Strict Linearizability and the Power of Aborting

Marcos K. Aguilera, Svend Frølund
Internet Systems and Storage Laboratory
HP Laboratories Palo Alto
HPL-2003-241
November 21st , 2003*

E-mail: marcos.aguilera@hp.com, svend.frolund@hp.com

shared objects, concurrency, linearizability, aborting, correctness condition, specification

Linearizability is a popular way to define the concurrent behavior of shared objects. However, linearizability allows operations that crash to take effect at any time in the future. This can be disruptive to systems where crashes are externally visible. In such systems, an operation that crashes should either not happen or happen within some limited time frame—preferably before the process crashes. We define *strict linearizability* to achieve this semantics.

Strict linearizability and wait-freedom are difficult to achieve simultaneously. For example, we show that it is impossible to obtain a strictly-linearizable wait-free implementation of objects as simple as multi-reader registers from single-reader ones. To address this problem, we augment our shared objects by allowing them to *abort* their operations *in the presence of concurrency*. An aborted operation behaves like an operation that crashes: it may or may not take effect (but if it does, it does before the abort). We show that with abortable operations, there are strictly-linearizable wait-free implementations of consensus and hence of any object.

* Internal Accession Date Only Approved for External Publication

Strict Linearizability and the Power of Aborting

Marcos K. Aguilera[*] and Svend Frølund[†]
HP Labs, Palo Alto, CA 94304

21 November 2003

Abstract—Linearizability is a popular way to define the concurrent behavior of shared objects. However, linearizability allows operations that crash to take effect at any time in the future. This can be disruptive to systems where crashes are externally visible. In such systems, an operation that crashes should either not happen or happen within some limited time frame—preferably before the process crashes. We define *strict linearizability* to achieve this semantics.

Strict linearizability and wait-freedom are difficult to achieve simultaneously. For example, we show that it is impossible to obtain a strictly-linearizable wait-free implementation of objects as simple as multi-reader registers from single-reader ones. To address this problem, we augment our shared objects by allowing them to *abort* their operations *in the presence of concurrency*. An aborted operation behaves like an operation that crashes: it may or may not take effect (but if it does, it does before the abort). We show that with abortable operations, there are strictly-linearizable wait-free implementations of consensus and hence of any object.

1 Introduction

Linearizability [7] has been widely used as the correctness condition for concurrent implementations of shared objects. Roughly speaking, linearizability requires that an operation appear to take place instantaneously at some time between its invocation and response. This simple requirement has many attractive features, from both a conceptual and a pragmatic point of view: *Composability* means that if an implementation is proven linearizable when its underlying objects' operations are instantaneous, then the implementation remains linearizable when its underlying objects are replaced with linearizable implementations. This property allows to build complex linearizable objects from simpler ones in a modular fashion.

Another attractive feature of linearizability is *weak limited effect*, which means that an operation can only take effect within a limited amount of time when its caller completes. For example, consider a shared register with two operations, $read$ and $write(v)$, with the usual semantics. If a process p invokes $write(v)$ and does not crash, then weak limited effect guarantees that the *write* can take effect only until the time p returns from the *write*'s invocation. This is in contrast to, for example, sequential consistency [8], in which the *write* can take effect at any arbitrary time in the future (as long as local order is respected).

Limited effect is an important property, because it prevents old operation instances from suddenly appearing mysteriously. For example, suppose that a client withdraws money from the bank in an automated teller machine, but the machine crashes during the transaction and does not debit the client's account. The client will be annoyed if, years later, the debit suddenly appears when the client has insufficient funds. Or suppose that a military officer presses a button to launch a missile during war, but the missile does not come out. It might be catastrophic if the missile is suddenly launched years later after the war is over.

Unfortunately, linearizability does not always ensure limited effect—hence the term *weak* limited effect. In fact, it only does so if the caller does not crash: if the caller crashes, then the operation may take effect at any arbitrary time in the future. These

[*]Email: marcos.aguilera@hp.com
[†]Email: svend.frolund@hp.com

pending operation instances can be quite disruptive. For example, a pending write can destroy the value of a register unpredictably at any time in the future.

In fact, one can find linearizable implementations such that, if a process p crashes while executing an operation, then another process q may cause p's operation to take effect long in the future, even after other processes have executed many operations (e.g., in [11]).

We would like to limit the effect of an operation by the time that the caller completes or crashes. Doing so results in what we call *strict linearizability*. Intuitively, strict linearizability prohibits pending operation instances, by requiring an operation to either take effect before a crash, or never take effect. Figure 1 illustrates this idea. More precisely, strict linearizability is a strengthening of linearizability that requires an operation to take effect at some time between its invocation and either its response (if it does not crash) or its crash (if it does).

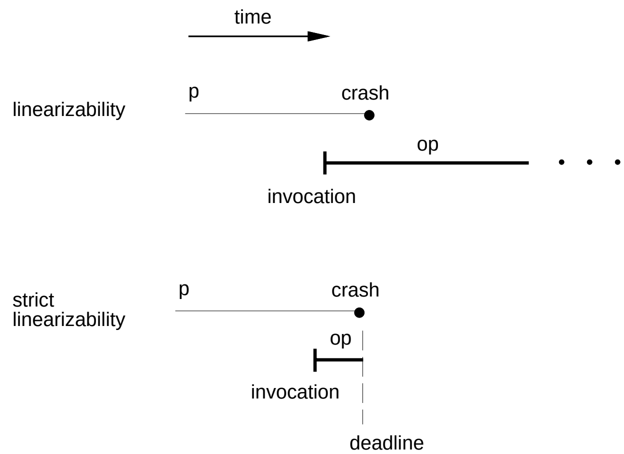

Figure 1: Difference between traditional and strict linearizability. With linearizability, the operation instance op may take effect at an arbitrary point after p crashes. With strict linearizability, op cannot take effect after the deadline created by p's crash.

Given that crashes are not observable events in asynchronous systems, strict linearizability raises two important questions: (1) does it really make sense to use these unobservable crashes to restrict the behavior of operations? (2) Is strict linearizability implementable at all?

We believe the answer to the first question is "yes", because crashes are often visible events at higher levels in the application. In fact, in practice crashes need to be eventually fixed, and hence they need to be either observable or forced upon the system. In those cases, with strict linearizability, the higher levels in the application can be assured that an operation that does not take effect before the issuer crashes will never take effect.

The answer to the second question is "it depends", as we now explain.

Wait-freedom. One difficulty with strict linearizability is that it clashes with wait-freedom. Roughly speaking, wait-freedom [5] guarantees that a process completes the execution of an operation in a finite number of its steps, regardless of the behavior of other processes. Wait-freedom is attractive because it provides a very strong form of fault-tolerance, by ensuring progress of a process even if all other processes in the system stop. Many implementations in the literature have aspired to achieve both wait-freedom and linearizability, to simultaneously provide strong fault-tolerance and strong consistency.

But what about wait-freedom and *strict* linearizability? It turns out to be very difficult to achieve both properties simultaneously. To see why, let us consider a hypothetical implementation of a shared register. Consider two scenarios. In the first one, suppose that process p invokes $write(v)$ and crashes before its response. Further suppose that a subsequent *read* by process q returns the old value of the register. Then, if the implementation is strictly linearizable, the *write* can never take effect: it cannot take effect before the crash because the *read* returns the old value, and it cannot take effect after the crash due to strict linearizability. Now consider a second scenario, which is similar to the first except that p does not crash, but only becomes very slow. The execution is indistinguishable to q, and so the *read* returns the old value. Therefore, the *write* can only take effect after the *read*. By introducing subsequent *read*'s by q in a similar fashion, it is possible to build a run where the *write* never takes effect, and hence never returns. This violates wait-freedom.

This intuitive argument can be formalized into impossibility results of many constructions that are considered basic for linearizability. For example, we can show that it is impossible to have a strictly-linearizable wait-free implementation of an object as simple as a multi-reader register using single-reader registers.

This negative result seems to limit the applicability of strict linearizability.

Abortable operations and liveness. To circumvent the impossibility result above, we allow operations to *abort* their execution under certain conditions. When an operation instance aborts, the caller receives a specially-designated response denoted $\perp$, which indicates that the operation instance may or may not have taken effect. And if it has taken effect, it did between the operation's invocation and abort. An aborted operation instance is similar to an operation instance whose caller crashes. The difference is that abort is intentionally initiated by an object, whereas a crash is not.

It is undesirable for an operation instance to abort, because it can be detrimental to liveness. Therefore we introduce various progress conditions to limit the occurrence of aborts. The *Strong progress* condition requires that a solo execution of an operation never abort. With strong progress, liveness is achieved in the absence of concurrency. In the presence of concurrency, executions of operations may abort. However, since an aborted operation instead returns a special value, processes are aware of the problem, and they can react appropriately. For example, processes can retry the operation after some exponentially backed-off delay. This heuristic guarantees liveness with high probability in the presence of some weak form of system synchrony.

Achieving strict linearizability with aborts. When aborting is allowed, we show that some of the basic constructions that apply to linearizability also apply to strictly linearizability. For example, we show how to construct a multi-writer multi-reader register from single-writer single-reader registers. The construction is similar to, but different from, the one for linearizability. Our implementations all satisfy strong progress.

Furthermore, we show that even some constructions that are known to be impossible with linearizability become actually possible with strict linearizability (and aborts). In fact, we show some surprising results: (1) it is possible to implement consensus from registers and, in fact, (2) it is possible to implement *any* object from registers. Our implementations never abort in solo executions, i.e., they satisfy strong progress.

These results show that the strict linearizability (with aborts) can be achieved.

Contributions. In summary, we make the following contributions:

- We define strict linearizability as a modification of linearizability to enforce limited effect and allow operations to abort their execution. We show that strict linearizability implies linearizability, and we show that strict linearizability is a local property.
- We use natural deduction rules as a precise way to formally specify strict linearizability.
- We consider strictly-linearizable wait-free implementations of objects. Without abort, we show that it is impossible to implement multi-reader register from single-reader registers.
- With abort, we give a strictly-linearizable wait-free implementation for any object (universal construction) using single-writer single-reader registers only. To do so, we start with single-writer single-reader registers and implement multi-writer multi-reader registers. We then use these registers to implement consensus. Finally, we use consensus and registers to implement any object.

Roadmap. We define our distributed system model in Section 2, and we define strict linearizability as a correctness condition relative to this model in Section 3. In Section 4 we introduce progress conditions that limit the situations under which an operation may abort its execution. We prove some interesting properties of strict linearizability in Section 5, including locality. In Section 6, we show that without aborts there is no strictly-linearizable wait-free implementation of a multi-reader atomic register from single-reader ones. In Section 7, we assume that operations may abort and we provide strictly-linearizable wait-free implementations of atomic registers and of consensus. We then show how to get a strictly-linearizable wait-free implementation of any object. Finally, in Section 8, we discuss related work. In the appendix, we give all the details of our register implementation, and we prove its correctness.

2 Model

We consider a distributed system with n processes: $p_1, \ldots, p_n$. Processes may fail by crashing; when a process crashes, it simply ceases to execute its algorithm (we do not consider Byzantine failures). We explicitly represent a crash through a special *crash event*. A correct process in a run is one for which there are not crash events in the run. Crash events are not visible to processes (but they may be visible to higher levels in the application).

Processes communicate by invoking operations on shared objects. The shared objects are always available and do not fail. The set Object contains all possible *objects*. Informally, each object has a set of *operations*, where each operation takes a *value* as input and returns a value as output. Values are taken from an infinite set Value.

We consider an asynchronous system, in which there is no bound on the time it takes a process to execute its instructions, including instructions that access shared objects.

3 Strict Linearizability

We first define, in Section 3.1, our representation of system executions as a *history*, which is a sequence of invocation, return, and crash events. Invocation and return events happen as processes access shared objects. We make some standard well-formed assumptions on histories, explained in Section 3.2. In particular, we assume that each process has at most one outstanding invocation at a time, so that there is no concurrency *within* a process; concurrent accesses by different processes is allowed.

Objects are defined through a *sequential specification* (Section 3.4), which specifies the behavior of an object in the absence of concurrency, that is, in a *sequential history* (Section 3.3). For example, a *register* object with *read* and *write* operations is specified through the requirement that in a sequential history a read return the most recently written value.

Intuitively, a history is *strictly linearizable* if there is a sequential history that is consistent with it from the point of view of the higher levels of the system and that complies with the sequential specification of all objects. We provide a precise definition in Section 3.5.

3.1 Events and Histories

We represent a system execution (also called a run) as a *history*. Roughly speaking, a history represents the ordering of *events* in the distributed system. More precisely, a history is a finite or infinite sequence of events. Intuitively, events are triggered by invocations and returns of operations and by the crash of processes. More precisely, there are three types of events:

- An *invocation event*, denoted $inv(op, v)^o_p$, represents the act of process p invoking operation op on object o with parameter v.
- A *return event*, denoted $ret(op, v)^o_p$, represents the act of process p receiving a response containing value v for operation op of object o.
- A *crash event*, denoted $crash_p$, represents the act of process p failing.

To represent an abort of execution, we use a return event with a specially designated value $v = \bot$, which is not part of Value. We call such a return event an *abort event*.

The set $\mathsf{History}$ contains all histories. Throughout the paper, the letter H (sometimes subscripted) denotes an element of $\mathsf{History}$. We use the following syntax for histories:

$$
\begin{array}{lcll}
H & ::= & H_1 \cdot H_2 \cdot \ \ldots \cdot H_n & | \\
 & & inv(op, v)^o_p & | \\
 & & ret(op, v)^o_p & | \\
 & & crash_p & | \\
 & & \lambda &
\end{array}
$$

where "$\cdot$" denotes sequence concatenation and λ denotes the empty sequence. In the following, we also use the notation "$\in$" and "$\notin$" to test whether an event appears in a history.

The *projection of a history H onto a process p*, denoted $H|p$, is the history obtained from H by dropping all events except those of p. The *projection of a history H onto an object o*, denoted $H|o$, is the history obtained from H by dropping all events except those of o and crash events.

For any finite history H and any process p, we define $\mathsf{last}_p(H)$ to be the last event in $H|p$, or λ if $H|p$ is the empty history.

3.2 Well-Formedness Assumptions

We assume that each process has at most one outstanding invocation at a time, that is, there is no concurrency *within* a process (but there can be concurrency *across* processes). More precisely,

$$H = H_1 \cdot inv(op, v)_p^o \cdot H_2 \cdot inv(op', v')_p^{o'} \cdot H_3 \Rightarrow ret(op, v'')_p^o \in H_2$$

Every return event must have a matching invocation. More precisely,

$$\begin{aligned} H =& H_1 \cdot ret(op, v)_p^o \cdot H_2 \Rightarrow \\ & inv(op, v')_p^o \in H_1 \\ H =& H_1 \cdot ret(op, v)_p^o \cdot H_2 \cdot ret(op', v')_p^{o'} \cdot H_3 \Rightarrow \\ & inv(op', v'')_p^{o'} \in H_2 \end{aligned}$$

A process crashes at most once, and after it crashes, it has no more events. More precisely,

$$\begin{aligned} H =& H_1 \cdot crash_p \cdot H_2 \Rightarrow \\ & crash_p \notin H_1 \wedge H_2|p = \lambda \end{aligned}$$

Finally, $\perp$ can only be part of return events. More precisely,

$$inv(op, v)_p^o \in H \Rightarrow v \neq \perp$$

3.3 Sequential Histories

A *sequential history* is an alternating sequence of invocation and return events that starts with an invocation event, and does not end with an invocation event. Sequential histories do not have crash or abort events. The set $\mathsf{SeqHistory}$ denotes all sequential histories. Throughout the paper, the letter S (sometimes subscripted) denotes an element of $\mathsf{SeqHistory}$. The syntax for sequential histories is the following:

$$\begin{array}{rcll} S & ::= & S_1 \cdot S_2 \cdot \ \ldots \cdot S_n & | \\ & & inv(op, v)_p^o \cdot ret(op, v')_p^o & | \\ & & \lambda & \end{array}$$

where $v, v' \neq \perp$.

3.4 Sequential Specification

We assume that each object has a sequential specification that captures the semantics of the object when it is invoked in a non-concurrent manner. We use the same notion of sequential specification as [7]: the sequential specification for an object is a set of sequential histories; each history in the sequential specification captures a particular "correct" interaction between the object and a number of processes that invoke it in a purely sequential manner. For any object o, we use $\mathsf{SeqSpec}_o$ to denote o's sequential specification. We assume that the empty history is always part of an object's sequential specification: $\forall o \in \mathsf{Object} : \lambda \in \mathsf{SeqSpec}_o$.

3.5 History Transformation

In general, a history contains concurrent operation instances[1], partial operation instances, crashes, and aborted operation instances. However, when reasoning about correctness, we would like to deal with simpler histories. We define a relation $\rightarrow$ to derive simpler histories from more complicated ones, while maintaining plausibility of execution. Intuitively, if $H \rightarrow H'$ then (1) H' is "consistent" with H from point of view of higher levels in the system, and (2) H' is simpler than H in the sense that H' has fewer concurrent operation instances, fewer crash events, fewer aborted operations, or fewer partial operation instances than H. Note that $\rightarrow$ is not symmetric.

We define $\rightarrow$ in Figures 2, 3, 4 and 5. Rule (1) defines $\rightarrow$ to be reflexive, and Rule (2) defines $\rightarrow$ to be transitive.

[1]Roughly speaking, an operation instance is an invocation and matching return event. This is defined in Section 4.1.

$$H \to H \tag{1}$$

$$\frac{H_1 \to H_2 \quad H_2 \to H_3}{H_1 \to H_3} \tag{2}$$

Figure 2: Reflexive and transitive rules

$$\frac{p \neq q}{H_1 \cdot inv(op, v)_p^o \cdot ret(op', v')_q^{o'} \cdot H_2 \to H_1 \cdot ret(op', v')_q^{o'} \cdot inv(op, v)_p^o \cdot H_2} \tag{3}$$

$$H_1 \cdot inv(op, v)_p^o \cdot inv(op', v')_q^{o'} \cdot H_2 \to H_1 \cdot inv(op', v')_q^{o'} \cdot inv(op, v)_p^o \cdot H_2 \tag{4}$$

$$H_1 \cdot ret(op, v)_p^o \cdot ret(op', v')_q^{o'} \cdot H_2 \to H_1 \cdot ret(op', v')_q^{o'} \cdot ret(op, v)_p^o \cdot H_2 \tag{5}$$

$$\frac{\mathsf{last}_p(H_1) = ret(op, v)_p^o \vee \mathsf{last}_p(H_1) = \lambda}{H_1 \cdot crash_p \cdot H_2 \to H_1 \cdot H_2} \tag{6}$$

$$\frac{\mathsf{last}_p(H_2) = \lambda}{H_1 \cdot inv(op, v)_p^o \cdot H_2 \cdot crash_p \cdot H_3 \to H_1 \cdot H_2 \cdot H_3} \tag{7}$$

$$\frac{\mathsf{last}_p(H_2) = \lambda \qquad v \in \mathsf{Value}}{H_1 \cdot inv(op, v')_p^o \cdot H_2 \cdot crash_p \cdot H_3 \to H_1 \cdot inv(op, v')_p^o \cdot H_2 \cdot ret(op, v)_p^o \cdot H_3} \tag{8}$$

Figure 3: Basic rules for reordering and dealing with crashes

$$\frac{H_2|p = \lambda}{H_1 \cdot inv(op, v)_p^o \cdot H_2 \to H_1 \cdot H_2} \tag{9}$$

$$\frac{(H_2 \cdot H_3)|p = \lambda}{H_1 \cdot inv(op, v)_p^o \cdot H_2 \cdot H_3 \to H_1 \cdot inv(op, v)_p^o \cdot H_2 \cdot ret(op, v')_p^o \cdot H_3} \tag{10}$$

Figure 4: Rules for dealing with operation instances that execute forever

$$\frac{\mathsf{last}_p(H_2) = \lambda}{H_1 \cdot inv(op, v)_p^o \cdot H_2 \cdot ret(op, \perp)_p^o \cdot H_3 \to H_1 \cdot H_2 \cdot H_3} \tag{11}$$

$$\frac{\mathsf{last}_p(H_2) = \lambda \qquad v \in \mathsf{Value}}{H_1 \cdot inv(op, v')_p^o \cdot H_2 \cdot ret(op, \perp)_p^o \cdot H_3 \to H_1 \cdot inv(op, v')_p^o \cdot H_2 \cdot ret(op, v)_p^o \cdot H_3} \tag{12}$$

Figure 5: Rules for dealing with aborts

Rules (3)–(5) allow the introduction of order among concurrent operation instances. Rule (6) enables removal of a crash event of a process p when no operations of p are "active". Rules (7) and (8) deal with partial operation instances, which may or may not take effect nondeterministically. Rules (9) and (10) deal with invocations without responses, which occur when a process executes forever without returning from an invocation (this could occur in some lock-free implementations). These rules are not needed for or applicable to histories in which every invocation is followed by a matching return or a crash. Finally, Rules (11) and (12) deal with aborted operations: these are essentially treated like crashes.

3.6 Definition of Strict Linearizability

We say that a well-formed history H is *strictly linearizable* if it can be transformed, under $\rightarrow$, to a sequential history where all object sub-histories are in the sequential specification of the respective objects:[2]

Definition 1

$$H \textit{ is strictly linearizable} \Leftrightarrow \\ \exists S \in \mathsf{SeqHistory}, \forall o \in \mathsf{Object} : \\ H \rightarrow S \wedge S|o \in \mathsf{SeqSpec}_o \quad (13)$$

We say that an implementation is strictly linearizable if all histories that it produces are strictly linearizable.

4 Restricting the Occurrence of Aborts

An object should not be allowed to always abort its operations, else it would be useless. Thus, we need to define properties that prevent objects from aborting in "good" circumstances. These are called *progress conditions*. In this paper we focus on a progress condition that we call *strong progress*. Roughly speaking, strong progress guarantees that if an operation instance runs solo then it does not abort. Here, "solo" is with respect to operations of the same object. We now make this more precise.

4.1 Operation Instances

Roughly speaking, an operation instance represents the execution of an operation within a history H. Unlike the events in H, operation instances in H are not atomic: they begin at an invocation event, and end in either a return of non-$\perp$ (successful), a return of $\perp$ (aborted), or a crash (partial). Or perhaps it never ends (infinite).

More precisely, we say that an *invocation event* $e = inv(op, v)^o_p$ *matches event* e' if $e' = ret(op, v')^o_p$ for some v', or $e' = crash_p$. Given an invocation event e in H and an event e' in H, we say that e *matches* e' *in* H if e' is the first event in H after e such that e matches e'.

Let e and e' be elements of a sequence H. We say that the pair $\langle e, e' \rangle$ is an *operation instance in* H if e matches e' in H. We also say that the pair $\langle e, \infty \rangle$ is operation instance in H if there is no event e'' in H such that e matches e'' in H (in this latter case, note that ∞ is not an event in H). If $e = inv(op, v)^o_p$ we say that o is the *object of operation instance* $\langle e, e' \rangle$ and p is the *process of operation instance* $\langle e, e' \rangle$.

We say that an operation instance $\langle e, e' \rangle$ is *successful* if e' is a return event whose value is not $\perp$. We say that $\langle e, e' \rangle$ is *aborted* if e' is an abort event. And $\langle e, e' \rangle$ is *complete* if it is either successful or aborted. We say $\langle e, e' \rangle$ is *partial* if e' is a crash event. We say $\langle e, e' \rangle$ is *finite* if it is partial or complete. We say $\langle e, e' \rangle$ is *infinite* if it is not finite (i.e., $e' = \infty$).

An event e *happens during an operation instance* $\langle e', e'' \rangle$ *in* H if e happens after e' in H and either $e'' = \infty$ or e'' is an event that happens after e' in H.

Two operation instances op_i and op_j are *concurrent* in H if either op_i's invocation event happens during op_j in H or if op_j's invocation event happens during op_i in H. An operation instance op_i in H is *solo in* H if there is no operation instance op_j such that (1) the objects of op_i and op_j are the same, and

[2] This definition if for a finite history H. If H is infinite, the situation is more complex and beyond the scope of this paper. One possibility is to define H to be strictly linearizable iff there exists an infinite sequential history S such that (1) for every i, there exists a history G such that $H \rightarrow P_i(S) \cdot G$, where $P_i(S)$ is the history with the first i events of S, and (2) $\forall o \in \mathsf{Object} : S|o \in \mathsf{SeqSpec}_o$. Our implementation correctness proofs all assume H is finite.

(2) op_i and op_j are concurrent in H.

4.2 Progress Conditions

Formally, a *progress condition* is a set $\mathsf{LegalAborts}$ of histories. An implementation of an object o satisfies a progress condition if for all histories H of the implementation, $H|o$ is in $\mathsf{LegalAborts}$.

Some example of progress conditions are the following, ordered by decreasing strength:

1. A solo operation instance does not abort.
2. For every process p, if there are infinitely many solo operation instances of p then infinitely many of those do not abort.
3. If there are infinitely many solo operation instances then infinitely many of those do not abort.
4. For every process p, if eventually only p has operation instances then there is a time after which operation instances do not abort.

In this paper, we focus on progress condition 1, which we call *strong progress*.

5 Properties of Strict Linearizability

We now prove some interesting properties about strict linearizability. We first show that strict linearizability implies linearizability. We then show that strict linearizability is a local property, like linearizability. (This result is *not* an immediate corollary of the first result.)

5.1 Strict Linearizability Implies Linearizability

We relate strict linearizability to traditional linearizability [7], and prove that strict linearizability implies traditional linearizability for histories without aborts. We exclude aborts because linearizability does not have this notion.[3]

[3] It is worth noting that our result holds for each history, that is, even if some implementation can sometimes abort, if it produces a strictly-linearizable history without aborts then we show that the history is also linearizable.

To allow the comparison between traditional and strict linearizability, we introduce some of the formalism used to define traditional linearizability. We only provide a summary of the various concepts; for a complete definition the reader should consult [7].

Based on the total order for events in a finite history, we introduce a partial order on the successful operation instances in the history. We say that a successful operation instance op_i *happens before* another successful operation instance op_j in a history H if the return event for op_i occurs before the invocation event for op_j in H. We write this as $op_i <_H op_j$, and use $<_H$ to refer to the set of operation pairs that satisfy this relation.

For any two histories H and H', we say that H and H' are *equivalent* if, for any process p, $H|p = H'|p$. Moreover, for any history H, $\mathsf{complete}(H)$ is the maximal subsequence of H consisting of only invocation events and matching return events. We say that a history H is *complete* if $H = \mathsf{complete}(H)$. We can now define (traditional) linearizability [7]:

Definition 2 *A finite history H without crash events and aborts is linearizable if there exists a sequential history S and return events $e_0, \ldots, e_m$ ($n \geq 0$) such that:*

- $\mathsf{complete}(H \cdot e_0 \cdot \ldots \cdot e_m)$ *is equivalent to* S.
- $<_H \subseteq <_S$.
- $\forall o \in \mathsf{Object} : S|o \in \mathsf{SeqSpec}_o$.

We now proceed to prove that strict linearizability implies linearizability.

Lemma 3 *Let H be a finite history without aborts. If a history H' satisfies $H \rightarrow H'$, then $<_H \subseteq <_{H'}$.*

PROOF: Let H be a finite history without aborts and let H' be a history such that $H \rightarrow H'$. Consider a single application of Rules (3)–(10) (we do not consider Rules (11)–(12) because H does not contain aborts). Let H_l be the history on the left-hand side, and let H_r be the history on the right-hand side, in one of these single applications.

In Rule (3), we have that $<_{H_l} \subseteq <_{H_r}$ because the rule orders op' before op. In Rules (4) and (5), we have that $<_{H_l} = <_{H_r}$. For Rule (6), we also have that

$<_{H_l}=<_{H_r}$ because removing a crash event does not change the operation instance ordering. For Rule (8) and (10), we have that $<_{H_l}\subseteq<_{H_r}$ because adding a return event introduces a new successful operation instance in the history, and may thus add to the operation instance order. Rule (7) and (9) do not change the operation instance order because H_l and H_r contain the same successful operation instances, and no events have been reordered. All in all, we have that every single application of Rules (3)–(10) satisfies the constraint $<_{H_l}\subseteq<_{H_r}$. We can now prove the lemma by straight-forward induction on the number of applications of these rules that is required to transform H to H'. ∎

Theorem 4 *Let H be a finite history without aborts and let H_{cf} be the history obtained from H by removing all crash events. If H is strictly linearizable then H_{cf} is linearizable.*

PROOF: Let H be a strictly linearizable finite history without aborts. Since H is strictly linearizable, we know that there exists a sequential history S such that $H \rightarrow S$ and such that $S|o \in \mathsf{SeqSpec}_o$ for all objects o.

We first show that we can add zero or more return events to H_{cf} and obtain a history H' such that $\mathsf{complete}(H')$ is equivalent to S. We show that this holds for any given process p. Since $H \rightarrow S$, we also have that $H|p \rightarrow S|p$, and the transformation of $H|p$ to $S|p$ involves application of Rules (6)–(10) only (transforming $H|p$ to $S|p$ does not change the ordering of events). Moreover, we can apply at most one of these rules: the application of any one of these rules prevents the subsequent application of the same rule or of another rule.

If we use Rule (6) to transform $H|p$ to $S|p$, we obtain $S|p$ by removing a crash event from $H|p$. In this case, we have $S|p = H_{cf}|p$. Moreover, $H_{cf}|p$ is a complete history because the last event in $H|p$, before the crash event, is a return event. Thus, we can construct H' by adding zero return events to H_{cf}. If we use Rule (7), we remove both a crash event and the invocation event of a partial operation instance from $H|p$. Because histories are well-formed, and from the pre-condition of the rule, we have that $S|p = \mathsf{complete}(H_{cf})|p$, and we can again construct H' by adding zero return events to H_{cf}. Finally, if we apply Rule (9), we remove the invocation event of an infinite operation instance. In this case, we have that $S|p = \mathsf{complete}(H)|p$. Moreover, because p does not crash, we also have that $\mathsf{complete}(H)|p = \mathsf{complete}(H_{cf})|p$, and we can again construct H' by adding zero return events to H_{cf}.

Consider now a transformation of $H|p$ to $S|p$ by Rule (8) or Rule (10). If we apply Rule (8), we remove a crash event and add a return event e to $H|p$. In this case we have that $S|p = (H_{cf} \cdot e)|p$, which is a complete history because the last event before the crash event in $H|p$ is an invocation event. Thus, we can construct H' be adding e to H_{cf}. If we apply Rule (10), we add a return event e' to $H|p$, and have that $S|p = (H \cdot e')|p$, which is complete because p does not crash and because H is well-formed. Again, we can construct H' by adding e' to H_{cf}. Thus, for any process p we can construct a history H', by adding zero or more return events to H_{cf}, such that $\mathsf{complete}(H')$ is equivalent to S.

We next show that $<_{H_{cf}}\subseteq<_S$. Observe first that $<_{H_{cf}}=<_H$. Since H has no aborts, and since $H \rightarrow S$, we know from Lemma 3 that $<_H\subseteq<_S$. We can now conclude that $<_{H_{cf}}=<_H\subseteq<_S$, which proves the theorem. ∎

5.2 Strict Linearizability is a Local Property

We now prove that strict linearizability is a local property [7], just like linearizability. For simplicity, we restrict attention to finite histories only.

Lemma 5 *Let H be a finite history and S be a sequential history. If H is equivalent to S, and if $<_H\subseteq<_S$, then $H \rightarrow S$.*

PROOF: Assume that $H \not\rightarrow S$. Since H and S are equivalent, they contain the same events. Moreover, since S is a sequential history, and since S is equivalent to H, we know that H does not contain crash events, aborts, infinite operation instances, or partial operation instances. Thus, the only difference between H and S is the ordering of events. However, because the histories only contain invocation and return events, and because histories are well-formed, we can use Rules (3)–(5) to change the order of any two events, except if this will change the order of operation instances. Thus, we conclude that there must be two operation instances op_i and op_j that are ordered differently in H and S. But this contradicts the fact that $<_H \subseteq <_S$. ■

Theorem 6 (Locality) *A finite history H is strictly linearizable if and only if, for all objects x, $H|x$ is strictly linearizable.*

PROOF: Consider first the "only if" part of the Theorem. Assume that $H \rightarrow S$ for some sequential history S, and assume that $S|x \in \mathsf{SeqSpec}_x$ for all objects x. We argue that $H|x \rightarrow S|x$. Consider a transformation of H to S through Rules (3)–(12). Selectively apply the same rules to transform $H|x$ to $S|x$ in the following manner. If a rule involves only events from x (i.e., an invocation of x or a return from x), then apply the rule. If the rule does not involve any events in $H|x$, ignore the rule. If the rule is Rule (6), then apply the rule. If the rule is Rule (7) or Rule (8), and the invocation event is for object x, then apply the rule, otherwise apply instead Rule (6) to remove the crash event.

Consider now that "if" part of the theorem, and assume that for all objects x there exists a sequential history S_x such that $H|x \rightarrow S_x$.

First, observe that there exists a transformation, under $\rightarrow$, from $H|x$ to S_x where we first apply Rules (6)–(12) to obtain a history H_x without crashes, aborts, or infinite operation instances, and then apply Rules (3)–(5) to reorder the events in H_x to obtain S_x. This observation follows from the following two facts:

- We can apply Rules (6)–(12) directly to $H|x$ without reordering any events first.
- Applying Rules (6)–(12) does not limit the way in which we can reorder events afterwards.

Second, observe that for all x, the rules that we use to obtain H_x from $H|x$ can also be applied to H:

- For Rules (9)–(12), this follows from the well-formedness of histories and the fact that these rules apply on a per-object basis. If the precondition of a rule is satisfied for $H|x$ then the same will be the case for H.
- For Rules (6)–(8), there are two cases to consider. If we apply Rule (6) to all per-object histories (the crash did not result in a partial operation instance in any per-object history), then we can also apply Rule (6) to H. Otherwise, there exists an object x such that we have to apply either Rule (7) or (8) to replace $crash_p$ in $H|x$. In this case we can replace $crash_p$ in the same manner in H. The ability to perform the same replacement in H as we do in $H|x$ follows from the fact that if $\mathsf{last}_p(H_2)$ is empty in $H|x$, then the same is true for H (otherwise H would not be well-formed).

From the first observation we know that, for all objects x, there exists a history H_x without crashes, aborts, or infinite operation instances, such that $H|x \rightarrow H_x \rightarrow S_x$. From the second observation we furthermore know that there exists a history H' such that $H \rightarrow H'$ and $H_x = H'|x$. Since H_x is strictly linearizable, and contains no crash events or aborts, we know from Theorem 4 that H_x is also linearizable. Since linearizability is a local property [7], we conclude that H' is also linearizable. This means that there exists a sequential history S such that H' is equivalent to S and such that $<_{H'} \subseteq <_S$. Notice that H' does not contain any partial operation instances, so we do not need to extend it in order to obtain a history that is equivalent to some S. From Lemma 5, we can now conclude that $H' \rightarrow S$, which proves the proposition since $H \rightarrow H'$ and $\rightarrow$ is transitive. ■

6 Impossibility of Strict Linearizability without Abort

If operations are not allowed to abort, we show that strictly-linearizable wait-free implementations are inherently difficult to achieve. More precisely, we show that there is no implementation of a multi-reader register from single-reader ones. The proof uses a technique that, we believe, can be used to show that other basic constructions are impossible without aborts.

To obtain stronger results, we assume that the given registers are multi-writer single-reader registers that never abort. Of course, our results hold a fortiori if they are instead single-reader single-writer and or if they may abort. Similarly, we assume that the target register need only be single-writer multi-reader, but our result holds a fortiori for a multi-writer multi-reader target register.

Theorem 7 *Consider a system with $n \geq 3$ processes. There is no strictly-linearizable wait-free implementation of a single-writer multi-reader register that never aborts from multi-writer single-reader registers that never abort.*

We prove the theorem by contradiction. Assume there is one such implementation. To differentiate between the operations of the register being implemented and the registers being used, we denote the former by capitalized words (i.e., "Read" and "Write") and the latter by non-capitalized words (i.e., "read" and "write").

Let *nil* be the initial value of the Register, and let p_w be the Writer of the register, and consider a run R in which p_w wishes to Write a value $v \neq nil$. We reach a contradiction by continuing this run in a way that the Write operation instance never completes.

Lemma 8 *Process p_w cannot complete its Write without writing to at least one register.*

PROOF: Indeed, suppose p_w completes its Write without writing to any registers. Then a Reader that executes afterwards cannot distinguish between a run prefix R_0 in which p_w Writes v and a run prefix R_1 in which p_w never Writes anything. But if the Reader executes from R_0 it has to return v, while from R_1 it has to return nil. This is impossible. ■

We now continue our construction of R. Let process p_w execute until the time t_1 when p_w has completed its first write to a register r_1. This is a multi-writer single-reader register, so it has a unique process p_{r_1} that is its reader. Let p_{s_1} be a process different from p_{r_1} and p_w.

After time t_1, p_w goes to sleep and p_{s_1} starts a Read.

Lemma 9 *The Read by p_{s_1} returns nil.*

PROOF: Indeed, process p_{s_1} does not notice the first write by p_w (since p_{r_1} is the only process that can do so). Therefore, from the point of view of p_{s_1}, the run up to time t_1 is identical to a run in which a Write never occurred. Therefore the Read by p_{s_1} has to return *nil*. ■

We now proceed by induction. Suppose that in R, we have (1) process p_w has written j times, where the last write was to register r_j and finishes at time t_j, (2) p_w has not yet finished its Write, (3) after time t_j, some process $p_{s_j} \neq p_w$ has executed a Read that returns *nil*.

We continue R by letting p_w resume its execution.

Lemma 10 *Process p_w will attempt to write to another register before completing the Write operation instance.*

PROOF: In order to obtain a contradiction, suppose that p_w completes its Write without any further writes to register. Construct a run R' that is identical to R except that process p_w crashes right before p_{s_j} starts its Read. Then, from the point of view of any process different from p_w, R and R' are indistinguishable. Now in R, suppose that after the Write of p_w completes, some process $q \neq p_w$ executes a Read. Then q Reads the value v Written by p_w, since the Read starts after the Write has completed. We now make q execute its Read in R'. Since R and R' are indistinguishable by q, it follows that q Reads v in R'. Therefore, in R' the Write of p_w is linearized at some point (rather than being eliminated). However, strict linearizability requires the linearization point to be before the crash of p_w—and hence before p_{s_j} starts its Read. Therefore, the Read of p_{s_j} must also return v. This contradicts condition (3) above of the induction hypothesis. ■

We continue R by letting p_w continue executing until it has written to another register (as ensured by Lemma 10). Let r_{j+1} be such a register, let t_{j+1} be the time when the write to r_{j+1} completes, let $p_{r_{j+1}}$ be the process allowed to read r_{j+1}, and let $p_{s_{j+1}}$ be a process different from $p_{r_{j+1}}$ and p_w.

After time t_{j+1}, we let $p_{s_{j+1}}$ execute a Read in R.

Lemma 11 *The Read by $p_{s_{j+1}}$ returns nil in R.*

PROOF: We can construct another run R' that is identical to R, except that p_w crashes right at time

t_j, but p_{s_j} executes its Read as in R (it does so because it cannot distinguish R and R'). Then, in R', we let $p_{s_{j+1}}$ execute its Read. Since $p_{s_{j+1}}$ cannot read r_{j+1}, in R' it will execute just as in R. Moreover, in R' the Write to v can never be linearized (it cannot be linearized by time t_j because the Read by p_{s_j} that follows it returns nil, and it cannot be linearized after time t_j by strict linearizability). Therefore the Read by $p_{s_{j+1}}$ returns nil in R'. Therefore the same happens in R. ■

Therefore in R, we have (1) process p_w has written $j+1$ times, where the last write was to register r_{j+1} and finishes at time t_{j+1}, (2) p_w has not yet finished its Write, (3) after time t_{j+1}, some process $p_{s_{j+1}}$ has executed solo a Read that returns *nil*.

This establishes the induction chain. We therefore get an infinite run R in which p_w never completes its Write. This is a contradiction.

7 Strict Linearizability with Abort: Everything is Possible

We how give strictly-linearizable wait-free implementations for various objects. The implementations may abort execution in the presence of concurrency. The first construction in Section 7.2 is for a multi-writer multi-reader register using a collection of single-writer single-reader registers. The second construction in Section 7.3 is for consensus using single-writer multi-reader atomic registers. We then use consensus and registers to provide a universal construction in Section 7.4. The universal construction takes an arbitrary object with a sequential specification, and provides a strictly-linearizable wait-free implementation of the object. All our implementations satisfy strong progress as long as the underlying objects also do.

7.1 Timestamps

Several of our constructions use timestamps, which we now describe. Timestamps are taken from a set with a total order represented by $<$, and with a smallest element denoted lowTS. Processes use the primitive newTS(*ts*) to generate a *globally unique* timestamp that is greater than ts.

A simple instantiation of timestamps is a pair $(counter, \textit{process-id})$, where *process-id* is used for global uniqueness and to break ties. newTS(*ts*) returns a counter one greater than *ts*'s together with the process id of the caller.

7.2 Multi-Writer Multi-Reader Register

In this section, we give a strictly-linearizable implementation of a *multi-writer multi-reader* register, that is, a shared register that can be written and read by any process in the system. To do so, we assume the availability of strictly linearizable *single-writer single-reader* registers, that is, registers that can be written by a single designated process and can be read by a (possibly different) designated process.[4]

Our construction uses $2n^2$ single-writer single-reader registers. The constructed register and the registers used in the construction have abortable operations and provide strong progress.

Algorithm 1 shows the construction. In what follows, we use capitalized words for the Read and Write operations being implemented, and non-capitalized words for the read and write operations of the underlying single-writer single-reader registers. The underlying registers are organized as two matrices: *ord* and *val*. Process p_i is the designated reader of the i-th row of the matrices and the designated writer of the i-th column.

We represent reads and writes to a register implicitly through variables (e.g., a write is represented through assignment to the register variable). At any time during the execution of a Read or Write, if some read or write aborts the execution, then the Read or Write will also abort. We do not represent this *abort propagation* explicitly in the code (this is similar to exception propagation in modern programming languages). However, for the interested reader, we present an unabridged version of the algorithm in Appendix A (which makes explicit how the abort propagation works), and we prove its correctness.

To Write, a process p_i executes four phases. In the first phase, p_i generates a timestamp for the Write

[4]These are among the most basic primitives in any distributed system, in which one node can communicate with another node. They should be either readily available or easy to implement in such systems.

Algorithm 1 Multi-writer multi-reader register implementation

SHARED VARIABLES:

ord$[1 \ldots n, 1 \ldots n]$: single-writer single-reader registers, initially lowTS
val$[1 \ldots n, 1 \ldots n]$: single-writer single-reader registers, initially $\langle$lowTS$, nil\rangle$

CODE FOR EACH PROCESS p_i:

procedure Write(v)
  new-ts $\leftarrow$ newTS$(\max_j\{ord[i,j]\})$
  write-ord(*new-ts*)
  write-val(*new-ts*, *v*)
  if *new-ts* $= \max_j\{ord[i,j]\}$ **then return** OK
  else return $\perp$

procedure Read()
  new-ts $\leftarrow$ newTS$(\max_j\{ord[i,j]\})$
  write-ord(*new-ts*)
  $\langle ts, v\rangle \leftarrow$ *read-latest-val*()
  if *ts* $>$ *new-ts* **then return** $\perp$
  write-val(*new-ts*, *v*)
  if *new-ts* $= \max_j\{ord[i,j]\}$ **then return** *v*
  else return $\perp$

procedure *write-ord*(*ts*)
  for $j \leftarrow 1$ to n **do** *ord*$[j,i] \leftarrow$ *ts*

procedure *write-val*(*ts*, *v*)
  for $j \leftarrow 1$ to n **do** *val*$[j,i] \leftarrow \langle ts, v\rangle$

procedure *read-latest-val*()
  return *val*$[i,*]$ with largest *val*$[i,*].ts$

that is higher than any timestamp in row i of *ord*. In the second phase, p_i states its intention to Write using the timestamp (procedure *write-ord*). Intuitively, this ensures that a write that does not complete is visible. In the third phase, p_i performs the actual writing (procedure *write-val*) by storing the Write's timestamp and value in i-th column of *val*. Finally, in the fourth phase, p_i checks if there is another process that stated its intention to Write, by checking if the previously generated timestamp is still the highest one in the *ord* matrix. If not, p_i aborts the Write.

A Read is very similar to a Write. It executes all the phases of Write plus an additional one: before storing a value in *write-val*, process p_i first determines what value to store. It does so by reading the i'th row in *val*, and picking the value with the highest timestamp (procedure *read-latest-val*). The intuition is that this is the most recent known value.

In the appendix, we give a proof of correctness for this algorithm, and show that it satisfies strong progress. We therefore have the following result:

Theorem 12 *Algorithm 1 is a strictly-linearizable wait-free implementation of a multi-writer multi-reader register from single-writer single-reader ones. It satisfies strong progress if the underlying registers satisfy strong progress.*

7.3 Consensus

We now consider consensus. We first give its definition, and then give a strictly-linearizable wait-free implementation of it. The definition is in terms of the properties that the consensus object satisfies in a concurrent execution. Alternatively, we could have defined it in terms of a sequential specification and then derived its properties as a consequence (doing so is a good exercise for the reader).

7.3.1 Definition

Consensus is defined in terms of an operation, *propose*(v), that returns a value or aborts, such that

- If a value is returned then that value has been previously proposed.
- If processes p_i and p_j return a value then the value is the same.

We use strong progress to limit the occurrence of aborts: if a process executes *propose* solo then it does not abort.

7.3.2 Implementation

Algorithm 2 shows a strictly-linearizable wait-free implementation of consensus from single-writer multi-reader registers. Processes share two arrays *ord* and *val* of single-writer multi-reader registers. The writer of *ord*$[i]$ and *val*$[i]$ is process p_i. *ord*$[i]$ stores a timestamp, and *val*$[i]$ stores a pair consisting of a timestamp and a value.

Algorithm 2 Consensus implementation

SHARED VARIABLES:

1: *ord*$[1..n]$: multi-reader registers, initially lowTS
2: *val*$[1..n]$: multi-reader registers, initially $\langle \mathsf{lowTS}, nil \rangle$

CODE FOR EACH PROCESS p_i:

3: **procedure** *propose*(v)
4:     $ts \leftarrow \mathsf{newTS}(\max_j\{ord[j]\})$
5:     $ord[i] \leftarrow ts$
6:     $\langle ts2, w \rangle \leftarrow val[*]$ with largest $val[*].ts$
7:     **if** $w = nil$ **then** $w \leftarrow v$
8:     $val[i] \leftarrow \langle ts, w \rangle$
9:     **if** $ts = max_j\{ord[j]\}$ **then** **return** w
10:     **else return** $\perp$

To propose a value v, a process p_i first obtains a timestamp *ts* by collecting the values of array *ord* and picking a higher timestamp than any seen. Process p_i then stores the picked timestamp in *ord*$[i]$, thereby changing the maximum timestamp to its own. Process p_i next collects the values of array *val* and picks the entry with the highest timestamp. If the value associated with that entry is *nil* then p_i changes that value to its proposed value v. Next, p_i writes to its entry *val*$[i]$ the timestamp *ts* and value w. Finally, p_i collects the values of *ord* once again. If the maximum timestamp is still its own, the process returns w as the decision value. Else, it aborts.

During execution of propose, if any operation on any of the registers aborts then the propose operation also aborts immediately after. As before, this abort propagation is not explicit in the code.

We now prove that the algorithm works. Consider a run R in which processes propose values to consensus and return values (or abort).

Lemma 13 *If a value is returned then that value has been previously proposed.*

PROOF: Through a simple induction argument we can easily show that for any process p_i, the *val*$[i].v$ always holds either *nil* or the value proposed by some process. The lemma follows because a process returns the value in *val*$[i].v$ if it is not *nil*, or its proposed value if it is *nil*. ■

Definition 14 *We say that a propose operation instance by some process p_i is* enacting *if p_i does not crash during its execution and ts $= \max_j\{ord[j]\}$ right after the assignment in line 8.*

Note that process p_i may return $\perp$ even if its propose operation instance is enacting, since $\max_j\{ord[j]\}$ may change between the executions of lines 8 and 9. However, if there are no enacting proposes then all processes that propose will always abort (since $\max_j\{ord[j]\}$ is a monotonically increasing value). In this case, correctness is trivial.

Thus, henceforth we assume that there is at least one enacting propose.

Definition 15 *Let F be the enacting propose in R to first execute the assignment in line 5,*[5] *p_F be the process that executes it, t_F be the time when p_F assigns in line 8, ts$_F$ be the timestamp in the assignment, and v_F be the value in the assignment.*

Lemma 16 *By time t_F, no processes have yet assigned a larger timestamp than ts$_F$ in line 5.*

PROOF: Indeed, if by time t_F some process had assigned a larger timestamp than *ts*$_F$ in line 5 then F would not be an enacting propose. ■

Lemma 17 *From time t_F, the val$[*]$ with largest val$[*].ts$ is always equal to v_F.*

[5] By "first execute" we mean the propose whose assignment is linearized first.

PROOF: Consider the execution of an enacting *propose* different from F by some process. If the assignment in line 8 happens before time t_F then it irrelevant for what happens from time t_F onward. So assume it happens after time t_F. If the assignment in line 5 happens before time t_F then by Lemma 16 the timestamp used in line 5 is smaller than *ts*$_F$ (it cannot be equal to *ts*$_F$ because we assume that timestamps are unique). Therefore, the assignment in line 8 does not change the *val*$[*]$ with the largest timestamp.

Thus, the only *proposes* that can change the *val*$[*]$ with the largest timestamp are those in which assignments in lines 8 and 5 happen after time t_F. Consider the set of all such *proposes*. Note that for any of them, the reads in line 6 also happen after time t_F. A trivial induction argument shows that the *val*$[*]$ with largest timestamp has value v_F: this is the value read in line 6, which is used to update *val*$[i]$ in line 8. ■

Corollary 18 *If process p_j returns a value upon proposing then it returns v_F.*

PROOF: Consider a *propose* by some process p_i. If p_i completes line 5 before F (the first enacting propose) does then this is not enacting and hence either aborts or it never completes. Now assume that p_i completes line 5 after F. There are two cases. (1) If p_i completes line 5 before time t_F, then the timestamp assigned in line 5 is smaller than *ts*$_F$ (it if were bigger then F would not be an enacting propose). Thus, when p_i reaches line 9, it will find a larger timestamp than its own, and it will abort. (2) If p_i completes line 5 after time t_F, then by Lemma 17 p_i will set w to v_F in line 6, and so p_i either aborts or returns v_F. ■

This shows correctness of the algorithm. Wait-freedom is immediate from the fact that the implementation has no loops. And strong progress follows from the fact that if process p_i runs solo then the timestamp assigned in line 5 continues to be the largest timestamp in vector *ord* when p_i executes line 9. Therefore, when running solo p_i does not abort. We therefore have the following result:

Theorem 19 *Algorithm 2 is a strictly-linearizable wait-free implementation of consensus. It satisfies strong progress if the underlying objects satisfy strong progress.*

7.4 Universal Construction

We now show how to get a strictly-linearizable wait-free implementation of any object from consensus and registers (universal construction [5]). To do so, we implement an *atomic list*. Intuitively, this object keeps track of a list of strings, initially empty. There is exactly one operation, *append*, which (1) appends a string passed as parameter to the list, and (2) returns the entire new list. Like with other objects in this paper, we allow *append* to abort.

It is clear that an atomic list can be used to implement any strictly linearizable object, by using the *append* operation with a string description of the operation of T to execute; the return value of append is then used to recompute the new state of T from the sequence of operations in the list. If append aborts, the operation of T also aborts.[6]

Figure 3 shows the implementation of an atomic list. It uses a vector *consensus* of consensus objects indexed by the natural numbers, and a vector *last* of single-writer multi-reader integer registers indexed by process numbers, where the writer of an element $last[i]$ is process p_i. As in previous algorithms, if during the execution of *append* any operation on $consensus[i]$ or $last[i]$ aborts, then the *append* also aborts immediately after. This is not explicitly represented in the code. A process also has a global local variable *seq* that stores an integer, initially 0.

To append a string s to the list, a process p_i needs to first obtain the current state of the list. To do so, p_i reads each value in vector *last*, in some arbitrary order, and assigns the largest value to *maxlast*. If that integer is zero (the initial value) then the current state of the list is empty. Else, p_i obtains the state of the list by reading the decision value from $consensus[maxlast]$. It does so by proposing a dummy value *nil* to this consensus object. (As we

[6] This implementation works for *deterministic* operations. For non-deterministic operations, one can use an extra vector of consensus objects to keep the state after each operation. More precisely, after a process gets a *list* (of operations) from *append*, it sets a variable *state* to the initial state of T and then for $i = 1, \dots, len(list)$, the process (1) executes the i-th operation in the list starting from *state*, (2) proposes the result to the i-th consensus (if consensus aborts, the operation of T also aborts), (3) sets *state* to the decision value. Once done with all i's, the process returns *state*.

will show, this consensus object will always have previously decided, so that *nil* can never be the decision value.) Process p_i then appends s to its local copy of the list, and increments its *seq* variable. This variable is used, together with the process id, as a unique identifier. Process p_i then tries to change the global state of the list by proposing its local list, together with the unique identifier, to the next consensus object. Next, p_i updates its entry $last[i]$ of the last vector. It then checks if the consensus proposal has actually decided on its proposed value or not. If it has, p_i is done and returns the new list. Else, p_i retries to append s to the list in exactly the same way as before, using the next consensus object. If it fails once again, p_i aborts its operation. Else, it returns the new list.

Algorithm 3 Atomic list implementation

SHARED VARIABLES:

1: $\mathit{consensus}[1..\infty]$: consensus objects
2: $\mathit{last}[1..n]$: single-writer registers, initially 0

CODE FOR EACH PROCESS p_i:

3: **procedure** initialization
4: $\quad \mathit{seq} \leftarrow 0$
5: **procedure** *append*(s)
6: $\quad \mathit{maxlast} \leftarrow \max_i\{\mathit{last}[i]\}$
7: $\quad$ **if** $\mathit{maxlast} = 0$ **then** $\mathit{list} \leftarrow \lambda$
8: $\quad$ **else** $\langle q, x, \mathit{list}\rangle \leftarrow \mathit{propose}(\mathit{consensus}[\mathit{maxlast}], \mathit{nil})$
9: $\quad \mathit{nextlist} \leftarrow \mathit{list} \cdot s$
10: $\quad \mathit{seq} \leftarrow \mathit{seq} + 1$
11: $\quad \langle j, x, \mathit{list}\rangle \leftarrow \mathit{propose}(\mathit{consensus}[\mathit{maxlast}+1], \langle i, \mathit{seq}, \mathit{nextlist}\rangle)$
12: $\quad \mathit{last}[i] \leftarrow \mathit{maxlast} + 1$
13: $\quad$ **if** $i \neq j$ or $x \neq \mathit{seq}$ **then**
14: $\qquad \mathit{nextlist} \leftarrow \mathit{list} \cdot s$
15: $\qquad \mathit{seq} \leftarrow \mathit{seq} + 1$
16: $\qquad \langle j, x, \mathit{list}\rangle \leftarrow \mathit{propose}(\mathit{consensus}[\mathit{maxlast}+2], \langle i, \mathit{seq}, \mathit{nextlist}\rangle)$
17: $\qquad \mathit{last}[i] \leftarrow \mathit{maxlast} + 2$
18: $\qquad$ **if** $i \neq j$ or $x \neq \mathit{seq}$ **then** **return** $\perp$
19: $\quad$ **return** *list*

We now show correctness of this algorithm. First note that there are no loops, and so the implementation is wait-free. Now consider a run of the above implementation and let H be the resulting history. For simplicity, we assume that no two invocations of $append(s)$ contain the same string s. We do not lose generality in doing so because the exact value of s does not really affect the essence of execution (note that s is only used in lines 9 and 14).

Definition 20 *Let* $M = \max_i\{last[i]\}$.

Note that the value of M changes with time.

Lemma 21 *M is monotonically nondecreasing.*

PROOF: Indeed, a process p_i only updates $last[i]$ with a value larger than the previous value of $last[i]$ since the *max* in line 6 includes $last[i]$. ■

Lemma 22 *For* $1 \leq j \leq M$, $consensus[j]$ *has decided some non-nil value, and for* $j > M$, $consensus[j]$ *has not decided nil.*

PROOF: The invariant of the lemma holds initially when $M = 0$, because the first consensus object is $consensus[1]$ and, for $j > 0$, $consensus[j]$ has not decided any value. Moreover, line 8 clearly keeps the invariant because (1) $maxlast \leq M$ since M is monotonically nondecreasing, and (2) before line 8 is executed, $consensus[maxlast]$ has already decided some value that is not *nil* by the invariant. Lines 11 and 16 also maintain the invariant because the proposal value is not *nil*. Finally, lines 12 and 17 may increment M, but the invariant is maintained due to the propose operation in lines 11 and 16, respectively. ■

Lemma 23 *For every* $j \geq M+2$, $consensus[j]$ *has not decided.*

PROOF: This holds because when a process proposes to $consensus[j]$, it is always the case that $j \leq M+1$. ■

Definition 24 *Let N be the index of the highest consensus object that decides.*

Lemma 25 *For all* $j = 1, \ldots, N$, $consensus[j]$ *decides some non-nil value.*

PROOF: Let M_{max} be the largest value of M in the execution. From Lemma 23, $N \leq M_{max} + 1$. Now the result follows from Lemma 22. ■

Note that the non-nil values proposed to consensus (lines 11 and 16) are of the form $\langle *, *, \mathit{list}\rangle$, where *list* is a non-empty list. Hence, the decision values are also of this form. This motivates the following definition:

Definition 26 *For $j = 1, \ldots, N$, let i_j and s_j be such that the decision of $\mathit{consensus}[j]$ is $\langle i_j, *, \mathit{list} \cdot s_j\rangle$.*

Lemma 27 *For $j = 1, \ldots, N$, some process invokes $\mathit{append}(s_j)$.*

PROOF: Indeed, the non-nil propose values are always of the form $\langle *, *, \mathit{list} \cdot s\rangle$ where s is the parameter to append. ■

Lemma 28 *If $j \neq k$ then $s_j \neq s_k$.*

PROOF: Recall that we are assuming that no two invocations to $\mathit{append}(s)$ have the same s. Note that $\langle *, *, \mathit{list} \cdot s_j\rangle$ can only be proposed during the execution of $\mathit{append}(s_j)$. Moreover, there can be at most two such proposes in the execution, and the second propose only happens if the first propose does not decide on the proposed value. Therefore at most one consensus object can decide on $\langle *, *, \mathit{list} \cdot s_j\rangle$. It follows that if $j \neq k$ then $s_j \neq s_k$. ■

Definition 29 *An $\mathit{append}(s)$ operation instance is* successful *if it executes without aborting or crashing. An $\mathit{append}(s)$ operation instance is* effective *if $s = s_j$ for some j.*

Intuitively, an effective append is one whose parameter s has been taken by one of the consensus.

Lemma 30 *If p executes $\mathit{append}(s)$ solo without crashing then $\mathit{append}(s)$ is successful.*

PROOF: Consider p's execution of $\mathit{append}(s)$, and let M_0 be the value of *maxlast* after p executes line 6. Note that at this time, $M = M_0$. Therefore, by Lemma 23, $\mathit{consensus}[M_0 + 2]$ has not decided any value. There are now two cases: (1) if the *if* in line 13 evaluates to false then p does not abort and so $\mathit{append}(s)$ is successful. (2) If the *if* in line 13 evaluates to true then execution reaches line 16. At this time, $\mathit{consensus}[M_0+2]$ has not yet decided any value, since p is executing solo, and therefore it will decide on the proposed value, and so the *if* in line 18 evaluates to false. Therefore, p does not abort and so $\mathit{append}(s)$ is successful. ■

Lemma 31 *If $\mathit{append}(s)$ is successful then it is effective.*

PROOF: Indeed, let p_i be the process to execute $\mathit{append}(s)$. Since p_i does not abort then the propose in line 11 or 16 returns the proposed value, which is of the form $\langle *, *, * \cdot s\rangle$. Therefore, the corresponding consensus decides on that value, and so $s = s_j$ for some j. ■

Lemma 32 *If $\mathit{append}(s_j)$ is effective then, when $\mathit{append}(s_j)$ returns or crashes, $M \geq j - 1$.*

PROOF: If $\mathit{append}(s_j)$ is effective then during its execution, the propose in either line 11 or 16 returns the proposed value. At that point, $M \geq j - 1$. The result follows from Lemma 21. ■

Lemma 33 *For $j \neq k$, if $\mathit{append}(s_j)$ and $\mathit{append}(s_k)$ are effective, and $\mathit{append}(s_j)$ returns or crashes before $\mathit{append}(s_k)$ is invoked, then $k > j$.*

PROOF: Let M_j be the value of M when $\mathit{append}(s_j)$ returns or crashes. By Lemma 32, we have $M_j \geq j - 1$. When $\mathit{append}(s_k)$ is later invoked by some process p_i, p_i will set $\mathit{maxlast}$ to a value $l \geq M_j$. Since the append of p_i is effective, the propose by p_i to either $\mathit{consensus}[l + 1]$ or $\mathit{consensus}[l + 2]$ returns the proposed value. Therefore, $k = l + 1$ or $k = l + 2$. In either case, $k - 1 \geq l \geq M_j \geq j - 1$. Thus $k \geq j$. Since $j \neq k$ by assumption, it follows that $k > j$. ■

Lemma 34 *If $\mathit{append}(s_j)$ is successful then it returns the list $s_1 \cdots s_j$.*

PROOF: Using the way in which $\mathit{nextlist}$ is assigned in lines 9 and 14, we can show through a simple induction on j that $consensus[j]$ decides on $\langle *, *, s_1 \cdots s_j \rangle$. The result then follows. ■

We define define a sequential history S using the s_j's as follows:

Definition 35 *Let* $S :=$
$inv(append, s_1)^o_{p_{i_1}} \cdot ret(append, s_1)^o_{p_{i_1}} \cdot$
$inv(append, s_2)^o_{p_{i_2}} \cdot ret(append, s_1 \cdot s_2)^o_{p_{i_2}} \cdot$
$\vdots$
$inv(append, s_N)^o_{p_{i_N}} \cdot ret(append, s_1 \cdots s_N)^o_{p_{i_N}}$

We now show how we can transform H (recall that H is the history of some execution of the atomic list implementation) into S using $\rightarrow$. We first use Rules (7) and (11) of $\rightarrow$ to remove from H any *non-effective* operation instances $append(s)$ that abort or crash. We then use Rule 6 to remove all crash events that are not part of any operation instance. Let H_1 be the resulting history. Since successful operation instances in H_1 are always effective (by Lemma 31), H_1 is only left with $append(s)$ operation instances that are effective.

We then use Rules (8) and (12) to transform an abort event or the remaining crash events into normal return events, as follows: let $append(s_j)$ be an operation instance that aborts or crashes. Replace the abort or crash event with $ret(append, s_1 \cdots s_j)^o_{p_{i_j}}$. We do that for all abort and crash events. Let H_2 be the resulting history.

Now consider event $inv(append, s_1)^o_{p_{i_1}}$ in H_2. By Lemma 33, there are no *ret* events in H (or in H_2) before this *inv* event. Therefore, by multiple applications of Rule (4), we can bring forward the *inv* event to the beginning of the history H_2. Then, by multiple applications of Rules (3) and (5), we can bring forward the *ret* event that matches this *inv* event right after the *inv* event.

We can repeat this process for all remaining *append* operation instances in order from $append(s_2)$ through $append(s_N)$. By doing so, we finish with a sequential history H_{final} of alternating *inv* and *ret* events for $s_1, s_2, \ldots, s_N$ in order. The *inv* events in H_{final} exactly match those in S. As for the *ret* events, by Lemma 34, the *ret* events in H_{final} of successful operation instances in H match those events in S. As for the other *ret* events in H_{final}, those come from replacing crashes or aborts above (going from H_1 to H_2). Therefore, by construction, those events also match those in S. We conclude that $H_{final} = S$.

Therefore, we have a strictly-linearizable implementation of an atomic list. We already showed that the implementation is wait-free. Moreover, it satisfies strong progress because it only aborts an operation if the underlying objects abort their operation. Therefore, we get the following result:

Theorem 36 *Algorithm 3 is a strictly-linearizable wait-free implementation of an atomic list. It satisfies strong progress if the underlying objects satisfy strong progress.*

As we argued before, it is easy to use an atomic list to build any other object, and so the following holds:

Theorem 37 *Any object has a strictly-linearizable wait-free implementation from single-writer single-reader registers. It satisfies strong progress if the underlying registers satisfy strong progress.*

8 Related Work

The general idea that concurrency may prevent successful completion goes as far back as database transactions that abort. The "safe" registers of [9] allow a read that is concurrent with a write to return an arbitrary value. This is different from our notion of abort because with safe registers, a process does not know if its read is successful or returns garbage. With obstruction-freedom [6], processes are not required to return from their operations in the presence of concurrency. This is in contrast to our work, in which processes instead return an abort indication.

Lots of prior work has considered *specific* problems or objects with abort (rather than a general framework as we do), including consensus in [10, 1] or a register variant in [2]. The storage registers of [4, 3] are examples of strictly linearizable implementations of registers on top of an asynchronous message-passing system. Abortable consensus [12] is a problem defined for message-passing systems, which resembles the consensus objects in this paper, but the conditions for aborting are very different.

The universal construction in our work is similar to the one in [5], but we do not need its "helping mechanism", whereby one process helps to complete another process's operation. These types of helping mechanisms appear frequently in the wait-free literature, but in general they are quite ad hoc and complicated to design. Finally, our implementation of a multi-writer multi-reader register from single-writer single-reader ones is heavily inspired by the one described in [11].

References

[1] R. Boichat, P. Dutta, S. Frolund, and R. Guerraoui. Deconstructing Paxos. *ACM SIGACT*, 34(1), March 2003.

[2] P. Dutta, S. Frolund, R. Guerraoui, and B. Pochon. An efficient universal construction for message-passing systems. In *International Symposium on Distributed Computing (DISC)*, 2002.

[3] S. Frolund, A. Merchant, Y. Saito, S. Spence, and A. Veitch. Building storage registers from crash-recovery processes, October 2003. Tech report HPL–SSP–2003–14.

[4] S. Frolund, A. Merchant, Y. Saito, S. Spence, and A. Veitch. Fab: Enterprise storage systems on a shoestring. In *Proceedings of the Ninth Workshop on Hot Topics in Operating Systems (HOTOS IX)*. USENIX, 2003.

[5] M. Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):123–149, January 1991.

[6] M. Herlihy, V. Luchangco, and M. Moir. Obstruction-free synchronization: double-ended queues as an example. In *Proceedings of the 23rd International Conference on Distributed Computing*, 2003.

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

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

[9] L. Lamport. On interprocess communication. *Distributed computing*, 1(1):77–101, 1986.

[10] B. Lampson. How to build a highly available system using consensus. In *Proceedings of the International Workshop on Distributed Algorithms, Springer-Verlag, LNCS (WDAG)*, September 1996.

[11] P. M. B. Vitanyi and B. Awerbuch. Atomic shared register access by asynchronous hardware. In *IEEE Foundations of Computer Science*, 1986.

[12] Private communication with Wei Chen, March 2003.

A Constructing an Atomic Register

We give the unabridged version of our register construction with explicit abort handling. We then show its correctness, by showing that it is a strictly-linearizable wait-free implementation.

Algorithm 4 Multi-writer multi-reader register implementation

SHARED VARIABLES:

ord$[1 \ldots n, 1 \ldots n]$: single-reader, single-writer registers, initially lowTS

val$[1 \ldots n, 1 \ldots n]$: single-reader, single-writer registers, initially $\langle nil, \mathsf{lowTS}\rangle$

CODE FOR EACH PROCESS p_i:

procedure Write(*val*)
- *new-ts* $\leftarrow$ generate-ts()
- **if** *new-ts* $= \perp$ **then return** $\perp$
- **if** inc-ord-ts(*new-ts*) $= \perp$ **then return** $\perp$
- **if** write-val(*val*, *new-ts*) $= \perp$ **then return** $\perp$
- **return** check(*new-ts*)

procedure Read()
- *new-ts* $\leftarrow$ generate-ts()
- **if** *new-ts* $= \perp$ **then return** $\perp$
- **if** inc-ord-ts(*new-ts*) $= \perp$ **then return** $\perp$
- *val* $\leftarrow$ get-latest-val(*new-ts*)
- **if** *val* $= \perp$ **then return** $\perp$
- **if** write-val(*val*, *new-ts*) $= \perp$ **then return** $\perp$
- **if** check(*new-ts*) $=$ OK **then return** *val*
- **else return** $\perp$

Algorithms 4 and 5 contain the atomic register construction. In the following, we prove that the algorithms correctly implement a multi-writer multi-reader atomic register.

To distinguish between operations on the constructed multi-writer multi-reader register and operations on the underlying single-writer single-reader registers, we use Read and Write to refer to the former and read and write to refer to the latter.

We use $\mathsf{Write}(v)$ to represent a Write operation instance whose invocation event has v as parameter value. We use $\mathsf{Read}(v)$ to represent a successful Read operation instance whose return event has v as parameter value. The value *nil* ($nil \in \mathsf{Value}$) represents the initial value of the register. To simplify the presentation, we assume that each value is written at most once (i.e., we never have two Write operation instances with the same value). We also assume that *nil* is not part of any Write operation instance. We use $\mathsf{write}_r(v)$ to represent a write operation instance on a the register r, and we use $\mathsf{read}_r(v)$ to represent a successful read operation on the register r.

For any history H, we extend the ordering $<_H$ on successful operation instances in H to also include aborted and partial operation instances. For any two operation instances op_i and op_j in H, we say that $op_i \rightarrow op_j$ if op_i's return or crash event precedes op_j's invocation event in H.

For any history H, we define the following subsets of Value:

- $\mathsf{Written}_H$ is the set of all values in invocation events for Write operation instances in H.
- $\mathsf{Commited}_H$ is the set of all values in invocation events for successful Write operation instances in H.
- Read_H is the set of all values in return events for successful Read operation instances in H.

We also call the set $\mathsf{Read}_H \cup \mathsf{Commited}_H$ the *observable* values in H, and define

$$\mathsf{Obs}_H \equiv \mathsf{Read}_H \cup \mathsf{Commited}_H.$$

In the following, R is any run of Algorithms 4 and 5, and H refers to any history that R may give rise to.

Algorithm 5 Auxiliary procedures

```
CODE FOR EACH PROCESS p_i:
procedure generate-ts()
   latest-ts ← lowTS
   for j ← 1 to n do
      ts ← ord[i,j].read()
      if ts = ⊥ then return ⊥
      if latest-ts < ts then
         latest-ts ← ts
   return newTS(latest-ts)
procedure inc-ord-ts(ts)
   for j ← 1 to n do
      if ord[j,i].write(ts) = ⊥ then  return ⊥
   return OK
procedure get-latest-val(new-ts)
   latest-ts ← lowTS
   for j ← 1 to n do
      v ← val[i,j].read()
      if v = ⊥ then return ⊥
      ⟨val, ts⟩ ← v
      if ts > new-ts then return ⊥
      if latest-ts < ts then
         latest-ts ← ts
         latest-val ← val
   return latest-val
procedure write-val(val, ts)
   for j ← 1 to n do
      if val[j,i].write(⟨val, ts⟩) = ⊥ then
         return ⊥
   return OK
procedure check(ts)
   for j ← 1 to n do
      ord-ts ← ord[i,j].read()
      if ord-ts = ⊥ or ord-ts > ts then return ⊥
   return OK
```

A.1 A Sufficiency Condition for Strict Linearizability of our Construction

Intuitively, a *conforming total order* is a totally-ordered set $(V, <)$ such that (a) V contains all the observable values in H, and (b) the ordering of values in V corresponds to the ordering of operation instances in H. More precisely:

Definition 38 *A totally ordered set $(V, <)$ is a conforming total order for H if* $\mathsf{Obs}_H \subseteq V \subseteq$

$\mathsf{Written}_H \cup \{nil\}$ *and if for all* $v, v' \in V$ *the following holds:*

$$nil \in V \Rightarrow nil \le v \quad (14)$$

$$\mathsf{Write}(v) \rightarrow_H \mathsf{Write}(v') \Rightarrow v < v' \quad (15)$$

$$\mathsf{Read}(v) \rightarrow_H \mathsf{Read}(v') \Rightarrow v \le v' \quad (16)$$

$$\mathsf{Write}(v) \rightarrow_H \mathsf{Read}(v') \Rightarrow v \le v' \quad (17)$$

$$\mathsf{Read}(v) \rightarrow_H \mathsf{Write}(v') \Rightarrow v < v' \quad (18)$$

■

Proposition 39 *If H has a conforming total order then H is strictly linearizable.*

PROOF: Assume that $(V, <)$ is a conforming total order for H. Because strict linearizability is a local property (Theorem 6), we can prove that H is strictly linearizable by proving that each object sub-history $H|o$ is strictly linearizable. By assumption, the registers in *val* and *ord* are strictly linearizable. Thus, if we use O to refer to the multi-writer multi-reader object in Algorithms 4 and 5, is sufficient to show that $H_O = H|O$ is strictly linearizable.

To show that H_O is strictly linearizable, we construct a sequential history S such that $H_O \rightarrow S$.

For every $v \in V$, construct a sequence S_v as follows:

$$S_v = \begin{cases} \mathsf{Write}(v) \cdot \mathsf{Read}_1(v) \cdot \ldots \cdot \mathsf{Read}_k(v) & v \neq nil \\ \mathsf{Read}_1(v) \cdot \ldots \cdot \mathsf{Read}_k(v) & \text{otherwise} \end{cases}$$

where k is the number of successful Read operation instances that return v in H_O ($k \ge 0$). Next, construct S in the following way:

$$S = S_{v_1} \cdot \ldots \cdot S_{v_m}$$

where $v_1 < v_2 < \ldots < v_m$ are the elements of V.

First observe that S belongs to the sequential specification of a multi-writer multi-reader register: in S, a Read operation instance always returns the value of the most recent Write operation instance.

We now show that $H_O \rightarrow S$. To do so, we start with H_O and successively explain which rules to apply until we obtain S. First, use Rules (6)–(12) to remove all partial, aborted, and infinite Read operation instances from H_O as follows. Let v be the parameter of a Write operation instance. If $v \in V$, use Rule (8), (10), or (12) to convert the Write operation instance to a successful Write; otherwise, use Rule (6), (7), (9), or (11) to remove the Write operation instance. We now have a history H'_O without crashes, aborts, and infinite aborted operation instances. Moreover, $H_O \rightarrow H'_O$.

We next show that $H'_O \rightarrow S$. We first claim that H'_O and S contain the same operation instances. To show the claim, note that every successful operation instance in H_O is part of both H'_O and S. Moreover, every unsuccessful Read operation instance (i.e., partial, aborted, and infinite Read operation instances) in H_O are in neither H'_O nor S. An unsuccessful Write operation instance $\mathsf{Write}(v)$ is part of S if and only if $v \in V$. But if $v \in V$, we convert the unsuccessful Write operation instance in H_O to a successful instance in H'_O by the above transformation. This shows the claim.

Now, assume for a contradiction that $H'_O \not\rightarrow S$. Because the histories contain the same set of operation instances, and because these are all successful operation instances, there must be two operation instances op_i and op_j that are ordered differently in H'_O and S. But this is impossible because the value ordering in V obeys the operation instance ordering in H_O and thereby in H'_O. ■

A.2 Constructing a Conforming Total Order

We show that our algorithm gives rise to a conforming total order. We construct a conforming total order for values in terms of the timestamps that are used to store these values in the underlying single-writer single-reader registers. To define the total order of values, we first introduce two types of internal events related to our algorithm: store events and order events.

An order event $\mathsf{ord}(v, ts)$ happens when a process invokes the *write-val* procedure with v and ts as parameters. A store event $\mathsf{st}(v, ts)$ happens when the *check* procedure returns OK to a process. The parameters v and ts are the same as in the ordering event of the operation instance that invokes *check*.

We use OE_R^v to denote the (possibly empty) set of ordering events that happen in R and that have v as first parameter. If $\mathsf{OE}_R^v \neq \emptyset$, we use ts_v to denote the smallest timestamp that is part of any ordering event in OE_R^v.[7] We similarly define SE_R^v as the set of store events that happen in run R and that have v as first parameter. Finally, we define SV_R to be the set of values that are part of store events in R.

Definition 40 *The order relation $<_{\mathit{val}}$ on SV_R is defined as follows:*

$$v <_{\mathit{val}} v' \quad \Leftrightarrow ts_v < ts_{v'} \quad v, v' \in \mathsf{SV}_R \tag{19}$$

■

The $<_{\mathit{val}}$ relation is a total order because different values are always stored with different timestamps. In the following, we omit the subscript from $<_{\mathit{val}}$, and simply use "$<$". With this convention, the symbol $<$ is overloaded to order both timestamps and values.

Lemma 41

$$\mathsf{Obs}_H \subseteq \mathsf{SV}_R \subseteq \mathsf{Written}_H \cup \{nil\}.$$

PROOF: Let $v \in \mathsf{Obs}_H$. Then either $v \in \mathsf{Commited}_H$ or $v \in \mathsf{Read}_H$. If $v \in \mathsf{Commited}_H$ then v is the parameter of a successful Write operation instance. The invocation of *check* in this instance thus returns OK, which means that R contains a store event with v as first parameter, and so $v \in \mathsf{SV}_R$. If $v \in \mathsf{Read}_H$ then v is the return value of a successful Read operation instance. Again, the invocation of *check* within this instance returns OK, and again we conclude that $v \in \mathsf{SV}_R$.

Now let $v \in \mathsf{SV}_R$. Consider any store event with v as first parameter. There are two cases to consider: (a) the event happens during a Read operation instance and (b) the event happens during a Write operation instance. In case (a), v is returned by *get-latest-val*, which means that v is stored in a *val* register. A simple induction shows that this only happens if $v = nil$ or if v is the parameter of some Write operation instance. In case (b), v is parameter of a Write operation instance. In either case, we have that $v \in \mathsf{Written}_H \cup \{nil\}$. ■

[7] Although ts_v depends on R we do not parameterize ts_v with that run for brevity.

Lemma 42 *Each register in val and ord contain monotonically increasing timestamps.*

PROOF: Consider register *ord*$[i,j]$. This register is only written by process p_j. But each process generates a monotonically increasing sequence of timestamps, which proves the lemma for *ord*. We can apply a similar reasoning to *val*. ■

Lemma 43 *Let v be any value in Obs_H. If $nil \in \mathsf{Obs}_H$ then $nil \leq v$.*

PROOF: Assume for a contradiction that nil and v are values in Obs_H, yet $v < nil$. From Lemma 41, we know that $v, nil \in \mathsf{SV}_R$.

Let p_k be the process at which the store event $\mathsf{st}(v, ts_v)$ happens, and let p_m be the process at which the store event $\mathsf{st}(nil, ts_{nil})$ happens.

Since no Write operation instance has nil as parameter, the store event $\mathsf{st}(nil, ts_{nil})$ must happen during a Read operation instance. Consider the $\mathsf{read}_{\mathit{val}[m,k]}(\langle v, ts\rangle)$ operation instance that happens during this Read operation (as part of *get-latest-val*). We claim that that $v = nil$ and $ts = \mathsf{lowTS}$. Assume otherwise. Since Read returns nil, *get-latest-val* also returns nil. So some read operation on a register in *val* must return $\langle nil, ts'\rangle$ with $ts' > \mathsf{lowTS}$. This means that there is an ordering event $\mathsf{ord}(nil, ts')$ in R. Since *get-latest-val* does not abort, we know that $ts' < ts_{nil}$, which contradicts the definition of ts_{nil} and thereby proves the claim. The Read operation also gives rise to $\mathsf{write}_{\mathit{ord}[k,m]}(ts_{nil})$ as part of the *inc-ord-ts* procedure. We know that $\mathsf{write}_{\mathit{ord}[k,m]}(ts_{nil})$ is linearized before $\mathsf{read}_{\mathit{val}[m,k]}(\langle nil, \mathsf{lowTS}\rangle)$ because they are invoked by the same process p_m in that order.

The store event $\mathsf{st}(v, ts_v)$ happens during some Read or Write operation. This Read or Write operation invokes *write-val* and gives rise to $\mathsf{write}_{\mathit{val}[m,k]}(\langle v, ts_v\rangle)$. We claim that $\mathsf{write}_{\mathit{val}[m,k]}(\langle v, ts_v\rangle)$ is linearized before $\mathsf{write}_{\mathit{ord}[k,m]}(ts_{nil})$. To prove this claim, observe first that $ts_v < ts_{nil}$. Thus, a linearization of $\mathsf{write}_{\mathit{ord}[k,m]}(ts_{nil})$ before $\mathsf{write}_{\mathit{val}[m,k]}(\langle v, ts_v\rangle)$ would either violate the fact that *ord*$[m,k]$ contains monotonically increasing timestamps (Lemma 42), or it would contradict the fact that $\mathsf{st}(v, ts_v)$ happens in R. This proves the claim. We conclude that $\mathsf{write}_{val[m,k]}(\langle v, ts_v\rangle)$ is linearized before $\mathsf{read}_{val[m,k]}(\langle nil, \mathsf{lowTS}\rangle)$. This contradicts Lemma 42, and completes the proof. ■

Lemma 44 *For any value v, if the event $\mathsf{ord}(v, ts)$ happens during a Write operation then $ts = ts_v$.*

PROOF: Since $\mathsf{ord}(v, ts)$ happens during a Write operation we know that $v \neq nil$. Assume for a contradiction that $\mathsf{ord}(v, ts_v)$ happens during a Write operation instance, but $ts \neq ts_v$. Consider now the event $\mathsf{ord}(v, ts_v)$. Since v is written at most once, this event happens during some Read operation instance executed by a process p_i. This Read operation instance will execute the *get-latest-val* procedure, which will return v. Moreover, some register in *val*$[i, -]$ contains v and a timestamp ts' that is smaller than ts_v. Since $v \neq nil$, some operation instance invokes write-val with v and ts', which means that R contains $\mathsf{ord}(v, ts')$. This contradicts the fact that ts_v is the smallest timestamp that is part of ordering events for v. ■

Lemma 45 *Let $v \neq v'$ be values in SV_R. If $\mathsf{st}(v, ts)$ happens in R with $ts > ts_{v'}$, and if $\mathsf{st}(v', ts')$ happens in R, then $ts > ts'$.*

PROOF: Assume for a contradiction that $\mathsf{st}(v', ts')$ happens in R with $ts' > ts$. Consider the set S of timestamps:

$$S = \{\hat{ts} : \hat{ts} > ts \wedge \mathsf{ord}(v', \hat{ts}) \in \mathsf{SE}_R^{v'}\}).$$

Then ts' is in S. Let ts_m be the smallest element in S. There are now two cases to consider: (a) $\mathsf{ord}(v', ts_m)$ happens during Write or (b) $\mathsf{ord}(v', ts_m)$ happens during Read. In case (a), we know that $ts_m = ts_{v'}$ by Lemma 44. Since $ts > ts_{v'}$ per assumption, we have a contradiction with the fact that ts_m is an element of S (all elements of S are greater than ts. For case (b), the *get-latest-val* procedure must have returned v' during a Read that generates ts_m as timestamp. This means that *get-latest-val* must have read v' from an element of *val* that has a timestamp ts'' that is smaller than ts_m. We claim that $ts'' > ts$. To show the claim, observe first that $ts'' \neq ts$ because $v \neq v'$. Next, assume for a contradiction that $ts'' < ts$. Since $\mathsf{st}(v, ts)$ happens in R, we know that some process p_i successfully stores $\langle v, ts\rangle$ in the registers *val*$[-, i]$. Moreover, since $ts < ts_m$, these store operations happen before the invocation of *get-latest-val* with ts_m as argument (otherwise, we would contradict Lemma 42). Thus, this invocation of *get-latest-val* would return v instead of v' because $ts'' < ts$, which is a contradiction. Thus we have $ts < ts''$, which means that ts'' is an element of S, and contradicts the fact that ts_m is the smallest element in S. ■

Lemma 46 *If $\mathsf{Read}(v) \in H$, then there exists a timestamp ts such that $\mathsf{st}(v, ts)$ happens during $\mathsf{Read}(v)$ in R.*

PROOF: Follows from the algorithm since the Read operation invokes *check*. ■

Lemma 47 *If $\mathsf{Write}(v)$ is in H and $v \in \mathsf{Obs}_H$, then $\mathsf{ord}(v, ts_v)$ happens during $\mathsf{Write}(v)$ in R and there exists a timestamp ts such that $\mathsf{st}(v, ts)$ happens in R.*

PROOF: If $v \in \mathsf{Commited}_H$ then $\mathsf{Write}(v)$ does not abort, and the lemma holds because both $\mathsf{ord}(v, ts_v)$ and $\mathsf{st}(v, ts)$ happen during $\mathsf{Write}(v)$.

Consider next the case where $v \notin \mathsf{Commited}_H$. Then $v \in \mathsf{Read}_H$. Consider a Read operation instance $\mathsf{Read}(v)$ in H. The existence of $\mathsf{st}(v, ts)$ follows from Lemma 46. Assume next that $\mathsf{ord}(v, ts_v)$ happens during a Read operation instance. Notice that $v \neq nil$ because $\mathsf{Write}(v)$ happens in H. Let ts' be the smallest timestamp such that $\mathsf{st}(v, ts')$ happens in R. We know from Lemma 44 that $\mathsf{st}(v, ts')$ happens during a Read operation instance (otherwise $\mathsf{ord}(v, ts')$ would happen during a Write operation instance, which would mean that $ts' = ts_v$). Consider now the Read operation instance that generates $\mathsf{st}(v, ts')$. The *get-latest-val* procedure must return v during this operation instance. This means that v must have been stored in a register in *val* with a timestamp ts'' that is smaller than ts'. This contradicts that fact that ts' is the smallest timestamp that is part of an ordering event for v. ■

Lemma 48 *If* $\mathsf{oper}_1 \rightarrow_H \mathsf{oper}_2$*, then the events generated during* oper_1 *have smaller timestamps than the events generated during* oper_2*.*

PROOF: Ordering and store events that happen during the same operation instance have the same timestamp. Thus, it is sufficient to prove the lemma for ordering events only. Consider two ordering events $\mathsf{ord}(v, ts)$ and $\mathsf{ord}(v', ts')$ that happen during oper_1 and oper_2 respectively. Let p_i be the process that execute oper_1 and p_j be the process that executes oper_2. During oper_1, p_i writes ts to *ord*[j,i], and p_j reads this register when it generates ts'. From Lemma 42, we conclude that $ts' > ts$. ■

Lemma 49 *For all values* $v, v' \in \mathsf{Obs}_H$*, the following holds:*

$$\mathsf{Write}(v) \rightarrow_H \mathsf{Write}(v') \Rightarrow v < v'$$

PROOF: From Lemma 47, we know that $\mathsf{ord}(v, ts_v)$ happens during $\mathsf{Write}(v)$ and that $\mathsf{ord}(v', ts_{v'})$ happens during $\mathsf{Write}(v')$. From Lemma 48, we conclude that $ts_v < ts_{v'}$, which proves the lemma. ■

Lemma 50 *For all values* $v, v' \in \mathsf{Obs}_H$*, the following holds:*

$$\mathsf{Read}(v) \rightarrow_H \mathsf{Read}(v') \Rightarrow v \leq v'$$

PROOF: Assume for a contradiction that $\mathsf{Read}(v) \rightarrow_H \mathsf{Read}(v')$, yet $v > v'$.

Let $\mathsf{st}(v, ts)$ the store event that happens during $\mathsf{Read}(v)$, and let $\mathsf{st}(v', ts')$ be the store event that happens during $\mathsf{Read}(v')$ (Lemma 46). From Lemma 48, we know that $ts < ts'$. Since $v > v'$, we have that $ts_v > ts_{v'}$, which implies that $ts_{v'} < ts_v < ts < ts'$. This contradicts Lemma 45. ■

Lemma 51 *For all values* $v, v' \in \mathsf{Obs}_H$*, the following holds:*

$$\mathsf{Write}(v) \rightarrow_H \mathsf{Read}(v') \Rightarrow v \leq v'$$

PROOF: Assume for a contradiction that $\mathsf{Write}(v) \rightarrow_H \mathsf{Read}(v')$, yet $v > v'$.

From Lemma 47, we know that $\mathsf{ord}(v, ts_v)$ happens during $\mathsf{Write}(v)$, and that $\mathsf{st}(v, ts)$ happens during R. Similarly, by Lemma 46, $\mathsf{st}(v', ts')$ happens during $\mathsf{Read}(v')$. We know that $ts_v < ts'$ (Lemma 48). There are now two cases to consider: (a) $ts > ts'$ and (b) $ts < ts'$. For case (a), we have that $ts_v < ts' < ts$ which contradicts Lemma 45. For case (b), we have that $ts_{v'} < ts_v < ts < ts'$, which also contradicts Lemma 45. ■

Lemma 52 *For all values* $v, v' \in \mathsf{Obs}_H$*, the following holds:*

$$\mathsf{Read}(v) \rightarrow_H \mathsf{Write}(v') \Rightarrow v < v'$$

PROOF: From Lemma 46, we know that $\mathsf{st}(v, ts)$ happens during $\mathsf{Read}(v)$ for some ts. From Lemma 47, we know that $\mathsf{ord}(v', ts_{v'})$ happens during $\mathsf{Write}(v')$. Lemma 48 implies that $ts < ts_{v'}$, which proves the lemma because $ts_v < ts$ by definition. ■

Proposition 53 *The totally ordered set* $(\mathsf{Obs}_H, <)$ *is a conforming total order.*

PROOF: Follows directly from Lemma 41, Lemma 43, and Lemma 49–52. ■

A.3 Proving Strong Progress

Proposition 54 *A solo operation instance does not abort.*

PROOF: Assume for a contradiction that there is a solo operation instance op that aborts.

Because the registers in *val* and *ord* satisfy strong progress, there are two places where op may abort: (a) the "if" statement in Algorithm 5, line 19 or (b) the "if" statement in Algorithm 5, line 32.

Consider first case (a). *new-ts* is the timestamp of op, and *ts* is the timestamp of some other operation instance op'. Since op runs solo, we must have that $op' \rightarrow op$. Since op' writes its timestamp into a register in *val*, it must have executed inc-ord-ts successfully. In particular, op' must have written its timestamp into one of the registers in *ord* that op reads when it executes generate-ts. However, the value of this register increases monotonically (Lemma 42), which contradicts the fact that *new-ts* $<$ *ts*.

Consider next case (b). *ts* is the timestamp of op, and *ord-ts* is the timestamp of some other operation instance op'. As in case (a), we have that $op' \rightarrow op$. This means that op' must store *ord-ts* in *ord*$[i,j]$ before op executes generate-ts. This contradicts the fact that *ord*$[i,j]$ contains a monotonically increasing sequence of timestamps. ∎

Finally, note that the implementation is clearly wait-free because all its loops are finite (in fact, they are bounded by n). Therefore, from Propositions 39, 53, and 54, we get the following result:

Theorem 55 *Algorithms 4 and 5 is a strictly-linearizable wait-free implementation of a multi-writer multi-reader register from single-writer single-reader ones. It satisfies strong progress if the underlying registers also satisfy strong progress.*