Wait-Free Computation in Message-Passing Systems: Preliminary Report

Maurice P. Herlihy Mark R. Tuttle

DEC Cambridge Research Lab
One Kendall Square
Cambridge, MA 02139

Abstract: We explore the time complexity of wait-free implementations of concurrent objects in synchronous, message-passing systems. Our technique is to reduce the (difficult) problem of analyzing all possible wait-free implementations for a particular object to the (more tractable) problem of analyzing a related decision problem. The decision problem we consider is *strong renaming*, in which an arbitrary subset of m out of n processors choose unique names in the range $1 \ldots m$, where m is not known in advance. We prove tight $\log m$ bounds on the number of rounds of communication needed to solve this renaming problem. As a result, we derive corresponding lower bounds for wait-free implementations of a variety of objects such as stacks, queues, priority queues, and fetch&add registers, as well as for decision problems such as ℓ-assignment and order-preserving renaming. Conversely, we show how a particular strong renaming algorithm can be transformed into an $O(m^{\frac{1}{2}+\epsilon})$ implementation of an object called an *increment register*, a substantial improvement over conventional $O(n)$ techniques. Our results suggest the existence of a nontrivial complexity hierarchy for wait-free implementations of concurrent objects.

1 Introduction

A *concurrent object* is a data structure shared by concurrent processors. An implementation of a concurrent object is *wait-free* if every nonfaulty processor can complete any operation in a finite number of steps despite the failure of $n-1$ other processors, where n is the number of processors in the system. In this paper, we prove a new lower bound for the time complexity of wait-free implementations of a class of concurrent objects in message-passing systems. We prove our lower bound in a *synchronous* model where processors may *crash* at any time. As a result, our lower bounds apply to many other message-passing models, including asynchronous and semi-synchronous, as well as models that permit lost messages, Byzantine failures, etc.

Upper bounds for this model are well known. For example, the *state machine* methodology of Lamport [17, 19] and Schneider [22] is a general-purpose technique that can be adapted to a variety of failure models. The object is implemented as a replicated collection of uninterpreted deterministic automata, and atomic broadcast is used to ensure that each automaton executes the same operations in the same order. In the crash failure model, atomic broadcast requires time $\theta(n)$, yielding an immediate upper bound for any wait-free concurrent object. It is also known that consensus [14] can be used to construct wait-free implementations of a concurrent objects in shared-memory models, and it is not difficult to show the same in message-passing models. Consensus, like atomic broadcast, is $\theta(n)$ in this model [11]. Given these bounds, it is natural to ask whether one can achieve better performance by designing "customized" algorithms that exploit the semantics of the object, in the same way that type-specific replication techniques have been developed for the atomic transaction model [13].

Although lower bounds have been extensively studied for decision problems, [3, 5, 7, 8, 9, 11, 21, 24] concurrent objects have received less attention. Our technique is to reduce the (difficult) problem of analyzing all possible wait-free implementations for a particular object to the (more tractable) problem of analyzing a related decision problem. Any lower bound for the decision problem translates immediately to a lower bound for the concurrent object. The lower bounds presented here are derived by analyzing the

This paper appeared in *Proceedings of the 9th Annual ACM Symposium on Principles of Distributed Computing*, pages 347–362, ACM, August 1990.

complexity of the *strong renaming problem*, informally defined as follows: Given a set of n processors with ids taken from an ordered set, design a protocol that allows an arbitrary subset of $1 \leq m \leq n$ processors to choose unique names in the range $1 \ldots m$, where m is not known in advance. This problem is called strong renaming because the range of names chosen must equal the number of participants, which is known to be impossible in asynchronous systems [1, 2]. Renaming algorithms are interesting in their own right and have been extensively studied in asynchronous systems [1, 2].

In this paper, we prove a tight $\log m$ bound on the number of rounds of communication needed to solve the strong renaming problem. This lower bound yields corresponding lower bounds for a variety of interesting data types. For example, it is a simple matter to solve strong renaming given, say, a wait-free first-in-first-out (FIFO) queue. One simply initializes the queue to hold the integers $1 \ldots n$, and each of the m processors dequeues its name. The $\theta(\log m)$ complexity of renaming translates directly into an $\Omega(\log m)$ lower bound on any wait-free implementation of the *dequeue* operation for FIFO queues. Minor variations of this argument yield lower bounds for operations of many other data types, including stacks, priority queues, *fetch&add* registers [16], and others. This argument also yields lower bounds for decision problems that solve strong renaming, including ℓ-assignment [4] and order-preserving renaming [2]. Elsewhere [14], a similar kind of reduction (to consensus [6, 10]) was used to derive impossibility results for wait-free concurrent objects in the asynchronous shared-memory model. Our results here show that reduction to a decision problem can be adapted to yield complexity as well as impossibility results.

How tight are these lower bounds for concurrent objects? In general, we don't know. The inherent complexity of a concurrent object is closely related to the algebraic structure of its operations. For example, consider a FIFO queue extended with a *peek* operation that returns but does not remove the oldest item. It is easy to show that the extended queue can solves consensus, and hence its operations require time $\theta(n)$ in the crash failure model. By contrast, the queue without the *peek* operation cannot solve consensus among three or more processors [14], and indeed, the inherent complexity of a FIFO queue in this model remains an open question. Nevertheless, there do exist simple concurrent objects for which type-specific algorithms yield better than $O(n)$ complexity. In this paper, we give a wait-free implementation of a register that provides an atomic *increment* operation in which each *increment* requires time $O(c^{\frac{1}{2}+\epsilon})$, where c is the number of concurrent operations starting in the same round. Interestingly, this implementation is just a simple transformation of a relatively inefficient $O(m^{\frac{1}{2}+\epsilon})$ strong renaming protocol.

The rest of this paper is organized as follows. We define our model of computation in Section 2. We define the strong renaming problem in Section 3, and in Section 4 we show how to reduce proving lower bounds on the complexity of several concurrent objects to proving lower bounds for the strong renaming problem. In Section 5, we prove our $\log m$ lower bound for strong renaming, and in Section 6 we show that this bound is actually tight with an efficient strong renaming algorithm. In Section 7 we show how to transform a particular renaming algorithm into a wait-free implementation of an increment register. Finally, we close with a discussion of some open problems in Section 8.

2 Model

We begin with a description of our model of computation. Loosely speaking, we consider synchronous systems of n unreliable processors $p_1, \ldots, p_n$ who share a global clock that starts at time 0 and advances in increments of one. We assume that any processor can send a message to any other processor. Computation proceeds via a sequence of *rounds*. Round k lasts from time $k-1$ to k, and consists of three phases. Each processor first receives the messages sent to it by other processors in the preceding round. It then performs some local computation, based on its current local state and the messages just received, to determine its next local state and the messages it wishes to send to other processors. Finally, it sends these messages to the other processors. Communication is reliable, in that a message sent is guaranteed to be delivered, but processors may crash at any time, perhaps after having set only a subset of the messages it wanted to send during that round.

More formally, a *global state* of the system is a tuple $(s_1, \ldots, s_n, e)$, where s_i is the *local state* of processor p_i and e is the state of the *environment*. A processor's local state contains its id, the time on the global clock, and the entire history of messages received from other processors so far. The environment contains the set P of processors in the system, and any other information of relevance to the system that cannot be deduced from processors' local states. An *execution* is a sequence $g_0 g_1 \ldots g_k$ of global states where g_0 is an *initial state*, and g_i is the global state of the system at time i.

We assume that every processor is following a deterministic *protocol* that determines what actions it performs. For our purposes, a protocol is simply a

function from a processor's local state (including its id and the messages it received in the last round) to its next local state and the set of messages it is to send to processors in the next round. Processors always follow their protocol, except that some processors may *crash* (or *fail*) in the middle of a round and do not participate in the protocol from that point on. We assume that the state of the environment contains the set F of processors who have failed in the execution thus far, and a tuple (p, k, S) for every $p \in F$ indicating p failed in round k after sending to processors in S. This means that p sends every message required by the protocol in rounds 1 through $k-1$, sends every message it is required to send to processors in S in round k and fails to send any message it is required to send to processors in $P - S$ in this round, and sends no messages in any later round. A processor is considered *faulty* in an execution if it fails in some round of that execution, and *nonfaulty* otherwise. We say that a processor *survives* its kth round if it has not failed in its first k rounds.

Our primary concern in this paper is the wait-free implementation of concurrent objects. Informally, an *object* is simply a data structure, possibly replicated at multiple processors. (We note in passing that in the message-passing model, where processors do not share memory, any wait-free implementation of an object must be replicated at every processor.) It has a *type*, which defines the set of possible *values* the object can assume, and a set of *operations* that provide the only means to access or modify the object. An operation is split into an *invocation* sent by a processor to an object in one round, and a *response* sent from the object to the processor in a later round.

Each object has a *sequential specification* that defines how it behaves when a sequence of operations are executed one at a time by a single processor; that is, for each operation in the sequence, the processor waits for that operation's response before issuing the next operation's invocation. If the object is shared by concurrent processors, however, it is necessary to give a meaning to executions in which several processors have issued invocations for several operations before receiving the corresponding responses. Loosely speaking, an object is *linearizable* [15] if each operation appears to take effect instantaneously at some point between the operation's invocation and response messages. This means that operations appear to have happened in some sequential order, one that preserves the order of nonconcurrent operations: if one operation ends before another begins, then the first appears to have happened before the second. In this sense, linearizability is a correctness condition for concurrent objects that permits programmers to reason about concurrent objects as if they were sequential.

More precisely, an execution e is *linearizable* if the sequential specification is satisfied by some sequential execution obtained by reordering events in e as follows: for every operation α appearing e, (i) choose some time t_α between α's invocation and response, and (ii) reorder e so that the invocation/response pair for α appears before the invocation/response pair for β iff $t_\alpha < t_\beta$. Notice that linearizability preserves the order of nonoverlapping operations: if α's response appears in an execution before β's invocation, then α is ordered before β. An object is *linearizable* if each of its executions is linearizable. Although linearizability was originally proposed for asynchronous systems, it applies equally well to synchronous systems. It is related to (but not identical to) correctness criteria such as sequential consistency [18] and strict serializability [20]. In this paper, we consider only linearizable, wait-free implementations of concurrent objects.

A processor crash can prevent an operation from ever returning a response. In this case, the linearizability condition permits the operation to be ordered at any point after the invocation, or to have no effect. The response effectively occurs at time infinity, and the operation may appear to occur at any point in the resulting interval, including infinity. In any event, the operation is *atomic*: it either appears to everyone to have taken effect at a particular point, or it appears to everyone never to have happened at all.

3 Strong Renaming

The *strong renaming* problem is defined as follows. Each processor has a unique id taken from a totally-ordered set of ids. Each processor also has a read-only input register and a write-once output register as part of its local state. At time 0, each processor's input register is initialized with either 0 or 1 (meaning "don't participate" or "participate," respectively), and its output register is empty. The initial state of a processor at time 0 consists simply of its id and the contents of its input and output registers. A protocol is a solution for the renaming problem tolerating $n-1$ failures if the output registers of the nonfaulty participating processors contain distinct values from $\{1, \ldots, m\}$ at the end of every execution in which at most $n-1$ processors fail, where $m \leq n$ is the number of participating processors.

4 Wait-free Lower Bounds

Our technique for proving lower bounds on the complexity of wait-free implementations of concurrent ob-

jects is to reduce the analysis of concurrent objects to the analysis of the strong renaming decision problem.

For example, an *ordered set* S is a data structure whose value is some subset of a totally ordered set T, with an *insert*(a) operation that adds $a \in T$ to S and a *remove* operation that removes minimum element $a \in S$ from S and returns a. Many interesting concurrent objects such as stacks, queues, and heaps are special cases of ordered sets. Because we can solve the strong renaming problem given any wait-free implementation of an ordered set, any lower bound on the renaming problem implies a lower bound on the *remove* operation of an ordered set:

Proposition 1: If $\Omega(f(m))$ rounds of communication are required by any protocol for strong renaming, where m is the number of participating processors, then $\Omega(f(c))$ rounds of communication are required by any wait-free implementation of the ordered set's *remove* operation, where c is the number of concurrent invocations of the *remove* operation.

Proof: Let S be a wait-free implementation of an ordered set, and let g an initial global state for S in a system with n processors $p_1, \ldots, p_n$. Without loss of generality, suppose the value of S is a subset of the integers. Consider the sequential execution of the operations *insert*$(1), \ldots,$ *insert*(n) starting in the global state g. This execution brings the system to a global state h in which the ordered set contains the values 1 through n. Any concurrent execution of m *remove* operations is guaranteed to return the values 1 through m in some order. Thus, a simple strong renaming algorithm takes h as its initial state and has each participating processor perform a *remove* operation and output the integer returned as its name. If the *remove* operation can be implemented in $f(c)$ time, where c is the number of concurrent processors, then the strong renaming problem can be solved in $f(m)$ time, where m is the number of participating processors. Consequently, any $\Omega(f(m))$ lower bound for strong renaming implies an $\Omega(f(c))$ lower bound for an ordered set's *remove* operation. □

Since an ordered set's *remove* operation is just a special case of a stack's *pop*, a queue's *dequeue*, and a heap's *min*, a lower bound for strong renaming implies a lower bound for each of these operations as well.

As another example, consider the increment register. An *increment register* is just a special case of a fetch&add register. The value of an increment register is just an integer, initially 0. The increment register provides an *increment* operation that atomically increments the value of the register and returns this new value. Again, a lower bound for renaming implies a lower bound for an increment register's *increment* operation:

Proposition 2: If $\Omega(f(m))$ rounds of communication are required by any protocol for strong renaming, where m is the number of participating processors, then $\Omega(f(c))$ rounds of communication are required by any wait-free implementation of the increment register's *increment* operation, where c is the number of concurrent invocations of the *increment* operation.

Proof: Let R be a wait-free implementation of an increment register, and let g be an initial global states for R in a system with n processors $p_1, \ldots, p_n$. Recall that R's initial value is 0 in g. Any concurrent execution of m *increment* operations is guaranteed to return the values 1 through m. Thus, a simple strong renaming algorithm takes g as its initial state and has each participating processor perform an *increment* operation and output the returned value as its name. □

These results show that we can prove lower bounds on the complexity of wait-free implementations of several interesting concurrent objects by proving lower bounds on the strong renaming decision problem. We note that similar results hold in many other models, such as asynchronous systems, and hence that lower bounds for strong renaming in these models also imply lower bounds for wait-free implementations of concurrent objects. These reductions can also be used to prove impossibility results. For example, since it follows from the results of [1, 2] that the strong renaming problem cannot be solved in asynchronous systems, we see that there is no wait-free implementation of an ordered set in such systems. Such results also follow from [14] where reductions to consensus are also used to prove several impossibility results for wait-free implementations. We note that similar reduction between renaming and other decision problems such as ℓ-assignment and order-preserving renaming are also possible, meaning that lower bounds on strong renaming imply lower bounds on these decision problems as well.

5 Renaming Lower Bound

In this section, we prove a lower bound of $\Omega(\log m)$ on the number of rounds of communication required to solve the strong renaming problem for a reasonably general class of protocols. We restrict our attention to *comparison-based* protocols in which we assume the existence of a totally ordered set of processor ids and we assume that the only operations a processor can perform on processor ids is to test for equality and order; that is, given two ids p and q, a processor can

test for $p = q$, $p \leq q$, and $p \geq q$, but cannot examine the structure of the ids in any more detail. The restriction to comparison-based protocols is a reasonable one to make in systems where there are many more processor ids than there are processors. In such systems, there may be no effective way to enumerate all possible processor ids, and no way to tell whether there exists a processor with an id between two other ids. Furthermore, since there are so many possible ids, it is not feasible to use processor ids as indices into data structures as is frequently done in wait-free implementations of objects like atomic registers (cf. [23]).

In order to make this restriction precise, it is convenient to assume that all protocols are of a particular form. Following Frederickson and Lynch [12], we assume without loss of generality that each processor's local state consists of its id, its initial state, and the entire history of messages it has received; and we assume that processors simply broadcast their entire local state in every round. We represent a processor's local state with a LISP S-expression. The initial state for a processor p in initial state s is written (p, s), and later states are written $(p\ m_0\ \ldots\ m_k)$, where $m_0, \ldots, m_k$ is the sequence of messages received in the previous round (themselves S-expressions representing the states of the sending processors), sorted by processor identifiers. By convention, p also sends to itself in each round. We note that the reason we can assume processor states have such a special representation is that from such a description of a processor's state we can reconstruct the value of every variable v appearing the the actual state. It is convenient to associate with every such variable v a *variable function* f_v mapping S-expressions to the value of v in the corresponding processor state represented by the S-expression.

Loosely speaking, two processor states s and s' are *order equivalent* if (i) they are structurally equivalent S-expressions, (ii) initial states from corresponding positions in s and s' are identical, and (ii) if two ids p and q taken from s satisfy $p \leq q$, then the two ids p' and q' taken from corresponding positions of s' satisfy $p' \leq q'$. Intuitively, two order equivalent states must look identical to any protocol that can only compare processor ids for order. More formally, a one-to-one function ϕ from a processor id p to a processor id $\phi(p)$ is *order-preserving* if $p \leq q$ implies $\phi(p) \leq \phi(q)$. Any such ϕ can be extended to processor states (S-expressions) by defining $\phi(p, s) = (\phi(p), s)$ and $\phi(p\ m_0\ \ldots\ m_k) = (\phi(p)\ \phi(m_0)\ \ldots\ \phi(m_k))$. Two processor states are *order equivalent* if there exists an order-preserving function ϕ mapping one to the other, and *order inequivalent* otherwise. A protocol is a *comparison-based* protocol if the value of variable functions are identical in order-equivalent states. In the context of the renaming protocol, for example, this means that the value of the output register is identical in order-equivalent states.

Given this restriction, a useful technique for proving lower bounds is to prove that we can keep all processors in order-equivalent states for a long time. In order to do so, we make use of the *sandwich* failure pattern. Given processors $p_1, \ldots, p_n$, suppose $n = 3m + 1$ for some m (the sandwich failure pattern can immediately fail one or two processors with the lowest ids at the beginning of the round and pretend they don't exist). The sandwich failure pattern causes the m processors $p_1, \ldots, p_m$ with the lowest ids and the m processors $p_{2m+2}, \ldots, p_{3m+1}$ with the highest ids to fail in the following way: each processor $p_{m+j} \in \{p_{m+1}, \ldots, p_{2m+1}\}$ receives messages only from processors $p_j, \ldots, p_{2m+j}$. Notice that each such processor p_{m+j} sees $2m + 1$ active processors, and sees its rank in this set of active processors as $m + 1$. Notice also that the active processors after a round of the sandwich failure pattern is always a consecutive subsequence of processors from the middle of the sequence of active processors at the beginning of the round. Using this failure pattern, we can prove the following:

Proposition 3: Given a system of n processors in the same (initial) state, it is impossible to force these processors into order-inequivalent states in fewer than $\log_3(n)$ rounds.

Proof: Notice that the sandwich failure pattern fails roughly 2/3 of the active processors, leaving roughly 1/3 remaining active in the next round. We claim that if $\ell \leq \log_3(n)$, then after ℓ rounds of the sandwich failure pattern the states s_i and s_j of processors p_i and p_j are related by the order-preserving function ϕ_{j-i} defined by $\phi_k(p_i) = p_{i+k}$ for $1 \leq i \leq n - k$ when $k \geq 0$ and for $k + 1 \leq n$ when $k < 0$; that is, $\phi_{j-i}(s_i) = s_j$. We proceed by induction on ℓ.

The result is immediate for $\ell = 0$ since each p_i's initial state is (p_i, s), and $\phi_{j-i}(p_i, s) = (\phi_{j-i}(p_i), s) = (p_j, s)$. For $\ell > 0$, suppose the hypothesis is true for $\ell - 1$. Suppose there are $3m + 1$ active processors at the beginning of round ℓ. (Again, we can immediately fail one or two processors with the lowest ids in the next round and assume that this is true.) Since the active processors at the beginning of round ℓ are a consecutive subsequence of $p_1, \ldots, p_n$, suppose they are $p_{a+1}, \ldots, p_b$. Notice that after another round of the sandwich failure pattern, the active processors are $p_{a+m+1}, \ldots, p_{b-m}$, and that state of processor p_{a+m+i} is

$$(p_{a+m+i}\ (p_{a+i}\ s_{a+i})\ \ldots\ (p_{a+2m+i}\ s_{a+2m+i}))$$

and the state of processor p_{a+m+j} is

$$(p_{a+m+j}\ (p_{a+j}\ s_{a+j})\ \ldots\ (p_{a+2m+j}\ s_{a+2m+j}))$$

It is easy to see that ϕ_{j-i} maps the state of p_{a+m+i} to p_{a+m+j}, as desired. □

Since the participating processors start a renaming protocol in order equivalent states (their initial states consist of their processor id, a 1 in their input register, and nothing in their output register), this proposition gives us the desired lower bound for strong renaming:

Corollary 4: Any comparison-based protocol for strong renaming requires $\Omega(\log_3 m)$ rounds of communication, where m is the number of participating processors.

As shown in the previous section, this result yields a $\Omega(\log_3 m)$ lower bound on a variety of decision problems and concurrent objects in the comparison model, where m is the number of concurrent processors. It is not known whether the same lower bound holds also for models in which processor identifiers have a richer structure. We now show that the technique of showing that processor states can be forced to remain order equivalent cannot be used to derive stronger bounds. If operations of objects such as stacks or queues require more than a logarithmic number of rounds, then this lemma strongly suggests that the additional cost cannot be an artifact of the comparison model, but is somehow inherent in the semantics of the objects themselves.

Proposition 5: There exists a protocol that leaves processors $p_1, \ldots, p_n$ in order-inequivalent states after $O(\log n)$ rounds.

Proof: We give a simple comparison-based algorithm in which processors choose distinct sequences of integers after a logarithmic number of rounds. In each round, each processor broadcasts its identifier and the sequence of integers constructed so far. Two processors *collide* in a round if they broadcast identical sequences. In round 1, processor p broadcasts (p) (and hence all processors collide with the empty sequence). Let $(p\ i_1\ \ldots\ i_{k-1})$ be the message p broadcast at round $k-1$, and i_k the number of processors less than p that collide with p at round $k-1$. In round k, p broadcasts $(p\ i_1\ \ldots\ i_k)$. Each processor halts when it does not collide with any other processor.

We claim that the set of processors that collide with a particular processor must shrink by approximately half at each round, yielding an $O(\log n)$ running time. Two processors that broadcast different sequences continue to do so, so the set of processors that collide with p at round k is a subset of the processor that collided with p at earlier rounds. Consider a set of ℓ nonfaulty processors that collide at round k, let p be the least processor in that set, and let q be the highest. Because q must count at least ℓ colliding processors with lower ids, and because p and q collide at round k, processor p must also see ℓ colliding processors with lower ids. Since p is minimal, however, the processors it counts must have failed before sending to q, hence at least $2\ell-1$ processors collided with p and q in round $k-1$, and at least $\ell-1$ have failed. □

6 Optimal Renaming

We note that the proof of the $\log m$ lower bound for renaming depends on the fact that the renaming problem requires two processors to do different things, which is impossible in order-equivalent states in the comparison model. The proof actually applies to many other situations in which processors are required to behave in nontrivially different ways. Because this proof scheme does not make heavy use of the semantics of the problems under consideration, it seems possible that the $\log m$ lower bound is so cheap that every decision problem or wait-free implementation requires significantly more time. In order to show that this is a meaningful lower bound in the comparison model, we now give a $\log m$ algorithm for strong renaming.

The algorithm itself is given in Figure 1. The basic idea is that if a processor p hears of 2^b other participating processors, then it chooses a b-bit name for itself one bit at a time, starting with the high-order bit and working down to the low-order bit. Every round, p sends an interval I containing the names it is willing to choose from. On the first round, when the processor has not yet chosen any of the leading bits in its name, it sends the entire interval $[1, 2^b]$. It sets its high-order bit to 0 if it finds it is in the bottom half of the set of processors it hears from interested in names from the interval $[1, 2^b]$, and to 1 if it finds itself in the top half. In the first case it sends the interval $[1, 2^{b-1}]$, and in the second it sends $[2^{b-1}+1, 2^b]$. In order to make an accurate prediction of the behavior of processors interested in names in its interval I, however, it must wait until every processor interested in names in I is interested *only* in names in I before choosing its bit and splitting its interval in half; that is, it must wait until its interval I is maximal among the intervals intersecting I. Continuing in this way, the processor chooses each bit in its name, and continues broadcasting its name until all processors have chosen their name.

define $rank(s, S) = |\{s' \in S : s' \leq s\}|$
define $bot(S) = \{s \in S : rank(s, S) \leq |S|/2\}$
define $top(S) = S - bot(S)$
define $bot(S, k) = \{s' \in S : rank(s', S) \leq 2^b$
where $1 \leq 2^b < k \leq 2^{b+1}\}$

broadcast p
$\mathcal{P} \leftarrow \{p' : p' \text{ received}\}$
$m \leftarrow \lceil \log |\mathcal{P}| \rceil$
$I \leftarrow [1, 2^b]$

repeat
  broadcast (p, I)
  $\mathcal{I} \leftarrow \{I' : (p', I') \text{ received and } I \cap I' \neq \emptyset\}$
  $\mathcal{P} \leftarrow \{p' : (p', I') \text{ received and } I \cap I' \neq \emptyset\}$
  if $I' \subseteq I$ for every $I' \in \mathcal{I}$ then
    if $p \in bot(\mathcal{P}, |I|)$
      then $I \leftarrow bot(I)$
      else $I \leftarrow top(I)$
until $|I'| = 1$ for all $I' \in \{I' : (p', I') \text{ received}\}$

return a, where $I = [a, a]$

Figure 1: A $\log m$ renaming protocol $\mathcal{A}$.

There are a few useful observations about the intervals processors send during this algorithm. The first is that if processor p sends the interval I_k during round k, then $I_k \supseteq I_{k'}$ for all $k' \geq k$. The second is that each interval I_k is of a very particular form; namely, every interval sent during an execution of $\mathcal{A}$ is of the form $[c2^j + 1, c2^j + 2^j]$ for some constant c. This is easy to see since the first interval I_1 a processor sends is of the form $[1, 2^b]$, and every succeeding interval I_k is of the form $top(I_{k-1})$ or $bot(I_{k-1})$. We say that an interval I is a *well-formed* interval if it is of the form $[c2^j + 1, c2^j + 2^j]$ for some constant c. It is easy to see that any two well-formed intervals are either distinct, or one is contained in the other. Notice that every well-formed interval I is properly contained in a unique minimal, well-formed interval $\hat{I} \supset I$. Furthermore, either $I = top(\hat{I})$ or $I = bot(\hat{I})$, and it is the low-order bit of the constant c that tells us which is the case. We define the operator $\hat{I}$ that maps a well-formed interval I to the unique minimal, well-formed interval $\hat{I}$ properly containing I

In every round of the algorithm, a processor p computes the set $\mathcal{P}$ of processors with intervals I' intersecting its current interval I. These processors in $\mathcal{P}$ are the processors p is competing with for names in I. When p sees that its interval I is a maximal interval (that is, all intervals I' received that intersect I are actually contained in I), processor p adjusts its set I to either $bot(I)$ or $top(I)$. Our first lemma essentially says that when p replaces I with $bot(I)$ or $top(I)$, there are enough names in I to assign a name from I to every competing processor. Furthermore, this lemma shows that when a processor's interval reduces to a singleton set, then this processor no longer has any competitors for that name.

Lemma 6: Suppose p sends interval I during round $k \geq 2$. If P is the set of processors sending an interval $I' \subseteq I$ during round k, then $|P| \leq |I|$.

Proof: We consider two cases: $I = bot(\hat{I})$ and $I = top(\hat{I})$.

First, suppose $I = bot(\hat{I})$. Consider the greatest processor q (possibly p itself) in P. Processor q sent some interval $J \subseteq I$ to p in round k, so consider the first round $\ell \leq k$ that q sent some interval $J \subseteq I$ to any processor (and hence to p) in round ℓ.

If $\ell = 2$, then J is of the form $[1, 2^b]$, where 2^b is an upper bound on the number of processors q heard from in round 1, and hence on the number of active processors in round k, and therefore on $|P|$, the number of processors sending intervals contained in I in round k. It follows that $|P| \leq 2^b = |J| \leq |I|$.

If $\ell > 2$, then q sent $J \subseteq I$ in round ℓ, and q sent a larger interval $\hat{J} \not\subseteq I$ in round $\ell - 1$. In fact, we must have $J = I$ and $\hat{J} = \hat{I}$, for if $J \subset I$ then $\hat{J} \subseteq I$ and ℓ is not the first round that q sent an interval contained in I. Let $\mathcal{P}$ be the set of processors sending an interval intersecting $\hat{I}$ to q in round $\ell - 1$. Since every processor $p' \in P$ sending an interval $I' \subseteq I$ to p in round k must also send an interval intersecting $\hat{I}$ to q in round $\ell - 1$, each of these processors must be contained in $\mathcal{P}$, and hence $P \subseteq \mathcal{P}$. Since q sent $\hat{I}$ in round $\ell - 1$ and $I = bot(\hat{I})$ in round ℓ, it must be the case that $q \in bot(\mathcal{P}, |\hat{I}|)$ at the end of round $\ell - 1$. Since $P \subseteq \mathcal{P}$ and since q is the greatest processor in P, it must be the case that $P \subseteq bot(\mathcal{P}, |\hat{I}|)$. It follows that $|P| \leq |\hat{I}|/2 = |I|$, as desired.

Now, second, suppose $I = top(\hat{I})$. The proof in this case is similar to the proof when $I = bot(\hat{I})$, except that q is now taken to be the *least* processor in P. □

Our second lemma shows that when a processor p selects an interval $I = [a, b]$, there are enough participating processors to assign all names $1, \ldots, a$ to participating processors. In particular, when p's interval becomes the singleton set $[a, a]$, then there are at least a participating processors, and hence a is valid name for p to choose. We say that a processor *holds* an interval $[a, b]$ during a round if $[a, b]$ is its interval at the beginning of the round, and hence the the interval it sends during that round (if it sends any interval at all).

Lemma 7: If $I = [a, b]$ is a maximal interval sent by any processor during round k, then there are at least $a - 1$ processors holding intervals $[a', b']$ during round k with $b' < a$.

Proof: By induction on k.

Suppose $k = 2$, the first round processors send any interval. Then I is of the form $[1, 2^b]$, and it is vacuously true that at least 0 processors hold intervals $[a', b']$ during round 2 with $b' < 1$.

Suppose $k > 2$, and the induction hypothesis holds for $k' < k$. Suppose $I = bot(\hat{I})$. Since I is a maximal interval sent in round k, either I or $\hat{I}$ is a maximal interval sent in round $k - 1$. Since intervals I and $\hat{I}$ have the same lower bound a, the induction hypothesis for $k - 1$ says that there are at least $a - 1$ processors who hold intervals $[a', b']$ during round $k - 1$ with $b' < a$. Since each of these processors holding an interval $[a', b']$ in round $k - 1$ must hold an interval $[a'', b'']$ contained in $[a', b']$ in round k, it follows that there are at least $a - 1$ processors holding intervals $[a'', b'']$ in round k with $b'' \leq b' < a$.

Suppose $I = top(\hat{I})$. Let p be the smallest processor ever sending I during the execution. If I is the interval p sent in round 2, then we are done, using the argument for the case of $k = 2$. Suppose, therefore, that I is not the interval sent by p in round 2. It follows that p must have sent the interval $\hat{I} = [\hat{a}, \hat{b}]$ in some round $k' < k$, and then sent the interval $I = top(\hat{I})$ in round $k' + 1$. Since p changed intervals between round k' and $k' + 1$, the interval $\hat{I}$ must have been a maximal interval in round k', and hence by the induction hypothesis at least $\hat{a} - 1$ processors hold intervals $[a', b']$ during round k' with $b' < \hat{a}$. Notice also that since p changed its interval from $\hat{I}$ to $I = top(\hat{I})$, at least $a - \hat{a} = |\hat{I}|/2$ of the processors p' sending an interval to p that intersected $\hat{I}$ satisfied $p' < p$. Furthermore, since $\hat{I}$ was a maximal interval, each of these intersecting intervals I' was contained in $\hat{I} = [\hat{a}, \hat{b}]$, and hence each of these processors p' sending an intersecting interval could not have been one of the $\hat{a} - 1$ processors sending an interval $[a', b']$ in round k' with $b' < \hat{a}$. Finally, since p is by assumption the smallest processor sending $I = top(\hat{I})$, and since any processor $p' < p$ sending an interval contained in $\hat{I}$ must continue sending intervals contained in $\hat{I}$, each of these processors must eventually send intervals contained in $bot(\hat{I}) = [\hat{a}, a - 1]$. Consequently, when I finally becomes a maximal interval (by round k at the latest), each of these processors is holding an interval contained in $bot(\hat{I})$, and so are at least $\hat{a} - 1 + (a - \hat{a}) = a - 1$ processors hold intervals $[a', b']$ in that round with $b' < a$. Since the interval held by these processors in round k is contained in the interval they hold in this round, the same is true in round k. □

Finally, since the algorithm terminates when every processor's interval is a singleton set, and since the size of the maximal interval sent during a round decreases by a factor of 2 every round, it is easy to prove that the algorithm $\mathcal{A}$ terminates in $\log m$ rounds.

Lemma 8: The algorithm $\mathcal{A}$ terminates in $\log m + 2$ rounds, where m is the number of participating processors.

Proof: Consider an arbitrary execution of $\mathcal{A}$. For each round k, let ℓ_k be the least upper bound on the size of the intervals I sent during round k. In round 2, each processor sends an interval of the form $[1, 2^b]$ where 2^b is the least power of two greater than the number of processors that processor heard from in round 1. It follows that $k_2 = 2^b \leq 2m$ for some m. For any round $\ell > 2$, any processor p sending an interval I of maximum size $k_{\ell-1}$ in round $\ell - 1$ sends one of the intervals $top(I)$ or $bot(I)$ in round ℓ, since all (well-formed) intervals it receives in round $\ell - 1$ intersecting I are actually contained in I. It follows that $k_\ell = k_{\ell-1}/2$ for any round $\ell > 2$, and hence that $k_{\log(2m)+1} \leq 2^b/(2m) \leq 1$. Thus, within $\log(2m) + 1 = \log(m) + 2$ rounds, all intervals sent are of size 1, and hence all processors terminate. □

With these results, we are done:

Theorem 9: The algorithm $\mathcal{A}$ solves the strong renaming problem, and terminates in $\log(m)+2$ rounds, where n is the number of participating processors.

Proof: First, all processors choose a name, since Lemma 8 says that the algorithm terminates in $\log(m)+2$ rounds, where m is the number of participating processors.

Second, the names chosen by processors are distinct. Suppose two processors p and p' chose the name a at the end of rounds k and $k' \geq k$, respectively. Processors p and p' must have sent the singleton set $I = [a, a]$ to all processors in rounds k and k', and intervals containing I in all preceding rounds. Since p could not have terminated in round k unless all intervals it received were singletons, both processors must have sent $I = [a, a]$ in round k, and this interval must have been a maximal interval (all intervals were singletons). It follows by Lemma 6 that $2 \leq |I| = 1$, which is impossible.

Finally, names chosen are in the interval $[1, m]$, where m is the number of participating processors. Consider the processor p choosing the highest name a chosen by any processor, and consider the last round k in which p send the singleton set $I = [a, a]$. The

```
suggestion(e, ℓ, R)
    winner[] ← a vector of max-entry
    for each (e', a', ℓ', E') ∈ R
        winner[a'] ← min{winner[a'], e'}
    for each (e', a', ℓ', E') ∈ R
        if winner[a'] = e' then value[e'] ← a'
    for each (e', a', ℓ', E') ∈ R by increasing e'
        if e' ∉ winner then
            value[e'] ← min{b > ℓ : b ∉ value}
    return value[e]

increment()
    wait for an even round k, then
        ℓ ← |{e' ∈ E : gen(e') < k}|
        e ← ⟨k, p⟩
        E ← E ∪ {e}
        broadcast (e, ℓ + 1, ℓ, E)
        receive (e', a', ℓ', E')
        repeat
            E ← union of the E' received
            R ← {(e', a', ℓ', E') : (e', a', ℓ', E')
                                received and gen(e') = k}
            ℓ ← min{ℓ' : (e', a', ℓ', E') ∈ R}
            a ← suggestion(e, ℓ, R)
            broadcast (e, a, ℓ, E)
            receive (e', a', ℓ', E')
        until generation k suggestions don't change
        return a
```

Figure 2: An increment register algorithm $\mathcal{R}$.

interval I must have been a maximal interval in round k, or p would have sent I in round $k+1$ as well. It follows by Lemma 7 that at least $a-1$ processors hold intervals $[a', b']$ with $b' < a$ in round k, and hence that a is at most the number m of participating processors. □

7 A Wait-free Increment Register

In this section, we show how a particular strong renaming algorithm can be transformed into a wait-free implementation of an *increment register*. Our implementation has the property that each invocation of the *increment* operation terminates in $O(c^{\frac{1}{2}+\epsilon})$ rounds, where c is the number of processors invoking the operation in the same round and ϵ is any positive real value.

The increment register algorithm $\mathcal{R}$ appears in Figure 2. The basic idea behind this algorithm is that every participating processor p repeatedly computes a value a and suggests to the rest of the processors that it be allowed to return this value a. Processor p makes this suggestion by broadcasting a to the other processors. It then collects all suggestions broadcast to it and looks to see how many other processors have suggested a for themselves, too. If p sees that it is the least processor suggesting a, then p chooses a for itself (we say that p is the *winner*). Otherwise, p recomputes a new value and tries again. In this sense, the protocol is similar to the uniqueness renaming protocol in [2].

One subtle aspect of this protocol is the way in which processors compute their suggestions. Every processor maintains a lower bound ℓ, and takes care that the value it suggests for itself is always above this lower bound. In addition to broadcasting its suggestion, it broadcasts its lower bound, and resets its lower bound to be the minimum of the lower bounds received from concurrent processors. The need for establishing a lower bound is to ensure linearizability; the need to repeatedly lower the lower bound is to ensure a reasonable time complexity. To compute its preference, a processor p first computes the winners for the various values using the suggestions it has received, and assigns these values to the winners. For the remaining processors, p assigns values to them one by one in increasing processor order by assigning to q the least unassigned value above p's lower bound ℓ. (We write $b \notin value$ to denote that b does not appear in the vector *value*.) The name it assigns to itself is its suggestion for the next round. We note that once a processor wins the value a, it will win the value a in every later round, but it must protect its victory by continuing to participate in the protocol until all other concurrent processors have successfully chosen a value for themselves, too.

Because a processor can invoke the *increment* operation on several different rounds, invocations are tagged with both the invoking processor's id and the round in which the operation was invoked. This tag is called an *entry*, which is a pair $\langle k, p\rangle$ consisting of a round number k and a processor id p. Entries are ordered lexicographically by round number and processor id, although a distinguished value *max-entry* is considered greater than every other entry. We say that the *generation* of the entry $\langle k, p\rangle$ is k. We write $gen(\langle k, p\rangle) = k$, and refer to p as a generation k processor. We also refer to an invocation tagged with $\langle k, p\rangle$ as an invocation by p, or as a generation k invocation.

Every processor maintains a set E containing all entries of which it has ever heard (initially, E is empty). It broadcasts this set[1] every round, and merges the

[1] In practice, it need only broadcast the additions to E since

sets it receives by taking their union. (By convention, a processor always sends to itself.) One important point not apparent in Figure 2 is that even if a processor p is not currently invoking the *increment* operation, it must eavesdrop on broadcasts by other processors p' who are invoking the *increment* operation and merge the sets E' they send with its own. This is necessary in order for p to have enough information to be able to invoke the operation itself at some later time. In particular, p initially sets its lower bound to the number of entries it has heard of from earlier generations, so E must be kept up to date. We note that invocations of the *increment* operation always begin on even rounds in order that earlier generations have enough time to relay all relevant entries seen to later generations before the later generations begin.

7.1 Correctness

The intuition behind a processor's setting its lower bound ℓ to the number of processors from previous generations is that there are already enough processors to account for the values 1 through ℓ. The correctness of the algorithm (in particular, linearizability) depends primarily on the fact that the lower bound is high enough to guarantee that processors from earlier generations never suggest (or choose) values above this lower bound. The following result shows this is true for the initially computed lower bound.

Lemma 10: If ℓ is the initial lower bound computed by a generation k processor, then every generation $k' < k$ processor surviving its second round suggests only values $a \leq \ell$.

Proof: Suppose $\ell = \ell_p$ is the initial lower bound computed by a generation k processor p, and suppose E_p is the set of entries p used to compute ℓ_p. Consider any generation $k' < k$ processor q surviving its second round, and let ℓ_q be its initial lower bound, computed using E_q. Let l and u be the number of entries $e' \in E_p$ with $gen(e') < k'$ and $gen(e') \leq k'$, respectively. Since q survived its second round, it sent E_q to p at the end of its first round, so $E_q \subseteq E_p$, and hence $\ell_q \leq l \leq u \leq \ell_p$.

Suppose q suggested the name $\ell'_q + i$ during its round $j \geq 2$, where $\ell'_q \leq \ell_q$ is its lower bound during its jth round. Then q heard from at least i generation k' processors in its jth round, so it heard from at least i such processors in its first round, at which point it added these processors to its set E_q and sent them to p in its second round. It follows that these i processors are contained in E_p, so the difference $u - l$ must be at least i, and hence $\ell'_q + i \leq \ell_q + i \leq l + i \leq u \leq \ell_p$. Thus, q suggests only values $a \leq \ell_p$, as desired. □

The next result shows that the successive lowering of lower bounds is a safe thing to do.

Lemma 11: If ℓ is any (not necessarily initial) lower bound held by a generation k processor, then every nonfaulty generation $k' < k$ processor suggests only values $a \leq \ell$.

Proof: Let $\ell_1, \ldots, \ell_d$ be the initial lower bounds computed by the generation k processors $p_1, \ldots, p_d$. Let q be a nonfaulty generation $k' < k$ processor. Since q survived its second round, Lemma 10 says that q suggests only names $a \leq \ell_i$ for every i, and thus $a \leq \ell_{\min}$ where $\ell_{\min} = \min\{\ell_1, \ldots, \ell_d\}$. Since ℓ was computed by repeatedly taking the min of lower bounds from processors of the same generation, it is the result of taking the min of some subset of $\ell_1, \ldots, \ell_d$. Thus, we have $\ell \geq \ell_{\min}$, and hence $a \leq \ell$ as desired. □

We have mentioned that our implementation of an increment register is a transformation of a simple strong renaming algorithm. The next result shows that this implementation can still be viewed as a strong renaming protocol.

Theorem 12: $\mathcal{R}$ solves the strong renaming problem.

Proof: First, notice that a nonfaulty processor p with entry e eventually chooses a value. Since only processors with lower entries from the same generation can keep p from committing on a value, and since the processor with the lowest entry from that generation in a given round either fails or commits, the number of processors that can keep p from committing decreases by at least one every round until p finally chooses a value.

Second, notice that no two nonfaulty processors choose the same value. Suppose to the contrary that p and q both choose a. Since processors always suggest names above their current lower bound, Lemma 11 implies that processors p and q must be from the same generation. Furthermore, p and q must have suggested a in the same round, or the later processor would have seen the earlier processor's suggestion and not suggested a for itself. If p and q suggest a in the same round, however, the larger will observe the smaller and will not choose a.

Finally, notice that if a processor p suggests a (and, in particular, chooses a), then there exist at least a processors; and hence that every processor chooses a value from the set $\{1, \ldots, m\}$, where m is

its last broadcast.

the number of participating processors. To see this, let ℓ_p be the lower bound computed by p using E_p, and suppose p has suggested the value $\ell + i$ in its jth round, where $\ell \leq \ell_p$ is its lower bound during its jth round. Then p's set E_p contained $\ell_p \geq \ell$ processors from earlier generations, and p heard from i processors from its own generation in its $j-1$st round, so there are at least $\ell + i$ processors. □

The desired result, however, is an implementation of an increment register:

Theorem 13: $\mathcal{R}$ is a linearizable, wait-free implementation of an increment register.

Proof: The proof of Theorem 12 shows that $\mathcal{R}$ is an implementation of an increment register: distinct invocations of the *increment* operation return distinct values, and each value less than any value returned is "accounted for" by other invocations, perhaps ones that have halted. To show linearizability, we must show that operations appear to take effect in "real-time" order: if one operation begins after another has finished, then the former returns a larger value. But this is obvious, since Lemma 11 implies that invocations by processors from one generation always return higher values than invocations by nonfaulty processors from previous generations. □

7.2 Time Complexity

We now analyze the running time of an invocation of the *increment* operation. For the rest of this section, fix a processor p invoking the *increment* operation in round r (generation r). Define the *concurrency set* to be the number of other processors invoking the *increment* operation in the same round, and let c be the size of this set. All processors and invocations will be of generation r unless explicitly stated otherwise.

A processor is *committed* at time k if it is a winner at time k (that is, if it can choose its suggested value), and we say that it has been *bounced* (by a lower processor) if it is not. A processor is *active* at time k if it has not committed at time k (it has bounced) and sends at least one message in round $k+1$. An active processor's *suggestion* at time k is the suggestion it sends in round $k+1$. A *collision* at time k is a set of active processors with the same suggestion at time k. Notice that at most one nonfaulty processor in a collision at time k will be able to commit at time $k+1$. Our analysis is based on discovering the most efficient way of maintaining large collision sets, and hence of keeping processors uncommitted.

We note that using a failure pattern somewhat similar to the sandwich failure pattern, it is possible to keep an invocation of the *increment* operation started at time 0 running for roughly $\sqrt{c}$ rounds. At time 0, a processor's lower bound is necessarily 0, so the failure pattern's strategy is to keep processors uncertain about the number of lower processors starting at time 0.

In general, though, differences in the lower bounds held by various processors can also cause large collisions. These differences are initially the result of failures by processors in very early generations, but we want the complexity of the algorithm to depend only on failures within the current generation. It is for this reason that processors adjust their lower bounds every round by taking the min of the lower bounds received from processors in their own generation. Now disagreement about the lower bounds must be the result of a failure in the current generation. Of course, now collisions can occur if processors change their lower bounds at unexpected times, and analyzing collisions when this happens is quite difficult. Fortunately, the following result shows that a processor's lower bound cannot change too many times.

Lemma 14: Processor p's lower bound can change at most $2\sqrt{c}$ times.

Proof: Consider the lower bounds $\ell_1 > \ell_2 \cdots > \ell_d$, and suppose that p's lower bound is lowered to ℓ_j for the first time after its k_jth round, for $j = 1, \ldots, d$.

It is clear that for each ℓ_j there must be a sequence $\sigma_j = q_{j,1}, \ldots, q_{j,k_j}$ of processors of p's generation such that (i) $q_{j,1}$'s initial lower bound is ℓ_j, (ii) every $q_{j,i}$ sent to $q_{j,i+1}$ in round i but failed to send to p, and (iii) q_{j,k_j} finally sent to p in round k_j. This is because ℓ_j must be the initial lower bound of some processor, and because p would have learned about ℓ_j and hence lowered its lower bound to ℓ_j before the end of its k_jth round if any $q_{j,i}$ with $i < k_j$ had not failed to send to p. Furthermore, no processor can appear in two sequences σ_j and $\sigma_{j'}$, with the possible exception that the final processor in the two sequences may be the same. To see this, suppose to the contrary that $q_{j,i} = q = q_{j',i'}$ where $i < k_j$ and $i' < k_{j'}$. Suppose further that $\ell_j > \ell_{j'}$, which implies that $k_j < k_{j'}$. We must have $i = i'$ since q cannot fail to p in two separate rounds. Since q's lower bound in this round is necessarily $\ell_{j'} = \min\{\ell_j, \ell_{j'}\}$, processor p learns about this lower bound after round k_j and sets its lower bound to $\ell_{j'} < \ell_j$ after round $k_j < k_{j'}$, a contradiction.

It follows that the number of distinct processors of the form $q_{j,i}$ must be at least

$$(k_1 - 1) + (k_2 - 1) + \cdots + (k_d - 1) \geq 0 + \cdots + (d-1) \geq d^2/4.$$

Since these processors are of p's generation, and since there are c such processors, we have $d^2/4 \leq c$ and hence $d \leq 2\sqrt{c}$. □

Even though analyzing the number of active processors remaining after a round in which p's lower bound changes is difficult, Lemma 14 says that we can essentially ignore these rounds since p's lower bound changes only $2\sqrt{c}$ times. Since the adversary scheduling processor failures is trying to keep as many processors active as possible, giving the adversary the benefit of the doubt, we can assume that the number of active processors does not decrease at all during such a round, knowing that the resulting running time will be at most $2\sqrt{c}$ longer than the actual running time.

With this motivation, we now construct a function $A(c, f_1, \ldots, f_k)$ giving an upper bound on the number of processors active after k rounds in which p's lower bound remains steady and f_i processors fail in the ith such round. Since this function is an upper bound on the number of active processors, if its value is less than 1, then all processors have terminated within the first k rounds. We then use calculus to show that this function is maximized by taking $f_1, \ldots, f_k$ to be roughly $\sqrt{c}-1, \ldots, \sqrt{c}-k$, and yet the value of the function is still less than 1 when k is roughly $\sqrt{c}$. Consequently, regardless of how many processors fail in each round, all invocations terminate within roughly $\sqrt{c}$ rounds.

We say that round k is a *good round* for p if p has the same lower bound ℓ at times $k-2$, $k-1$, k, and $k+1$, and a *bad round* otherwise. We define good and bad rounds only for rounds of the form $k = 4i$. Since p broadcasts its lower bound ℓ in round $k-1$, everyone has a lower bound of at most ℓ at time $k-1$. Some set B_k of "bad" processors may have lower bounds different from ℓ at time $k-1$, but they do not send to p in round k and hence must fail. Some set B'_k of processors who do not fail in round k may hear from processors in B_k in round k, and hence have lower bounds different from ℓ at time k, but they do not send to p in round $k+1$ and hence must also fail. Let the set G_k of "good" processors be the complement of B'_k in the set of processors surviving to time k. Notice that G_k and B'_k partition the processors surviving to time k into two sets, those who receive only ℓ as lower bounds from other processors, and those who do not.

As the following result shows, the adversary must fail a large number of processors in order to construct large collisions in good rounds.

Lemma 15: Suppose k is a good round for p. If f processors fail round in k, then collisions at time k contain at most $f+1$ processors from G_k.

Proof: Suppose C is a collision at time k in an execution e containing $f+2$ processors from G_k. Consider the execution g differing from e only in that every processor sending a message to at least one processor in G_k in round k of e sends messages to every processor in round k of g. (Notice that this does not change the messages processors in B_k send, and hence does not change the lower bounds processors in G_k receive.) Since each processor $q \in G_k$ in the collision C is (by definition) uncommitted at time k in e, it has heard from a smaller processor in round k with the same time $k-1$ suggestion. Since the same must be true of round k in g where every processor receives even *more* messages, q is uncommitted at time k in g and computes a time k suggestion. In fact, since every processor $q \in G_k$ receives the same set of messages at time k in g, they compute the same vector w as the value of the vector *winner*, and compute unique (!) suggestions for themselves.

Given a general vector w' computed as the value of *winner*, define a *hole* in w' to be a name a such that $w'[a]$ does not contain an entry. Given two values a and b, define the *distance* between a and b in w' to be the number of holes between a and b in w'. Given two such vectors w' and w'', we say w' is a *subvector* of w'' they agree on nonempty positions in w'. Notice that the distance between a and b in w'' is at most the distance in the subvector w'.

We now consider an arbitrary processor $q \in G_k$ in the collision C, and follow the movement of its suggestion as we move through a sequence of executions from g back to e, failing one by one each of the processors $p_1, \ldots, p_d$ that fail in round k of e. Let g_i be the run differing from g only in that $p_1, \ldots, p_i$ send in round k of g_i precisely as they do in round k of e. We claim that if a_i and a are q's suggestions at time k in g_i and g, then $a_i \leq a$ and the distance between them in w is at most i. We proceed by induction on i. Since the case of $i = 0$ is vacuously true ($g_0 = g$), suppose $i > 0$ and the claim is true for $i-1$. Let w_j be the vector *winner* computed by q at time k in g_j, and notice that it is a subvector of w.

If $w_i = w_{i-1}$, then the only difference in q's computation of its suggestions a_i and a_{i-1} is that q may not have to compute a suggestion for p_i before computing its suggestion a_i in g_i. Since every processor that $q \in G_k$ hears from has the same lower bound, and since $w_i = w_{i-1}$, this means that a_i will be no higher than a_{i-1}, and that the distance between them in w_i is at most 1.

If $w_i \neq w_{i-1}$, then the only difference between the two vectors is that p_i's time $k-1$ suggestion b is now a hole in w_i. Let b^* and b_* be the holes in w_i (and hence in w_{i-1}) just above and below b, respectively. Again, the only difference in q's computation of its

suggestions a_i and a_{i-1} is that q does not have to compute a suggestion for p_i before computing a_i in g_i for itself. Again, a_i will be no higher than a_{i-1}, and that the distance between them in w_i is at most 1. Notice, however, that a_i may fall only from b^* to b in w_i instead of all the way down to b_* as it would in w_{i-1}.

In either case, $a_i \leq a_{i-1} \leq a$. Furthermore, the distance between a and a_{i-1} is at most $i-1$ in w, and the distance between a_i and a_{i-1} is at most 1 in w_i (and hence in w since w_i is a subvector of w), so the distance between a and a_i in w is at most i.

Thus, q's suggestion in g falls to its suggestion in e, moving distance at most one in w with the failure of each processor. Since processors $q \in G_k$ in C have distinct suggestions a_q in g and the same suggestion $a \leq a_q$ in e, and since there are $f+2$ processors from G_k in C, some processor $q \in G_k$ in C must have to move down at least $f+1$ holes in w from its suggestion in g to its suggestion in e. This means that at least $f+1$ processors must fail in round k, contradicting the fact that only f processors fail. □

Notice that a set of failing processors can cause the greatest number of processors to remain active by creating as few collisions as possible, since at most one processor in every collision will be able to commit. Define

$$A(c,f) = \frac{(c-f)3f}{(3f+1)}.$$

We have the following:

Lemma 16: Suppose k is a good round for p. If c processors are active at time $k-1$ and f processors fail in round k, then the number of active processors at time $k+1$ is at most $A(c,f)$.

Proof: Notice that B_k is some subset of the f processors failing in round k (since p does not hear from them at time k). Furthermore, $G_k \cup B'_k$ is the set of $c-f$ processors surviving to time k, and all processors in B'_k fail in round $k+1$ (since p does not hear from them at time $k+1$).

Suppose $|G_k| \leq (c-f)/2$. Then $|B'_k| > (c-f)/2$, and since every processor in B'_k has failed by time $k+1$, the number of active processors at time $k+1$ is at most

$$\begin{aligned}(c-f) - |B'_k| &< (c-f) - \frac{(c-f)}{2} = \frac{(c-f)}{2} \\ &\leq \frac{(c-f)3f}{(3f+1)} = A(c,f).\end{aligned}$$

On the other hand, suppose $|G_k| > (c-f)/2$. Some processors in G_k commit at time k, and the rest bounce. Let m_c and m_b be the number of processors in G_k that commit and bounce, respectively. By Lemma 15, no time k collision contains more than $f+1$ processors from G_k, so there must be at least $m_b/(f+1)$ time k collisions. Since the lowest processor in every collision is guaranteed either to fail or to commit by time $k+1$, at least $m_b/(f+1)$ of the bouncing processors are no longer active at time $k+1$. Since the same is clearly true for the m_c processors in G_k committing at time k, at least

$$\frac{m_b}{(f+1)} + m_c > \frac{m_b + m_c}{(f+1)} = \frac{|G_k|}{(f+1)} > \frac{(c-f)}{2(f+1)}$$

processors in G_k are no longer active at time $k+1$. It follows that there are at most

$$(c-f) - \frac{(c-f)}{2(f+1)} < (c-f) - \frac{(c-f)}{(3f+1)} = A(c,f)$$

active processors at time $k+1$. □

Since the concurrency set has size c, the number of active processors at the beginning of the first good round is $c_1 \leq c$. Lemma 16 says that $A(c_1, f_1)$ is an upper bound on the number of active processors after this first good round if f_1 processors fail in this round, and hence so in $A(c, f_1)$ since $A(c,f)$ is monotonic in c. Similarly, it is easy to see that at most $A(A(c,f_1),f_2)$ processors are active after two good rounds in which f_1 fail in the first and f_2 in the second. Continuing in this way, if we define

$$A(c, f_1, \ldots, f_k) = A(A(c,f_1), f_2, \ldots f_k),$$

then Lemma 16 and a simple argument by induction on k yields:

Lemma 17: If f_i processors fail in the ith good round for p, then the number of processors active after k good rounds is at most $A(c, f_1, \ldots, f_k)$.

We note that

Lemma 18: $A(c, f_1, \ldots, f_k)$ is monotonic in c.

The following result tells us that in order to keep the greatest number of processors active, we need only fail

$$a(c) = \frac{1}{3}(\sqrt{3c+1} - 1)$$

processors in a round with c active processors. Notice, by the way, that $A(c, a(c)) = 3a(c)^2$.

Lemma 19: For a fixed c, the value of $A(c,f)$ is maximized when $f = a(c)$.

Proof: The function

$$\frac{\partial A}{\partial f}(c,f) = \frac{(3f+1)(3c-6f) - 9(cf-f^2)}{(3f+1)^2}$$

has a single positive root at $a(c)$. □

Intuitively, therefore, failing $a(c)$ processors every round is the best strategy the adversary has for keeping $\mathcal{R}$ from terminating. To make this precise, define

$$\begin{aligned} A_1(c) &= A(c, a(c)) \\ A_k(c) &= A_{k-1}(A_1(c)) \end{aligned}$$

Informally, $A_k(c)$ is an upper bound on the number of active processors after following this strategy for k good rounds starting with c active processors. The following shows that the strategy of failing $a(c)$ processors every round is indeed the adversary's best strategy:

Lemma 20: $A_k(c)$ is the maximal value of $A(c, f_1, f_2, \ldots, f_k)$.

Proof: By induction on k. For $k = 1$, Lemma 19 implies $A_1(c) = A(c, a(c))$ is the maximal value of $A(c, f_1)$. For $k > 1$, assume the statement is true for $k-1$. Since $A(c, f_1) \leq A_1(c)$, and since Lemma 18 implies that $A(c, f_2, \ldots, f_k)$ is monotonic in c, we have

$$\begin{aligned} A(c, f_1, f_2, \ldots f_k) &= A(A(c, f_1), f_2, \ldots, f_k) \\ &\leq A(A_1(c), f_2, \ldots, f_k). \end{aligned}$$

Finally, the induction hypothesis implies that

$$A(A_1(c), f_2, \ldots f_k) \leq A_{k-1}(A_1(c)) = A_k(c)$$

□

Since $A_k(c)$ is the upper bound on the number of active processors after k good rounds, we can prove the following lemma showing that bounding the running time of an invocation with concurrency set of size c reduces to finding the least value of k for which $A_k(c) < 1$.

Lemma 21: If $A_k(c) < 1$, then any invocation with concurrency set of size c terminates within k good rounds.

Proof: Consider any invocation in which f_i processors fail in the ith good round, $i = 1, \ldots, k$. The protocol terminates when the number of active processors falls below one. $A(c, f_1, \ldots f_k)$ is an upper bound for the number of active processors at the end of the kth good round by Lemma 17, and $A_k(c)$ is an upper bound for $A(c, f_1, \ldots, f_k)$ by Lemma 20. □

The next three lemmas are simply technical results needed to help us find this least such k.

Lemma 22: For $0 > \alpha > 1$,

$$\lim_{c \to \infty} c^\alpha - (c-1)^\alpha = 0.$$

Proof: Let $f(c) = c^\alpha$. The derivative $f'(c) = \alpha c^{\alpha-1}$ approaches zero for large c because $\alpha < 1$. By the Mean Value Theorem of Calculus, there exists an x_c between each $c-1$ and c such that

$$f'(x_c) = f(c) - f(c-1)$$

Since the left-hand side limits to zero, so does the right-hand side. □

For the remainder of this section, let $f(c) = c^{2-\epsilon}$.

Lemma 23: For sufficiently large c,

$$f(c) < f(c-1) + \frac{2}{\sqrt{3}}\sqrt{f(c-1)}.$$

Proof: Let $\alpha = (2-\epsilon)/2$.

$$f(c) - f(c-1) = c^{2\alpha} - (c-1)^{2\alpha}$$

The last expression is equivalent to:

$$[c^\alpha - (c-1)^\alpha][c^\alpha + (c-1)^\alpha]$$

By Lemma 22,

$$\lim_{c \to \infty} c^\alpha - (c-1)^\alpha = 0.$$

In particular, there exists a C such that $c^\alpha - (c-1)^\alpha \leq 1/(4\sqrt{3})$ for $c > C$. Also,

$$\lim_{c \to \infty} c^\alpha + (c-1)^\alpha = 2(c-1)^\alpha,$$

so there exists a C' such that $c^\alpha + (c-1)^\alpha \leq 4(c-1)^\alpha$ for $c > C'$. For $c > \max(C, C')$,

$$\begin{aligned} [c^\alpha - (c-1)^\alpha][c^\alpha + (c-1)^\alpha] &\leq \frac{1}{\sqrt{3}}(c-1)^\alpha \\ &< \frac{2}{\sqrt{3}}\sqrt{f(c-1)}. \end{aligned}$$

□

Lemma 24: $A(f(c)) < f(c-1)$ for sufficiently large c.

Proof: From Lemma 23,

$$3f(c) < 3f(c-1) + 2\sqrt{3f(c-1)}.$$

Completing the square by adding one to each side,

$$3f(c) + 1 < (\sqrt{3f(c-1)} + 1)^2.$$

Taking the square root,

$$\sqrt{3f(c)+1} < \sqrt{3f(c-1)} + 1.$$

Subtracting one from each side,

$$\sqrt{3f(c)+1} - 1 < \sqrt{3f(c-1)},$$

and squaring,

$$(\sqrt{3f(c)+1}-1)^2 < 3f(c-1),$$

we have

$$A(f(c)) < f(c-1)$$

since $A(c) = 3a(c)^2$. □

Lemma 25: $A_k(f(k)) < 1$ for sufficiently large k.

Proof: By induction on k. For $k = 1$,

$$A_1(f(1)) = A_1(1) = 3a(1)^2 = \frac{1}{3}(\sqrt{4}-1)^2 < 1.$$

For $k > 1$, assume the result for $k-1$. By Lemma 24, and because A_{k-1} is monotonic:

$$A_k(f(k)) < A_{k-1}(f(k-1)).$$

The right-hand side is less than or equal to 1 by the induction hypothesis. □

Theorem 26: Fix $\epsilon > 0$. Any invocation of the *increment* operation with a concurrency set of size c terminates within $O(c^{\frac{1}{2}+\epsilon})$ rounds.

Proof: Given $\epsilon' > 0$ and sufficiently large c, Lemma 25 and the monotonicity of $A_k(c)$ imply that if $c \leq f(k) = k^{2-\epsilon'}$, then $A_k(c) \leq A_k(f(k)) < 1$. Lemma 21 therefore implies that the invocation halts in $c^{1/(2-\epsilon')} = c^{\frac{1}{2}+\epsilon}$ good rounds. Since p's lower bound can change at most $2\sqrt{c}$ times, and since good and bad rounds are defined only for rounds of the form $k = 4i$, the required number of good rounds must occur within $4(c^{\frac{1}{2}+\epsilon} + 2\sqrt{c}) = O(c^{\frac{1}{2}+\epsilon})$ rounds. □

8 Conclusions

This paper represents a first step in exploring the complexity hierarchy of wait-free concurrent objects in message-passing systems. It was previously known that any object (or decision problem) can be implemented in $O(n)$ rounds using atomic broadcast, where n is the degree of replication. Little was known, however, about whether there exist nontrivial objects or decision problems that can be implemented more efficiently. In this paper, we identify two simple but nontrivial examples: strong renaming is a decision problem that can be solved in time logarithmic in the number of active participants, and an increment register is an object that can be implemented in time approximately the square root of the number of concurrent operations. These results demonstrate that algorithms that exploit the semantics of the problem can sometimes be substantially more efficient than general-purpose algorithms. We believe that the fundamental open problem in understanding synchronous message-passing systems is to elucidate the nature of this complexity hierarchy.

The first step to understanding the complexity hierarchy is to establish lower bounds. We have proposed a general and effective technique for deriving type-specific lower bounds for concurrent objects: reduction to a decision problem. In this paper, we establish a lower bound on strong renaming, and we use this lower bound to derive lower bounds for increment registers, ordered sets, and related data types. Reduction to a decision problem is an effective technique because concurrent objects are hard to analyze directly. Unlike decision problems, in which processors start simultaneously, compute for a while, and halt with their outputs, concurrent objects have unbounded lifetimes during which they must handle an arbitrary number of operations, these operations can be invoked at any time, and the order in which operations are invoked is often important. It is an interesting open question to identify other decision problems of complexity intermediate between strong renaming and consensus that also yield lower bounds for wait-free concurrent objects.

Upper bounds have proven more difficult than lower bounds. Our approach here has been to try to convert solutions to decision problems into implementations of long-lived objects. For example, the increment register implementation given in Section 7 is based on a simple $O(m^{\frac{1}{2}+\epsilon})$ strong renaming algorithm. The resulting $O(c^{\frac{1}{2}+\epsilon})$ implementation is substantially more efficient than an $O(n)$ general-purpose algorithm using atomic broadcast, especially since the degree of concurrency c itself is typically much less than n, the total number of processors. On the other hand, this implementation is also substantially less efficient than the $\log c$ lower bound implied by the reduction to renaming. Our attempts to adapt the $\log m$ renaming algorithm were unsuccessful, primarily because we were unable to achieve linearizability — despite our best efforts, a later operation could sometimes return a lower value than an earlier operation. We leave as open questions the problem of establishing better upper bounds for increment registers, stacks, queues, and related objects.

Acknowledgements We thank Margaret Tuttle for her comments on our proofs and presentation.

References

[1] H. Attiya, A. Bar-Noy, D. Dolev, D. Koller, D. Peleg, and R. Reischuk. Achievable cases in an asyn-

chronous environment. In *Proceedings of the 28th IEEE Symposium on Foundations of Computer Science*, pages 337–346, October 1987.

[2] H. Attiya, A. Bar-Noy, D. Dolev, D. Peleg, and R. Reischuk. Renaming in an asynchronous environment. *Journal of the ACM*, July 1990.

[3] J.E. Burns and N.A. Lynch. The Byzantine firing squad problem. *Advances in Computing Research: Parallel and Distributed Computing*, 4:147–161, 1987. Available as Technical Report MIT/LCS/TM–275, MIT Laboratory for Computer Science.

[4] J.E. Burns and G.L. Peterson. The ambiguity of choosing. In *Proceedings of the 8^{th} Annual ACM Symposium on Princples of Distributed Computing*, pages 145–158, Edmonton, Alberta, August 1989.

[5] B. Coan, D. Dolev, C. Dwork, and L. Stockmeyer. The distributed firing squad problem. In *Proceedings of the 17th ACM Symposium on Theory of Computing*, pages 335–345, May 1985. Available as IBM Research Report RJ 5343, 1986.

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

[7] D. Dolev and H.R. Strong. Polynomial algorithms for multiple processor agreement. In *Proc. 14th ACM Symp. on Theory of Computing*, pages 401–407, May 1982.

[8] C. Dwork, N.A. Lynch, and L. Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM*, 35(2):288–323, April 1988.

[9] C. Dwork and Y. Moses. Knowledge and common knowledge in a Byzantine environment I: crash failures (extended abstract). In Joseph Y. Halpern, editor, *Theoretical Aspects of Reasoning about Knowledge: Proceedings of the 1986 Conference*, pages 149–170. Morgan Kaufmann, 1986. To appear in *Information and Computation*. Also available as MIT Technical Memo MIT/LCS/TM–300.

[10] 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.

[11] M.J. Fischer and N.A. Lynch. A lower bound for the time to assure interactive consistency. *Information Processing Letters*, 14(4):183–186, June 1982.

[12] G.N. Frederickson and N.A. Lynch. Electing a leader in a synchronous ring. *Journal of the ACM*, 34(1):98–115, January 1987.

[13] M.P. Herlihy. A quorum-consensus replication method for abstract data types. *ACM Transactions on Computer Systems*, 4(1), February 1986.

[14] M.P. Herlihy. Impossibility and universality results for wait-free synchronization. In *Seventh ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC)*, August 1988.

[15] M.P. Herlihy and J.M. Wing. Axioms for concurrent objects. In *14th ACM Symposium on Principles of Programming Languages*, pages 13–26, January 1987.

[16] 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*, August 1986.

[17] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[18] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9):690, September 1979.

[19] L. Lamport. The part-time parliament. Technical Report 49, Digital Equipment Corporation, Systems Research Center, September 1989.

[20] C.H. Papadimitriou. The serializability of concurrent database updates. *Journal of the ACM*, 26(4):631–653, October 1979.

[21] M. Pease, R. Shostak, and L. Lamport. Reaching agreement in the presence of faults. *Journal of the ACM*, 27(2):228–234, 1980.

[22] F.B. Schneider. Implementing fault-tolerant services using the state machine approach: a tutorial. Technical report, Cornell Computer Science Dept., November 1987.

[23] E. Styer and G.L. Peterson. Tight bounds for shared memory symetric mutual exclusion problems. In *Proceedings of the 8th Annual ACM Symposium on Principles of Distributed Computing*, pages 177–192, August 1989.

[24] M.R. Tuttle. *Knowledge and Distributed Computation*. PhD thesis, M.I.T., 1989.