
            new.value := a[Q].function(new.value)
       end if
     end for
  return new
  end make-prefer
```

Figure 8: The Make-Prefer Operation

```
compare&swap(old, new: value) returns(boolean)
  for i in 1..2 do
     last := latest(𝒢)
     if last.value ≠ old
       then return false
       end if
     prefer := [value: new, horizon: last.horizon]
     toggle := ¬ last.horizon[P]
     prefer.horizon[P] := toggle
     decision := consensus(prefer)
     𝒢[P] := [past: 𝒢[P].present,
              winner: decision,
              present: advance(𝒢, P)]
     if latest(𝒢).winner.horizon[P] = toggle
       then return true
       end if
     end for
  return false
  end compare&swap
```

Figure 9: Compare&Swap Construction