1 Introduction

Lamport [21] introduced and axiomatically defined three related classes of memory called safe, regular and atomic registers. He has also defined *sequential consistency* [19], which can be viewed as another class of memory. In an earlier work [16], we showed that sequentially consistent memory is weaker than atomic memory and incomparable to safe and regular memories[1] and suggested an approach for developing new weakly consistent memories. We advocate the study and use of weakly consistent memories and believe this area represents a rich source of research that may yield insight into the nature of concurrency. Moreover, weakly consistent memories are the subject of active research in the hardware community and serve as a promising bridge between theoretical and practical work. In this paper we present the first formal study, to our knowledge, of a memory strictly weaker than sequential consistency. In this paper, we precisely define, implement and prove correct one member of a family of weakly consistent *causal memories* that we have developed.

Causal memories are based on Lamport's concept [18] of *potential causality.* Potential causality provides a natural ordering on events in a distributed system where processes communicate via message passing. We introduce a similar notion of causality based on reads and writes in a shared memory environment. Causal memories require that reads return values consistent with causally related reads and writes and we say that "reads respect the order of causally related writes." Since causality only partially orders events, readers may disagree on the relative ordering of concurrent writes. This provides independence between concurrent writers and greatly reduces consistency maintenance (synchronization) costs. Various interpretations of this informal definition give rise to a family of related memories; stronger definitions simplify programming while weaker definitions allow more concurrency. In this paper we present one particularly natural definition.

Weakly consistent memories are of more than theoretical interest. Many researchers [4,6,8,13,22,27] seek to provide a shared memory abstraction across distributed hardware as an alternative to the message-passing primitives traditionally available. Such *virtual* or *distributed shared memories* (DSM) are attractive because they allow processes to access local and remote information uniformly, thereby simplifying programming. But traditional consistent shared memory can be costly to implement in a distributed environment. Most existing or proposed DSM implementations resemble multiprocessor cache coherency protocols, but the cost of communicating between processors in a distributed system can be orders of magnitude greater than in a tightly-coupled multiprocessor. Moreover, implementing consistent memory

[1]Sequentially consistent memory is weaker than multi-writer atomic memory and incomparable to both single-writer and multi-writer safe and regular memories.

involves a classical tradeoff between memory access latency and scalability. A simple argument [23,24] can be used to show that no consistent memory can both retain low latency and scalability. This tradeoff between latency and scalability represents a significant efficiency problem since it forces applications to pay the costs of consistency even if they are highly parallel and involve little synchronization. A number of techniques [5,9,17] have been suggested to improve the efficiency of DSM implementations, but all provide only partial remedies to the fundamental problem of latency and scale for consistent memory. Designers of large multiprocessors [11,14] that strive to develop scalable, low-latency parallel computers face this same problem. Recent research [1,3,14,16,23] suggests that a weakening of memory consistency can reduce the costs of consistency maintenance while maintaining a viable "target" model for programmers.

Causal memory is closely related to the ISIS causal broadcast and, thereby, to the notion of causally ordered messages [7]. But causal memory is more than a collection of "locations" updated by causal broadcasts, and there are significant differences in the two models. First, memory has overwrite semantics and messages have queuing semantics. A message recipient can be assured that it will eventually receive all messages that have been sent, but repeated reads cannot guarantee that all values written will be read. "Hidden writes," values overwritten before they are read, are always possible. A message recipient is forced to receive all earlier messages in order to receive a later, interesting message. Since a process may read memory locations in any order it chooses, it may read a value v_1 from location x much later that a value v_2 from location y, even when the write operation that stores v_1 in x is causally before the write of v_2 to y. Such a behavior would violate causal ordering in a message system. However, causal broadcast is similar to the definition of causal memory given here in that any two causally related messages must be received in the same order at all processes.

It is far from clear that there is a "best" kind of shared memory model for use with distributed systems. By exploring the various possibilities, we hope to get a better understanding of what is feasible and what is desirable. This paper carefully examines one such possibility.

2 Shared Memory Systems

In a *distributed system with shared memory* $\mathcal{S}$, processes in $\mathcal{P}$ read and write shared memory locations in $\mathcal{M}$. We assume both $\mathcal{P}$ and $\mathcal{M}$ to be finite. Each process can execute any of a number of *operations* (and the execution of every operation of $\mathcal{S}$ is associated with some process in $\mathcal{P}$). Two operations are externally visible: *read* and *write*. Each read or write operation by process p acts on some named location x and has an associated value v. A write operation by process p, $w_p(x)v$, associates value v with location x, while a read operation, $r_p(x)v$, reports that v is in this location. A special value, $\perp$, which cannot be written, is used to denote the value of a location before any writes have acted on it. We may omit the parameters of a read or write operation when they are not important. For example, $w(x)$ denotes a write by some process acting on location x with an unspecified value.

2.1 Partially Ordered Sets

In a distributed system, processes are usually assumed to execute asynchronously and without reference to a global clock. The operations of an execution are thus not generally ordered sequentially, but are related by some partial order. We use partially ordered sets to represent executions histories. In this subsection, we define terms and notation that will be used throughout the remainder of the paper.

We will need to define various irreflexive[2] partially ordered sets (posets) on operations. A *poset* is a pair $(S, \rightarrow)$ where S is a set and $\rightarrow$ is a relation on S (i.e., $\rightarrow \subseteq S \times S$) that is irreflexive, transitive, and antisymmetric.[3] We normally use infix notation for expressing the relation $\rightarrow$, writing $s \rightarrow s'$ rather than say, $(s, s') \in \rightarrow$.

Sometimes, we need to define a poset that uses a restricted version of a relation defined on another poset. If $\overrightarrow{S} = (S, \underset{S}{\rightarrow})$ is a poset and $T \subseteq S$, we write $\overrightarrow{T} = (T, \underset{S}{\rightarrow})$ to mean $(T, \underset{T}{\rightarrow})$, where $\underset{T}{\rightarrow}$ is the restriction of $\underset{S}{\rightarrow}$ to $T \times T$. This economy of notation allows us to avoid introducing excess relation names such as "$\underset{T}{\rightarrow}$."

For any poset $\overrightarrow{S}$ and element $s \in S$, the *prefix of* s *in* S is the set $\overrightarrow{S}(s) = (S(s), \underset{S}{\rightarrow}) = \{s\} \cup \{s' \in S : s' \underset{S}{\rightarrow} s\}$. Poset $\overrightarrow{S}$ has the *finite prefix property* if, for every operation $s \in S$, $\overrightarrow{S}(s)$ is finite.

A *sequential* poset $\overrightarrow{S} = (S, \underset{S}{\rightarrow})$ has the additional property ("relatedness") that for all $s, s' \in S$, if $s \neq s'$, then either $s \underset{S}{\rightarrow} s'$ or $s' \underset{S}{\rightarrow} s$. We will have occasion to concatenate sequential posets. The following lemma shows this makes sense.

Lemma 2.1: *Let $\overrightarrow{S}$ and $\overrightarrow{T}$ be sequential posets such that $S \cap T = \emptyset$. Let $\underset{ST}{\rightarrow}$ be the union of $\underset{S}{\rightarrow}$, $\underset{T}{\rightarrow}$, and all pairs (s, t) such that $s \in S$ and $t \in T$. Then $(S \cup T, \underset{ST}{\rightarrow})$ is a sequential poset.*

Proof: We show $\underset{ST}{\rightarrow}$ is irreflexive, transitive and antisymmetric, and that for all

[2] A "partial order" is usually taken to be reflexive. Our definition is more convenient for our purposes. Note that a poset by our definition is equivalent to a transitive directed acyclic graph.

[3] We generally use the symbol "$\rightarrow$" for relations. Other symbols (such as $\rightsquigarrow$,$\Rightarrow$) are used to distinguish special relations that have specific meaning for this paper.

$x, y \in S \cup T$, either $x \underset{ST}{\rightarrow} y$ or $y \underset{ST}{\rightarrow} x$. We call the rule for adding pairs of the form (s, t) with $s \in S$ and $t \in T$ the "ST-rule."

- Suppose $\underset{ST}{\rightarrow}$ is reflexive. Then there is a pair (x, x) in $\underset{ST}{\rightarrow}$. But this pair cannot be in $\underset{S}{\rightarrow}$ or $\underset{T}{\rightarrow}$, and it cannot come from pairs added by the ST-rule because $S \cap T = \emptyset$. Thus, $\underset{ST}{\rightarrow}$ is irreflexive.

- Let (x, y) and (y, z) be any two pairs in $\underset{ST}{\rightarrow}$. If all of x, y and z are in S or all are in T, then (x, z) is in $\underset{ST}{\rightarrow}$ by the transitivity of $\underset{S}{\rightarrow}$ or $\underset{T}{\rightarrow}$. Otherwise, either $x \in S$ and $y \in T$ or $y \in S$ and $z \in T$. In the first case, z must also be in T or there could be no pair (y, z) in $\underset{ST}{\rightarrow}$. But then (x, z) is in $\underset{ST}{\rightarrow}$ by the ST-rule. In the second case, x must be in S or there could be no pair (x, y) in $\underset{ST}{\rightarrow}$. But then (x, z) is in $\underset{ST}{\rightarrow}$ by the ST-rule. In all cases, (x, z) is in $\underset{ST}{\rightarrow}$, so $\underset{ST}{\rightarrow}$ is transitive.

- Suppose there are two pairs (x, y) and (y, x) in $\underset{ST}{\rightarrow}$. ($x \neq y$ since $\underset{ST}{\rightarrow}$ is not reflexive). At least on of them must come from the ST-rule or one of $\underset{S}{\rightarrow}$ and $\underset{T}{\rightarrow}$ would not be antisymmetric. But since $S \cap T = \emptyset$, both pairs must come from the ST-rule, implying both x and y are in both S and T, violating the assumption that $S \cap T = \emptyset$.

- Suppose $x, y \in S \cup T$. If both x and y are in S or both are in T, then either (x, y) or (y, x) is in $\underset{ST}{\rightarrow}$ because S and T are sequential posets. If one of x and y is in S and one is in T, then, by the ST-rule, either (x, y) or (y, x) is in $\underset{ST}{\rightarrow}$. Thus all elements of $S \cup T$ are related one way or the other by $\underset{ST}{\rightarrow}$, and hence $(S \cup T, \underset{ST}{\rightarrow})$ is sequential.

$\square$

Note that the lemma does not hold, in general, if $S \cap T \neq \emptyset$. When the intersection is empty, however, the described poset does exist, and we are justified in giving it a name, $\overrightarrow{S} \cdot \overrightarrow{T} = (S \cup T, \underset{ST}{\rightarrow})$, where $\underset{ST}{\rightarrow}$ is defined as in the lemma. For convenience, we let $\overrightarrow{S} \cdot t$ where $t \notin S$ stand for $\overrightarrow{S} \cdot (\{t\}, \emptyset)$, so that we can concatenate single elements to a sequential poset.

A poset $\overrightarrow{T} = (T, \underset{T}{\rightarrow})$ *respects* poset $\overrightarrow{S} = (S, \underset{S}{\rightarrow})$ if for all $t, t' \in T \cap S$, $t \underset{S}{\rightarrow} t'$ implies $t \underset{T}{\rightarrow} t'$. If sequential poset $\overrightarrow{T}$ respects $\overrightarrow{S}$ and $S \subseteq T$, then $\overrightarrow{T}$ *sequences* $\overrightarrow{S}$, and we sometimes say $\overrightarrow{T}$ is a *sequencing* of $\overrightarrow{S}$. Note that T can have more elements than S in this case. When $T = S$ and $\overrightarrow{T}$ sequences $\overrightarrow{S}$, then $\overrightarrow{T}$ is a *topological sort* of $\overrightarrow{S}$. It is well-known that such a topological sort exists for every poset.

2.2 Execution Histories

Consider any finite or infinite execution of a distributed system with shared memory, $\mathcal{S}$. We assume that every operation in the execution is distinct, so that, for example, two successive reads that happen to return the same value are distinguished. Let H be the set of operations that occur in the execution. In general, an *execution history* is just a poset $(H, \underset{H}{\rightarrow})$. We will consider several special types of execution histories.

We assume that processes are sequential (execute one operation after another). The operations of each process in H should thus be sequential. We define *process order*, $\underset{H}{\Rightarrow}$, to be the relation on operations of H defined so that $o \underset{H}{\Rightarrow} o'$ if and only if o and o' are operations of the same process and o' immediately follows o from the point of view of the process (i.e., if o is an operation of p, $o \underset{H}{\Rightarrow} o'$ represents a single step of p). Let H^p be the subset of H consisting of all operations of p in H. Then $(H^p, \overset{+}{\underset{H}{\Rightarrow}})$ is a sequential poset, where $\overset{+}{\underset{H}{\Rightarrow}}$ is the transitive closure of $\underset{H}{\Rightarrow}$. We refer to $\overset{\Rightarrow}{H} = (H, \overset{+}{\underset{H}{\Rightarrow}})$ as the *process execution history.*

For simplicity of exposition, we assume that all writes to a location are uniquely valued (Misra [26] makes a similar assumption). For every read, $r(x)v$, there is at most one write, $w(x)v$, acting on x with value v. The *reads* relation, $\underset{H}{\mapsto}$, on H is defined so that $o \underset{H}{\mapsto} o'$ if and only if o' is a read, $o = r(x)v$, and o is the unique corresponding write, $o = w(x)v$. Note that $(H, \underset{H}{\mapsto})$ is a poset because $\underset{H}{\mapsto}$ is trivially an irreflexive partial order.

In a distributed system, operations are not instantaneous and can overlap in complex ways. A *timing* τ assigns a real time (generally not available to the processes) to the points at which an operation o is invoked, $inv_\tau(o)$, and the time at which it responds, $rsp_\tau(o)$. Timing τ is *proper* if, for all operations $o \in H$, we have $inv_\tau(o) \leq rsp_\tau(o)$. Given a proper timing, we can define the time ordering, $\underset{H^\tau}{\rightarrow}$, that relates operations in H that do not overlap in time to be$\{(s,t)\colon s,t \in H \text{ and } rsp_\tau(s) < inv_\tau(t)\}$. Thus, $s \underset{H^\tau}{\rightarrow} t$ means that operation s finishes before, and hence does not overlap, operation t. It is should be clear that $\underset{H^\tau}{\rightarrow}$ is an irreflexive partial order. A *timed execution history* of an implementation of a system $\mathcal{S}$ with timing τ is $\overrightarrow{H^\tau} = (H, \underset{H^\tau}{\rightarrow})$.

A timed execution history $\overrightarrow{H^\tau} = (H, \underset{H^\tau}{\rightarrow})$ is *well-formed* if τ is proper and

- $\overrightarrow{H^\tau}$ has the finite prefix property,
- $\overrightarrow{H^\tau}$ respects $(H, \overset{+}{\underset{H}{\Rightarrow}})$, and
- there are no operations $o, o' \in H$ such that $o \underset{H^\tau}{\rightarrow} o'$ and $o' \underset{H}{\mapsto} o$.

We assume that, for any system execution, there is a timing τ such that $\overrightarrow{H}^{\tau}$ is well-formed. (If not, the execution has some strange property; for example, the last part of the condition prohibits a read that finishes before the corresponding write has begun.) Note that well-formedness is related to how the set of possible executions is generated and is not something that is normally controllable by the algorithms written for the processes. We do not treat the generation of executions formally here.

Another important type of execution history relates the operations of H causally. A *single causal step* connects either a write to the corresponding read or two successive operations of the same process. Notationally, we write $\overset{1}{\underset{H}{\leadsto}}$ for the union of $\underset{H}{\mapsto}$ and $\underset{H}{\Rightarrow}$. Define the *causality relation* $\underset{H}{\leadsto}$ to be the transitive closure of $\overset{1}{\underset{H}{\leadsto}}$. (We will use the symbol "$\leadsto$" throughout to stand for causality type relations.) The *causal execution history* of an execution is $\widetilde{H} = (H, \underset{H}{\leadsto})$. It is immediate that $\widetilde{H}$ respects $(H, \overset{+}{\underset{H}{\Rightarrow}})$ and $(H, \underset{H}{\mapsto})$. Note that our assumption of the existence of a well-formed timed execution history for every execution implies that $\underset{H}{\leadsto}$ is an irreflexive partial order, as the notation indicates.

The notion of causality that we use is similar to that defined by Lamport [18]; we use write and read operations instead of send and receive actions. In Section 5, we will give a definition of "message causality," which is equivalent to Lamport's definition of potential causality.

We normally expect that a read should return the value of the "most recent" write. "Most recent" is not well-defined for an arbitrary poset, but for sequential posets we have the following definition.

> Let $\overrightarrow{H} = (H, \underset{H}{\rightarrow})$ be an execution history that is sequential, and let o be any operation in H. If there is any write acting on x in $H(o)$, the *last write to location x before o in H* is the unique write acting on x, $w(x)$, such that there is no write $w'(x)$ in H for which $w(x) \underset{H}{\rightarrow} w'(x) \underset{H}{\rightarrow} o$. Define $last(\overrightarrow{H}(o), x)$ to be the value written by o, if o is a write to x, and otherwise to be the last value written to x before o, if defined, or $\perp$, if not.

A natural definition for a sequential execution history to be legal can now be given.

> A sequential execution history $(H, \underset{H}{\rightarrow})$ is *legal* if and only if for every read operation $r(x)v \in H$, $last(\overrightarrow{H}(r(x)v), x) = v$.

Note that the existence of a legal sequential execution history $(H, \underset{H}{\rightarrow})$ implies that for every read in H, there is a corresponding write (except for the special case of a read returning $\perp$).

2.3 Atomic Memory

An atomic memory should behave as if all operations are sequential. A timed execution history $\overrightarrow{H^\tau}$ *satisfies atomicity* if there exists a legal sequencing $\overrightarrow{S}$ of $\overrightarrow{H^\tau}$. A memory is *atomic* if for each execution there is a proper timing τ and a well-formed timed execution history $\overrightarrow{H^\tau}$ that satisfies atomicity.

Atomic memory requires that the order of nonoverlapping operations in H be preserved in the sequential poset $\overrightarrow{S}$. Single-writer atomic memory was formalized by Lamport [20] and the multi-writer case by Misra [26]. Herlihy and Wing's consistency condition [15] for general objects, linearizability, is also precisely equivalent to atomic memory when restricted to objects (registers) that support read and write operations.

2.4 Sequentially Consistent Memory

Sequential consistency, introduced by Lamport [19], is a weakening of atomic memory that does not require operations to "take effect" during their execution interval. Executions must "appear," however, to be the result of some sequential interleaving of the individual process histories. Note that a sequentially consistent memory is weaker than an atomic one because the former need not preserve, in the sequencing, the order of nonoverlapping operations at different processes.

A process execution history $\overrightarrow{\overrightarrow{H}}$ *satisfies sequential consistency* if there exists a legal poset $\overrightarrow{S}$ that sequences $\overrightarrow{\overrightarrow{H}}$. (Note that the legality of $\overrightarrow{S}$ also implies that $\overrightarrow{S}$ respects $\underset{H}{\mapsto}$.) A memory is *sequentially consistent* if each of its process execution histories satisfies sequential consistency. Note that the definition of sequential consistency does not refer to timed execution histories, but depends only on the definition of process order.

2.5 Causal Memory

Atomic and sequentially consistent memories require all processes to agree on a single equivalent sequential history. Causal memory allows each process to "perceive" a different sequential history as long as these histories are consistent with causality.

In describing the other forms of memory, we implicitly assumed that processes did not fail. One of our implementations of causal memory allows processes to fail by stopping. The definition below thus refers to "correct" processes, those that do not fail in the execution. In the case that all processes are assumed to be non-failing, the definition below compares directly with the other forms of memory.

When processes may fail and messages can be dropped, some writes may be completed without any correct process knowing of them. Our definition will only require correct processes to schedule writes that are known. An operation o is *known* to

$$\begin{array}{lll} p: & w(x)1 & r(x)2 \\ q: & w(x)2 & r(x)1 \end{array}$$

Figure 1: Causal but not Sequentially Consistent

process p if o is an operation of p or there is an operation o' of p such that $o \underset{H}{\leadsto} o'$. Let H^{known} be the set of all *write* operations in H that are known to some correct process. The we will require that process p schedule all of its own operations, H^p and all known writes, H^{known}. Let $H_p = H^p \cup H^{known}$.

We now define correctness for an implementation of causal memory. A causal execution history $\overset{\leadsto}{H}$ *satisfies causality* if, for each process $p \in \mathcal{P}$, there is a legal poset $\overrightarrow{S_p} = (S_p, \underset{S_p}{\rightarrow})$ such that $S_p \subseteq H$ and $\overrightarrow{S_p}$ sequences $(H_p, \underset{H}{\leadsto})$ and respects $\overset{\leadsto}{H}$. A memory is *causal* if each of its causal execution histories satisfies causality.

The definition of causal requires that all known writes be scheduled. But a write does not become known until its location is read (or a causally later write is read). In general, we might want an algorithm to schedule writes as they are learned about (say from a message), rather than waiting until they are read. This "early scheduling" is allowed by the definition because $\overrightarrow{S_p}$ is not a topological sort of $(H_p, \underset{H}{\leadsto})$ and so can contain additional operations. Note that these additional operations must be from H, and they must still be ordered with respect to $\underset{H}{\leadsto}$.

Causal memory derives its flexibility by allowing each process a different sequential poset S_p and by eliminating reads of other processes from that history. In effect, concurrent writes may appear to occur in different orders to different processes. Figure 1 shows a correct execution on causal memory that is not sequentially consistent. The sequential posets demonstrating that the execution satisfies causality are $\overrightarrow{S_p}$ defined so that $w(x)1 \underset{S_p}{\rightarrow} w(x)2 \underset{S_p}{\rightarrow} r(x)2$ and $\overrightarrow{S_q}$ defined by $w(x)2 \underset{S_q}{\rightarrow} w(x)1 \underset{S_q}{\rightarrow} r(x)1$.

This definition of causal memory implies a liveness requirement. Since $\overrightarrow{S_p}$ must contain the writes of all correct processes in H (because each is trivially known to a correct process), each such write must appear in some finite prefix of $\overrightarrow{S_p}$. Thus, the "effects" of the writes of correct processes cannot be indefinitely delayed.

Other forms of weakly consistent memory can be defined using this formalism including *PRAM* (Pipelined RAM) defined by Lipton and Sandberg [23] and *slow memory* given by Hutto and Ahamad [16].

```
COORDINATOR
while (¬done)
   wait until (∀i complete_i)
   foreach i
      complete_i := false
   wait until (∀i changed_i)
   if (converged())
      done := true
   foreach i
      changed_i := false
```

```
WORKER_i
while (¬done)
   t_i := (b_i − Σ_{j=1}^{i−1} a_{i,j} x_j − Σ_{j=i+1}^{n} a_{i,j} x_j) / a_{i,i}
   complete_i := true
      wait until (¬complete_i)
   x_i := t_i
changed_i := true
      wait until (¬changed_i)
```

Figure 2: Synchronous Iterative Linear Solver on Causal Memory

3 Programming with Causal Memory

Causal memory can be easily programmed. We show a synchronous iterative linear equation solver that works correctly on atomic memory and show it correct on causal memory as well.

Consider a parallel iterative algorithm that solves $Ax = b$, where A is a known matrix, b is a known vector, and x is the vector that is the solution. We use x_i^{k+1} to represent the value of the ith component of x in phase $k+1$: $x_i^{k+1} = \left(b_i - \sum_{j=1}^{i-1} a_{i,j}\, x_j^k - \sum_{j=i+1}^{n} a_{i,j}\, x_j^k\right) \Big/ a_{i,i}$. Computing x_i^{k+1} requires access to all x_j^k ($i \neq j$) from the previous iteration.

Figure 2 shows the solver. All booleans are initially *false* and "**wait until** (B)" means"**while** $(\neg B)$ *skip*". Worker$_i$ computes x_i. The coordinator tests for convergence and synchronizes each Worker$_i$ twice per iteration using a barrier technique: before reading the various x_j from phase k and before writing x_i for phase $k+1$. (By having workers read alternately from x_i and t_i, we could eliminate the first synchronization.)

The protocol in Figure 2 is easily shown to be correct on atomic memory. To show that the protocol is correct on causal memory, we argue that the reads of each x_i in phase k return the value written in phase $k-1$, just as with atomic memory. A similar argument holds for the handshake bits ($complete_i$ and $changed_i$). Let $w_i^k(x_i)v$ denote a write of x_i by p_i in phase k. Consider the causal relations established between p_j's write of x_j in phase $k-1$ and p_i's read of x_j in phase k in some execution H of the code in Figure 2. (Subscript 'c' denotes the coordinator.)

$$
\begin{array}{ll}
(1) & w_j^{k-1}(x_j)v \underset{H}{\Rightarrow} w_j^{k-1}(changed_j)true \underset{H}{\mapsto} r_c^{k-1}(changed_j)true \\
(2) & r_c^{k-1}(changed_j)true \overset{+}{\underset{H}{\Rightarrow}} w_c^{k-1}(changed_i)false \\
(3) & w_c^{k-1}(changed_i)false \underset{H}{\mapsto} r_i^{k-1}(changed_i)false \overset{+}{\underset{H}{\Rightarrow}} r_i^k(x_j)v
\end{array}
$$

Above, (1) holds because the two writes are consecutive operations of p_j and the coordinator eventually notices that p_j has copied t_j to x_j. (2) is the key to the argument and holds because the coordinator must read *changed* true from *all* workers (j in particular) before setting *changed* false for all workers (i in particular) and initiating the next phase. (3) holds because p_i must eventually see $changed_i$ false and begin the next phase. At the beginning of phase k, p_i reads all components of x (except x_i) and, in particular, x_j. Taken together (1)–(3) imply that $w_j^{k-1}(x_j)v \underset{H}{\rightsquigarrow} r_i^k(x_j)v$. Since p_j is the only process to write to x_j, and since all prior writes to x_j by p_j causally precede $w_j^{k-1}(x_j)v$, and since all subsequent writes to x_j by p_j causally follow $w_j^{k-1}(x_j)v$, p_i's phase k read of x_j *must* return p_j's phase $k-1$ write. Since we assumed an arbitrary i and j this argument shows that all reads of x in the computation return the value computed in the previous iteration, the same value returned when the computation is executed on atomic memory.

4 The Simple Algorithm

This section presents a simple and efficient implementation of causal memory using message passing primitives. We then prove its correctness with respect to the formal definition introduced in Section 2. This implementation uses an adaptation of *vector timestamps* [12,25]. It requires ordered ("FIFO") channels and is not fault-tolerant. That is, the implementation assumes no processes fail and that all messages sent are delivered exactly once and in the order sent. Since all process are correct, all writes are known, so for this section, H^{known} is the set of all writes. In the next section, we present a more complex implementation that tolerates stopping failures and lossy communication channels.

In our implementation, each process p maintains and accesses a local (private) copy M of the abstract shared causal memory $\mathcal{M}$. A process reads from memory simply by reading from this copy of memory. In addition to M, process p maintains a vector clock t, which it uses to timestamp outgoing messages. This is a vector of natural numbers, one for each process in the system. Informally, $t[q]$ is the number of writes by q of which p is aware. Two vectors can be compared by comparing their components. Vector t_1 is *less than or equal* to t_2 ($t_1 \leq t_2$) if each of t_1's components is less than or equal to t_2's corresponding component; t_1 is *less than* t_2 ($t_1 < t_2$) if it is less than or equal to t_2 and is not equal to t_2. Before writing a value v to location x, a process increments its own component of its vector clock t. It then writes to its local copy of memory and informs the other processes of its write by sending them $\langle x, v, t\rangle$. Such a message is called a *write message* and has *timestamp* t. We sometimes informally talk of a process receiving a write instead of a write message. Upon receiving a write message, a process updates the component of its vector clock that corresponds to the sending process. The vector timestamps on these messages are used to determine which writes can be applied to the process's memory and which must be left pending.

The implementation is shown in Figure 3. It consists of an initialization routine and three basic operations. The *read* operation is executed whenever a read of a location x by p is invoked. The value stored in $M[x]$ is returned to p. The *write* operation is executed whenever a write to a location x by p is invoked. Process p increments $t[p]$, writes to the value $M[x]$, and then sends an appropriate write message to all other processes. The *receive* operation is executed when a process p receives such a message. Process p first updates the appropriate component of its logical clock; recall that the component for q counts the number of q's writes of which p is aware. The message is then added to p's set of pending writes. A write message must remain pending as long as its vector timestamp reflects some write of which p is not aware. This can be detected by comparing p's vector clock to the timestamp of the write; the write is processed only if its timestamp is less than or equal to p's vector clock. Each time a write message is received, all pending messages are checked; those that can be processed are applied to memory in an order consistent with their timestamps. Note that the implementation admits the execution shown in Figure 1. Thus, our implementation is neither atomic nor sequentially consistent.

In order to prove the correctness of the implementation, we introduce the following notation: if o is an operation of a process q, let $ts(o)$, the *timestamp* of o, be the value of q's vector clock immediately after o completes. Note that for a write operation o, $ts(o)$ is the same as the timestamp included in the write message.

The following two lemmas are used in the proof of correctness. The first asserts that the causal relation $\underset{H}{\leadsto}$ is reflected in vector timestamps:

Lemma 4.1: *Suppose that $\widetilde{H} = (H, \underset{H}{\leadsto})$ is a causal execution history of the implementation and let o and o' be a two operations such that $o \underset{H}{\leadsto} o'$. Then $ts(o) \leq ts(o')$. Furthermore, if o' is a write operation, then $ts(o) < ts(o')$.*

Proof: Since $o \underset{H}{\leadsto} o'$, there is some shortest list of operations $o = o_0, o_1, \ldots, o_k = o'$, where $k > 0$ and the operations are connected by single causal steps, i.e., $o_i \overset{1}{\underset{H}{\leadsto}} o_{i+1}$ for $0 \leq i < k$. We continue by induction on k.

```
/* Initialization: */
    foreach x ∈ 𝓜 do
        M[x] := ⊥
    foreach q ∈ 𝓟 do
        t[q] := 0
    pending := ∅

/* Read action: to read from x */
    return(M[x])

/* Write action: for p to write v to x */
    t[p] := t[p] + 1
    M[x] := v
    send ⟨x, v, t⟩ to all others

/* Receive action: upon receipt of ⟨x_q, v_q, t_q⟩ from q */
    t[q] := t_q[q]
    pending := pending ∪ {⟨x_q, v_q, t_q⟩}
    foreach ⟨x, v, s⟩ ∈ pending do
        if s ≤ t then
            pending := pending − {⟨x, v, s⟩}
            process := process ∪ {⟨x, v, s⟩}
    Let T⃗ be a topological sort of process by vector timestamp
    foreach ⟨x, v, s⟩ ∈ process do   /* In order by T⃗ */
        M[x] := v          /* This applies the write */
        remove ⟨x, v, s⟩ from process
```

Figure 3: Simple Implementation of Causal Memory

For the basis, assume $k = 1$. Then either $o \underset{H}{\Rightarrow} o'$ or $o \underset{H}{\mapsto} o'$. If $o \underset{H}{\Rightarrow} o'$, then o and o' are operations of the same process p. Since no process ever decrements any component of its vector clock, $ts(o)$ must be less than or equal to $ts(o')$. Furthermore, if o' is a write operation, then p increments its local component before executing the write, so $ts(o) < ts(o')$.

If $o \underset{H}{\mapsto} o'$, then o is a write operation, say $w_q(x)v$, and o' is a corresponding read, say $r_p(x)v$. Note that the write message associated with o includes the timestamp $ts(o)$. It is clear that p cannot read v from x until p has applied the write to its memory. Process p will not apply the write until its timestamp is greater than or equal to $ts(o)$. Since no component of p's timestamp is ever decremented, it is still greater than or equal to $ts(o)$ when it reads v, so $ts(o) \leq ts(o')$.

Now assume $k > 1$ and the lemma holds for lists shorter than k in length. Then $ts(o) \leq ts(o_{k-1})$ and $ts(o_{k-1}) \leq ts(o')$ by the induction hypothesis, so $ts(o) \leq ts(o')$. If o' is a write operation, then o_{k-1} is an operation of p, so $ts(o_{k-1}) < ts(o')$ by the induction hypothesis and hence $ts(o) < ts(o')$, as required.

□

The next lemma is used to show the liveness of the implementation:

Lemma 4.2: *Let* $\overset{\rightsquigarrow}{H}$ *be a causal execution history of the implementation and suppose that* o *is a write operation of process* p*. Then each process* q *eventually applies* o *to its memory.*

Proof: Suppose that $ts(o) = t$. First, for arbitrary processes p and q, write o by p will eventually be received (via a message) by q, since message passing is reliable. Further, since timestamps are finite vectors of natural numbers and every write is assigned a distinct timestamp, there can only be a finite number of writes that precede o, that is, that have timestamps less than t. By the argument above, all of these writes will eventually be received (in FIFO order from the senders) by q. Once all writes preceding o are received by q, these writes, including o, are applied to q's memory in some topological order. □

We can now assert the correctness of the implementation:

Theorem 4.3: *Let* $\overset{\rightsquigarrow}{H}$ *be a causal execution history of the implementation. Then* $\overset{\rightsquigarrow}{H}$ *satisfies causality.*

Proof: To prove this theorem, it is necessary to show that for each process p there is a legal sequential poset $\overrightarrow{S_p} = (S_p, \underset{S_p}{\rightarrow})$ with $H_p \subseteq S_p \subseteq H$ such that $\overrightarrow{S_p}$ respects $\overset{\rightsquigarrow}{H}$.

The sequential poset $\overrightarrow{S_p}$ for p is generated by simply concatenating all writes as they are applied to p's memory and all reads as they occur. $\overrightarrow{S_p}$ is a validly defined by Lemma 2.1 because each write operation has exactly one message and hence is concatenated just once. By Lemma 4.2, $\overrightarrow{S_p}$ includes all write operations in H, and thus all of H_p. $\overrightarrow{S_p}$ is clearly legal because all reads and writes apply directly to p's copy of memory and each read thus reads the value most recently written. It remains to be seen that $\overrightarrow{S_p}$ respects $\overset{\rightsquigarrow}{H}$.

Let o_1 and o_2 be two operations in S_p such that $o_1 \underset{H}{\rightsquigarrow} o_2$. One of the following five cases must hold:

- Both o_1 and o_2 are operations by p. Note $o_1 \underset{H}{\rightsquigarrow} o_2$ implies that o_1 is applied by p before o_2. Since p's operations are always scheduled immediately when they are applied, $o_1 \underset{S_p}{\rightarrow} o_2$.

- o_1 is a write by another process (call it q) and o_2 is a read operation by p. By Lemma 4.1, $ts(o_1) \leq ts(o_2)$. In particular, $ts(o_1)[q] \leq ts(o_2)[q]$, so since messages are received from q in FIFO order, p receives o_1 before o_2 takes place. Consider the time at which p sets its vector timestamp to $ts(o_2)$ (this happens during some receive occurring *before* o_2 executes). The write o_1 is applied no later than this time and thus is sequenced before o_2, and hence $o_1 \underset{S_p}{\rightarrow} o_2$.

- o_1 is a write by another process and o_2 is a write operation by p. By the definition of $\underset{H}{\rightsquigarrow}$, this can only be the case if there is a read operation o' by p such that $o_1 \underset{H}{\rightsquigarrow} o' \underset{H}{\rightsquigarrow} o_2$. By applying the two cases above, we conclude that $o_1 \underset{S_p}{\rightarrow} o' \underset{S_p}{\rightarrow} s_2$, so $o_1 \underset{S_p}{\rightarrow} o_2$.

- o_1 is an operation by p and o_2 is a write by another process. By Lemma 4.1, $ts(o_1) < ts(o_2)$. Since the value of p's vector clock is nondecreasing, when o_1 occurs, o_2 cannot have been applied by p because o_2's timestamp is too high. Thus, o_2 is sequenced after o_1, so $o_1 \underset{S_p}{\rightarrow} o_2$.

- o_1 and o_2 are both writes by processes other than p. By Lemma 4.1, $ts(o_1) < ts(o_2)$. Consider the point at which p moves o_2 from *pending* to *process*. Let r be the process writing o_1. Since $ts(o_1)[r] \leq ts(o_2)[r] \leq t[r]$, where t is p's vector clock value at this point, the message containing o_1 has been received. If o_1 has already been applied at this point, then $o_1 \underset{S_p}{\rightarrow} o_2$. If o_1 has not been applied at this point, then, because $ts(o_1) < ts(o_2)$, it must also be in *process*. Since $ts(o_1) < ts(o_2)$, p applied o_1 before o_2 (the topological sort preserves timestamp order). Thus, $o_1 \underset{S_p}{\rightarrow} o_2$.

In all cases, $o_1 \underset{S_p}{\rightarrow} o_2$, so the proof is complete. □

5 The Robust Algorithm

In this section we present a fault-tolerant implementation of causal memory. We call this implementation the Robust algorithm to distinguish it from the Simple (but fragile) algorithm presented in the last section. The Robust algorithm is very fault-tolerant but requires large messages (the message size is $O(mn^2)$ bits, where $m = |\mathcal{M}|$, $n = |\mathcal{P}|$, and we assume memory values and the components of vector timestamp need only a constant number of bits each). The Robust algorithm tolerates unreliable channels (lost, duplicate or out-of-order messages) and correct processes may continue execution despite up to $n-1$ stopping failures. Processes never block on reads or writes, so the Robust algorithm is wait-free. While the algorithm potentially requires large messages, processes may send and receive according to any live schedule. Thus, sends and receives may be scheduled to the advantage of the process and of the system.

5.1 Causal Memory with Failures

We consider computations in which process might fail by stopping (without announcement). A process *fails* in a computation if it takes only a finite number of steps (we do not consider any other form of process failure).

In addition to process failures, we tolerate failures in the message passing system. Messages may be reordered, duplicated or dropped (but not corrupted). We do require the following liveness condition on channels:

> If process p sends an infinite number of distinct messages to q, then an infinite number of distinct messages are delivered to q, assuming q attempts to receive an infinite number of times.

Note that this does not prohibit the channel from dropping an infinite number of messages. The restriction on distinct messages prohibits the channel from repeating a fixed message an infinite number of times to satisfy the infinite delivery requirement. We assume that the send and receive operations of our algorithms to continue to execute forever, even when no more reads or writes are required.

5.2 Description of the Algorithm

As in the Simple algorithm, each process maintains a complete local copy of the shared causal memory M and a vector timestamp t. No *process* or *pending* sets are used, but the Robust algorithm keeps a set of location name, value, vector timestamp triples: $Z = \{\langle x_1, v_1, t_1\rangle, \langle x_2, v_2, t_2\rangle, \ldots, \langle x_k, v_k, t_k\rangle\}$. The restriction $Z[x]$ of Z contains all the triples in Z with first element x. (Essentially, $Z[x]$ holds all the values written to x by the most recent concurrent writes the process has learned about.) We will show that $Z[x]$ contains at most n triples, where n is the number of processes. When a process receives a message (containing the Z of some other process), it updates its copy of Z and modifies M appropriately. We will show that this implements causal memory correctly.

Besides initialization, there are four basic operations: read, write, send and receive. A read operation, as for the Simple algorithm, simply returns the value of the

/* Initialization: */

> $Z := \emptyset$
> **foreach** $x \in \mathcal{M}$ **do**
> $M[x] := \perp$
> **foreach** $q \in \mathcal{P}$ **do**
> $t[q] := 0$

/* Read action: to read from x: */

> **return**($M[x]$)

/* Write action: for p to write v to x: */

> $t[p] := t[p] + 1$
> $M[x] := v$
> /* Delete all triples in Z *covered* by $\langle x, v, t\rangle$ */
> $Z := (Z - Z[x]) \cup \{\langle x, v, t\rangle\}$

/* Send action: by any live schedule */

> **send** $\langle Z, t\rangle$ **to** all others

/* Receive action: by any live schedule, on receipt of $\langle Z_q, t_q\rangle$ from q */

> **foreach** $r \in \mathcal{P}$ **do**
> $t[r] := \max(t[r], t_q[r])$
> /* Delete all triples in Z_q *dominated* by Z */
> $Z'_q := Z_q - \{\langle x_q, v_q, s_q\rangle \in Z_q \colon \exists \langle x, v, s\rangle \in Z$ **such that** $s_q \leq s\}$
> /* Choose some fixed topological sort, $\overrightarrow{T}$, of Z'_q by timestamps */
> **foreach** $x \in \mathcal{M}$ **do**
>   **if** $Z'_q[x] \neq \emptyset$ **then**
>   **foreach** $\langle x, v, s\rangle \in Z[x]$**do**   /* In order by $\overrightarrow{T}$ */
> $M[x] := v$ /* This applies the write */
> /* Delete triples in Z *covered* by Z'_q */
> $Z' := Z - \{\langle x, v, s\rangle \in Z \colon \exists \langle x, v_q, s_q\rangle \in Z'_q$ **such that** $s < s_q\}$
> $Z := Z'_q \cup Z'$

Figure 4: Robust Implementation of Causal Memory

appropriate location from M, the local copy of memory. The write operation is almost as simple, incrementing the process's own component of t (the local copy of the vector timestamp) and storing the value in M. Then Z is updated, which will be explained below. Notice that a write does not execute a send (as in the Simple algorithm). Intuitively, this is because message delivery is not guaranteed; a message sent by a single send operation associated with a write might never be received. Instead, there is a separate send operation that sends the local Z value and timestamp to all other processes whenever it is invoked. We make no requirements on when sends should be invoked except that a process must continue to send (i.e., a correct process will send an infinite number of times).

Before describing the purpose of the Z set, we need some terminology. Each triple is originally created by a write operation that writes the given value to the given location and has the stated timestamp. Thus, we can speak of the write operation that corresponds to each triple in Z. Informally, we use Z to refer to the set of operations associated with Z and sometimes refer to triples as if they were the corresponding write.

As in the Simple algorithm, each operation o has a timestamp $ts(o)$, which is the local value of t of the process executing o when the operation is completed. For any operations o and o', we say o *dominates* o' if $ts(o') \leq ts(o)$. We also apply the term "dominates" to timestamps themselves and say a timestamp t dominates an operation o when $ts(o) \leq t$. A set of operations O dominates set O' if every element of O' is dominated by some element of O. If o and o' are writes to the same location and o dominates o', then we say that write o *covers* write o'. We extend the cover relation to sets of operations as follows: a set of operations O covers set O' if for every write operation $o' \in O'$ there is a write operation $o \in O$ such that o covers o'.

In the Simple algorithm, writes were accumulated locally until they could be applied. Effectively, the Robust algorithm applies writes as soon as it learns of them from a message. Informally, we say a process *learns about a write w* when it first receives a message that contains an operation dominating w. We say a write has been *scheduled* when it has been effectively applied. By "effectively applied" we mean the action of the write has been taken into account, even though the write itself might not be applied (because a covering write is actually applied). The important thing is that a write will never be applied after it has been effectively applied. Informally, the invariant we want to maintain is that, just after operation o by process p is executed, the following holds:

- local timestamp t dominates all operations p has learned about,
- all the writes that p has learned about have been scheduled, and
- the Z set covers all writes that p has learned about.

We now return to the discussion of how a write updates Z. Consider a write operation by p acting on location x. All triples corresponding to writes to x are deleted from p's Z set and then the current write triple is added. Since the current timestamp is strictly greater than all operations that p has learned about, it is clear that the new write covers all the operations that were deleted and, by transitivity, all writes to x that p has learned about. Writes to other locations in Z are not affected. Thus, Z still covers all writes that p has learned about.

We now consider the most complicated operation of the algorithm, the receive. A receive message has the form $\langle Z_q, t_q \rangle$. The first part of the receive operation updates t to the componentwise maximum of the previous value of t and t_q. Since the previous value of t dominates all operations that p learned about previously and t_q dominates all those covered by Z_q, the new value of t clearly dominates all the writes that p has learned about now (including those in Z_q itself). Thus, the first condition of the invariant is obtained.

The next action of the receive operation sets Z'_q to Z_q, minus those operations dominated by Z. Thus, Z'_q dominates all the new writes about which p is to learn but does not contain any writes that p has already scheduled. As will be shown in the proof, the topological sort $\overrightarrow{T}$ (treated as a poset on the writes corresponding to the triples of Z'_q) respects $\overset{\leadsto}{H}$, so it gives a valid order in which to apply the new writes. The proof will show that $\overrightarrow{T}$ can be extended to include new writes that are dominated by Z'_q but are not explicitly represented in Z'_q. Thus, in applying $\overrightarrow{T}$, we achieve the second condition of the invariant and effectively apply all writes p has now learned about. Before Z'_q is merged with the current value of the Z set, triples in Z that are dominated by Z'_q are deleted. This is purely an efficiency step and is not required for correctness. Since the reduced set still dominates all writes p has now learned about, the final condition of the invariant is maintained.

5.3 Some New Notation

To prove the correctness of the Robust algorithm we require some new notation. In addition to the read and write operations described previously, execution histories may also contain *send* and *receive* operations. A send operation by process p, $send_{p,q}\langle Z, t\rangle$, initiates a transfer of the message $\langle Z, t\rangle$ from process p to process q while a receive operation by process q, $receive_{p,q}\langle Z, t\rangle$, completes transfer of the message $\langle Z, t\rangle$ from process q to process p. We omit the message contents and the secondary process where appropriate. For example, $send_{p,q}\langle Z, t\rangle$, $send_{p,q}$, and $send_p$ may all denote the same send operation but with decreasing specificity.

We have previously defined the causality relation $\underset{H}{\leadsto}$ on the operations in an execution history H as the transitive closure of the process order relation $\underset{H}{\Rightarrow}$ and the reads $\underset{H}{\mapsto}$ relation. In analogy to the reads relation, we define the *receives* relation on sends and receives in H. As with writes in H, we assume all send operations are unique. The receives relation pairs each receive operation $o_R = receive_{p,q}$ in H with the unique send operation $o_S = send_{q,p}$, also in H, from which it receives, and we write $o_S \underset{H}{\hookrightarrow} o_R$. Similarly to the definition of single step causality ($\overset{1}{\underset{H}{\rightsquigarrow}}$), we define single step message causality to be $\overset{1}{\underset{H^m}{\rightsquigarrow}}$, the union of $\underset{H}{\hookrightarrow}$ and $\underset{H}{\Rightarrow}$. *Message causality* is then, $\underset{H^m}{\rightsquigarrow}$, the transitive closure of $\overset{1}{\underset{H^m}{\rightsquigarrow}}$. The *message-causal execution history* is $\overset{\rightsquigarrow}{H^m} = (H, \underset{H^m}{\rightsquigarrow})$.

According to the definition of prefix, for any operation o of process p, $\overset{\rightsquigarrow}{H^m}(o) = (H^m(o), \underset{H^m}{\rightsquigarrow})$. Note that the writes in $H^m(o)$ are exactly the writes that p has learned about after executing o. Define $H_p^m(o)$ to be all operations of p and all writes in $H^m(o)$. Then let $\overset{\rightsquigarrow}{H_p^m}(o) = (H_p^m(o), \underset{H^m}{\rightsquigarrow})$. For any operation o of process p in H, we will define a legal poset $\overrightarrow{S_p^m}(o)$, which is a topological sort of $\overset{\rightsquigarrow}{H_p^m}(o)$. Taking the limit of $\overrightarrow{S_p^m}(o)$ over all the operations of p will give a sequential poset $\overrightarrow{S_p^m}$, which we will show sequences $(H_p, \underset{H^m}{\rightsquigarrow})$. From this, we can show $\overrightarrow{S_p^m}$ sequences $(H_p, \underset{H}{\rightsquigarrow})$.

The *memory projection* $M(\overrightarrow{S_p^m}(o))$ of $\overrightarrow{S_p^m}(o)$ is the memory defined by $M(\overrightarrow{S_p^m}(o))[x] = last(\overrightarrow{S_p^m}(o), x)$ for each x. This relates the sequential poset $\overrightarrow{S_p^m}$ to a state of memory. We refer to process p's copy of memory generally as M_p. We use the notation $M_\perp$ to refer to the initial state of memory. Initially, $M_p = M_\perp$ for all processes p. For any operation o of p, $M_p(o)$ is the state of p's copy of memory immediately after operation o is executed. We will show $M(\overrightarrow{S_p^m}(o)) = M_p(o)$.

Define $O(Z)$ to the the set of write operations corresponding to the triples in Z. For each process p and operation o of p, let $Z_p(o)$ be $O(Z)$, where Z is the value of p's Z set immediately after operation o. Initially, $Z_p = \emptyset$. We will show that for all operations o, $S_p^m(o)$ covers $Z_p(o)$.

5.4 Correctness

To prove that the Robust algorithm correctly implements a causal memory we must show that it only admits causal execution histories that satisfy causality. In particular, for each execution history $\overset{\rightsquigarrow}{H}$ allowed by the Robust algorithm and for every correct process p there exists a legal poset $\overrightarrow{S_p}$ that sequences $(H_p, \underset{H}{\rightsquigarrow})$ and that respects $\overset{\rightsquigarrow}{H}$. Recall that H_p includes all operations of p and all writes known to correct processes.

In the proof of the critical lemma (5.3), we will need a slightly stronger fact about timestamps than we had in Lemma 4.1 for the Simple algorithm. In particular, we will need to know that the partial order on writes based on timestamps is identical to the partial order given by $\underset{H^m}{\leadsto}$. Recall that the timestamp of any operation is the value of the timestamp of the executing process immediately after the operation completes.

Lemma 5.1: *Suppose that $\overset{\leadsto}{H^m}$ is a message-causal execution history of the Robust algorithm and let o and o' be any two operations.*

- *If $o \underset{H^m}{\leadsto} o'$, then $ts(o) \leq ts(o')$. Furthermore, if o' is a write operation, then $o \underset{H^m}{\leadsto} o'$ implies $ts(o) < ts(o')$.*
- *If o is a write operation and $o \neq o'$, then $ts(o) \leq ts(o')$ implies $o \underset{H^m}{\leadsto} o'$.*

Proof:

- Assume $o \underset{H^m}{\leadsto} o'$. Since $o \underset{H^m}{\leadsto} o'$, there is some shortest list of operations $o = o_0, o_1, \ldots, o_k = o'$ for $k > 0$ such that $o_i \overset{1}{\underset{H^m}{\leadsto}} o_{i+1}$ for $0 \leq i < k$. We continue by induction on k.

  For the basis, assume $k = 1$. Then either $o \underset{H}{\Rightarrow} o'$ or $o \underset{H}{\hookrightarrow} o'$. If $o \underset{H}{\Rightarrow} o'$, then o and o' are operations of the same process p. Since no process ever decrements any component of its vector clock, $ts(o)$ must be less than or equal to $ts(o')$. Furthermore, if o' is a write operation, then p increments its local component before executing the write, so $ts(o) < ts(o')$.

  If $o \underset{H}{\hookrightarrow} o'$, then o is a send operation by some process q, and o' is receive operation by some process p, $p \neq q$. Since the algorithm sets p's value of t to the componentwise maximum of its current value and the value of $t_q = ts(o)$ of q (in the received message), clearly $ts(o) \leq ts(o')$.

  Now assume $k > 1$ and the lemma holds for lists shorter than k in length. Then $ts(o) \leq ts(o_{k-1})$ and $ts(o_{k-1}) \leq ts(o')$ by the induction hypothesis, so $ts(o) \leq ts(o')$. If o' is a write operation, then o_{k-1} is an operation of p, so $ts(o_{k-1}) < ts(o')$ by the induction hypothesis and hence $ts(o) < ts(o')$, as required.

- Assume o is a write operation, $o \neq o'$, and that $ts(o) \leq ts(o')$. Let p be the process executing write o. Let O be the set of operations including o' and all $o_1 \in H$ such that $ts(o)[p] \leq ts(o_1)[p]$ and $o_1 \underset{H^m}{\leadsto} o'$. (The notation $ts(o)[p]$ refers to p's component of $ts(o)$.) Since O is finite (by the finite prefix property) and nonempty, there is some $o_1 \in O$ such that, for all $o_2 \in H$, $o_2 \overset{1}{\underset{H^m}{\leadsto}} o_1$ implies $ts(o_2)[p] < ts(o)[p]$. Let o_1 be such an operation, and let the process executing o_1 be q.

  Suppose $q \neq p$. Note that o_1 is not the first operation of q because then $ts(o_1)[p] = 0$, while $ts(o)[p]$ is at least 1. Thus, there is an operation o_2 of process q such that $o_2 \underset{H}{\Rightarrow} o_1$. By the choice of o_1, $ts(o_2)[p] < ts(o)[p] \leq ts(o_1)[p]$. Since $q \neq p$, o_1 must be a receive (only writes and receives change the local timestamp, and a write would only change q's component). Let o_3 be the unique send operation such that $o_3 \underset{H}{\hookrightarrow} o_1$. Again by the choice of o_1, $ts(o_3)[p] <$

$ts(o)[p] \le ts(o_1)[p]$. But the receive operation sets $ts(o_1)[p]$ to the maximum of $ts(o_2)[p]$ and $ts(o_3)[p]$, both of which are less than $ts(o)[p]$. This contradicts the assumption that $ts(o)[p] \le ts(o_1)[p]$, so it must be that $q = p$. Since $ts(o)[p] \le ts(o_1)[p]$ and timestamp component values are nondecreasing, either $o = o_1$, or $o \overset{+}{\underset{H}{\Rightarrow}} o_1$ and hence $o \underset{H^m}{\leadsto} o_1$. In either case by the transitivity of $\underset{H^m}{\leadsto}$ and the assumption that $o \neq o'$, $o \underset{H^m}{\leadsto} o'$, as required.

□

In the critical lemma (5.3), we will need to show that every receive operation preserves causal correctness of $\overrightarrow{\widetilde{H_p^m}}(o)$ by properly maintaining a sequencing $\overrightarrow{S_p^m}(o)$ of $\overrightarrow{\widetilde{H_p^m}}(o)$ at each step. We will construct $\overrightarrow{S_p^m}(o)$ inductively, concatenating not only the new writes in Z_q for each receive operation, but also new writes that are covered, but not explicitly represented in Z_q. For this, the following easy lemma will be useful.

Lemma 5.2: *Let $\overrightarrow{S} = (S, \underset{S}{\rightarrow})$ be a poset and T be a subset of S. For any sequential poset $\overrightarrow{T} = (T, \underset{T}{\rightarrow})$ that is a topological sort of $(T, \underset{S}{\rightarrow})$, there exists a sequential poset that is a topological sort of $\overrightarrow{S}$ and that respects $\overrightarrow{T}$.*

Proof: Suppose $\overrightarrow{T}$ as required by the lemma exists. Let $\underset{ST}{\rightarrow}$ be the union of $\underset{S}{\rightarrow}$ and $\underset{T}{\rightarrow}$. Suppose there is a cyclic list of operations in S, $C = s_0, s_1, \ldots, s_k$, where $s_0 = s_k$ and for $0 \le i < k$, $s_i \underset{ST}{\rightarrow} s_{i+1}$. Without loss of generality, assume C is a shortest such list. Certainly, $\underset{ST}{\rightarrow}$ is irreflexive since both $\underset{S}{\rightarrow}$ and $\underset{T}{\rightarrow}$ are, so $k \ge 1$. All the operations in C cannot be in T or $\overrightarrow{T}$ would not be a poset since by transitivity of $\overrightarrow{T}$, we would have $s_1 \underset{T}{\rightarrow} s_0$, which would violate antisymmetry. Also, the operations of C cannot all be in $S - T$ or S would not be a poset for the same reason. Thus $k > 1$. Assume without loss of generality that $s_0 \in T$ and $s_1 \in S - T$. Then $s_0 \underset{S}{\rightarrow} s_1$ and $s_1 \underset{S}{\rightarrow} s_2$, for s_1 is not in T. But since $\underset{S}{\rightarrow}$ is transitive, $s_0 \underset{S}{\rightarrow} s_2$, and C is not the shortest list, contrary to assumption. Thus, there is no cyclic list of operations of S ordered by $\underset{ST}{\rightarrow}$.

Now let $\overrightarrow{S'} = (S, \underset{S'}{\rightarrow})$ where $\underset{S'}{\rightarrow}$ is the transitive closure of $\underset{ST}{\rightarrow}$. Suppose $\underset{S'}{\rightarrow}$ is not antisymmetric. Then there is a pair of operations $x, y \in S$ such that $x \underset{S'}{\rightarrow} y$ and $y \underset{S'}{\rightarrow} x$. If it is not the case that $x \underset{ST}{\rightarrow} y$, then there is a list of operations $x = s_0, s_1, \ldots, s_k = y$, where for $0 \le i < k$, $s_i \underset{ST}{\rightarrow} s_{i+1}$ (otherwise, (x, y) would not be added to $\underset{S'}{\rightarrow}$ by taking the transitive closure). Similarly, there is such a list connecting y to x in $\underset{ST}{\rightarrow}$. But then there is a cyclic list operations of S in $\underset{ST}{\rightarrow}$, which we have shown is impossible, so $\underset{S'}{\rightarrow}$ is antisymmetric. By a similar argument, $\underset{S'}{\rightarrow}$ is irreflexive. Now any topological sort of $\overrightarrow{S'}$ is a topological sort of $\overrightarrow{S}$ that respects $\overrightarrow{T}$, as required. □

We are now ready to present and prove the critical lemma. For convenience, for every process p, let o_0^p be a fictitious operation of p that represents the setting of its initial state. This allows us to address all reachable states of p in a consistent way. In particular, we can write $\widetilde{\overrightarrow{H}}{}_p^m(o_0^p) = (\{o_0^p\}, \emptyset)$, $\overrightarrow{S_p^m}(o_0^p) = (\{o_0^p\}, \emptyset)$, $M_p(o_0^p) = M_\perp$, and $Z_p(o_0^p) = \emptyset$. (Note that since o_0^p is fictitious, $\{o_0^p\}$ is actually the empty set.)

Lemma 5.3: *Let $\widetilde{\overrightarrow{H}}{}^m$ be a message-causal execution history of the Robust algorithm and p be a process. For every operation o of p, there is a poset $\overrightarrow{S_p^m}(o)$ that is a legal topological sort of $\widetilde{\overrightarrow{H}}{}_p^m(o)$. Furthermore, the memory projection, $M(\overrightarrow{S_p^m}(o))$ is equal to $M_p(o)$, and $Z_p(o)$ covers $S_p^m(o)$.*

Proof: We tag every operation in H with a pair for purposes of induction. For any operation o of process p, let $tag(o) = (R, C)$, where R is the number of receive operations in $H_p^m(o)$ and C is the number of operations of p since the most recent receive operation of p, or the number of operations of p since the beginning of the computation if there is no receive by p in $H_p^m(o)$. By our convention $tag(o_0^p) = (0, 0)$ for all p. Tags are ordered lexicographically: $(R_1, C_1) \leq (R_2, C_2)$ if and only if $R_1 < R_2$ or $R_1 = R_2$ and $C_1 \leq C_2$. We can now do induction over operations by tag order.

The basis holds trivially because for every process p, $\widetilde{\overrightarrow{H}}{}_p^m(o_0^p) = (\{o_0^p\}, \emptyset)$, $\overrightarrow{S_p^m}(o_0^p) = (\{o_0^p\}, \emptyset)$, $M_p(o_0^p) = M_\perp$, and $Z_p(o_0^p) = \emptyset$.

Let p be any process and o be any operation of p in H such that $(0, 0) < tag(o)$ (that is, o is any non-fictitious operation). Assume the lemma holds for all operations with tags less than o. There are two cases depending on whether o is a receive or not. Let o' be the operation of p just before o (if o is the first operation of p, then $o' = o_0^p$).

1. o is a not a receive.
   We let $\overrightarrow{S_p^m}(o) = \overrightarrow{S_p^m}(o') \cdot o$ and show that the conditions of the lemma are satisfied.

   - $[\overrightarrow{S_p^m}(o)$ is a legal topological sort of $\widetilde{\overrightarrow{H}}{}_p^m(o)]$.
     Since $o \notin S_p^m(o')$, $\overrightarrow{S_p^m}(o)$ is a sequential poset by Lemma 2.1. Since o is not a receive, o' is the only operation such that $o' \underset{H^m}{\overset{1}{\rightsquigarrow}} o$. Thus, $H_p^m(o) = \{o\} \cup H_p^m(o')$. Since, by the induction hypothesis, $S_p^m(o')$ contains exactly the operations of $H_p^m(o')$, $S_p^m(o)$ contains exactly the operations of $H_p^m(o)$. (Note that, even if o' is fictitious, this reasoning still works.) Since $\overrightarrow{S_p^m}(o')$ is legal, $\overrightarrow{S_p^m}(o)$ is trivially legal if o is a send or write operation. If o is a read of location x, it returns $M_p(o')[x] = M(\overrightarrow{S_p^m}(o'))[x] = last(\overrightarrow{S_p^m}(o'), x)$.

Since this is the value of the most recent write of x in $S_p^m(o)$, $\overrightarrow{S_p^m}(o)$ is legal. Finally, $\overrightarrow{S_p^m}(o)$ respects $\widetilde{H^m}$ because $\overrightarrow{S_p^m}(o')$ respects $\widetilde{H^m}$ and for all operations $o'' \in H_p^m(o')$, $o'' \underset{H^m}{\leadsto} o$. Thus, $\overrightarrow{S_p^m}(o)$ is a legal topological sort of $\widetilde{H_p^m}(o)$.

- $[M(\overrightarrow{S_p^m}(o)) = M_p(o)]$.
  Sends and reads do not affect memory, so in these cases $M_p(o) = M_p(o') = M(\overrightarrow{S_p^m}(o')) = M(\overrightarrow{S_p^m}(o') \cdot o) = M(\overrightarrow{S_p^m}(o))$. If o is a write of value v to location x, $M_p(o)[x] = v$ and for all $y \neq x$, $M_p(o)[y] = M_p(o')[y]$. Since the last write to x in $\overrightarrow{S_p^m}(o)$ is o, $last(\overrightarrow{S_p^m}(o), x) = v = M_p(o)[x]$. Also, for all $y \neq x$, $last(\overrightarrow{S_p^m}(o), y) = last(\overrightarrow{S_p^m}(o'), y) = M_p(o')[y] = M_p(o)[y]$. Thus, $M(\overrightarrow{S_p^m}(o)) = M_p(o)$, as required.

- $[Z_p(o)$ covers $S_p^m(o)]$.
  Sends and reads do not affect the Z set of a process, so if o is a send or a read, $Z_p(o) = Z_p(o')$. Since $Z_p(o')$ covers $S_p^m(o')$ and o is not a write, $Z_p(o)$ covers $S_p^m(o)$. If o is a write to location x, then the algorithm deletes all triples corresponding to writes to x from Z and inserts a triple corresponding to write o. By Lemma 5.1, $ts(o)$ is greater than the timestamp of every operation o'' such that $o'' \underset{H^m}{\leadsto} o$. Thus, o covers all other writes to x in $H_p^m(o)$ and hence in $S_p^m(o)$. Triples in Z for locations other than x are unaffected by operation o, so, since $Z_p(o')$ covers $S_p^m(o')$, $Z_p(o)$ covers $S_p^m(o)$.

2. o is a receive.
   Let Z_q, Z'_q and Z' be the sets assigned during o, and assume the message received, $\langle Z_q, t_q \rangle$, was sent by o_q of process q. Since $H_q^m(o_q)$ has fewer receives than $H_p^m(o)$, $tag(o_q) < tag(o)$ and the induction hypothesis applies to q as well as o'. Let Q be the set of write operations in $H_q^m(o_q)$ but not in $H_p^m(o')$.

   Let $\overrightarrow{T}$ be the topological sort of $O(Z'_q)$. Note that, since $O(Z'_q)$ consists only of write operations, $\overrightarrow{T}$ respects $\widetilde{H^m}$ by Lemma 5.1. Since $O(Z'_q) \subseteq Q$, by Lemma 5.2, $\overrightarrow{T}$ can be extended to a sequential poset over the operations of Q, say $\overrightarrow{Q}$, that also respects $\widetilde{H^m}$. Now let $\overrightarrow{S_p^m}(o) = \overrightarrow{S_p^m}(o') \cdot \overrightarrow{Q} \cdot o$, which is a sequential poset by Lemma 2.1 because $S_p^m(o') \cap Q = \emptyset$ and o is in neither $S_p^m(o')$ nor Q.

   - $[\overrightarrow{S_p^m}(o)$ is a legal topological sort of $\widetilde{H_p^m}(o)]$.
     We have just noted that $\overrightarrow{S_p^m}(o)$ is a sequential poset. By construction,

$S_p^m(o) = H_p^m(o)$. Since $\overrightarrow{S_p^m}(o')$ is legal and $\overrightarrow{Q} \cdot o$ contains no reads, $\overrightarrow{S_p^m}(o)$ is also legal. We claim that $\overrightarrow{S_p^m}(o') \cdot \overrightarrow{Q}$ respects $\widetilde{H}^m$. If not, because $\overrightarrow{S_p^m}(o')$ and $\overrightarrow{Q}$ both respect $\widetilde{H}^m$, there must be operations $o_1 \in Q$ and $o_2 \in S_p^m(o)$ such that $o_1 \underset{H^m}{\rightsquigarrow} o_2$. But by Lemma 5.1, $ts(o_1) \le ts(o_2)$, and o_1 cannot be in Q, because it is dominated by an operation (o_2) in $S_p^m(o')$. Thus, $\overrightarrow{S_p^m}(o') \cdot \overrightarrow{Q}$ respects $\widetilde{H}^m$, and so does $\overrightarrow{S_p^m}(o) = \overrightarrow{S_p^m}(o') \cdot \overrightarrow{Q} \cdot o$, because for every operation $o'' \in H_p^m(o') \cup H_q^m(o_q)$, $o'' \underset{H^m}{\rightsquigarrow} o$. Thus, $\overrightarrow{S_p^m}(o)$ is a legal topological sort of $\widetilde{H}_p^m(o)$.

- $[M(\overrightarrow{S_p^m}(o)) = M_p(o)]$.
  The only locations in M affected by a receive are those that occur in Z_q'. Let x be a location acted on by a write in Z_q'. Then, $M_p(o)[x]$ is set to the value of the last write to x in $\overrightarrow{T}$, which is exactly the same value as $last(\overrightarrow{Q}, x)$ (since $O(Z_q')$ covers Q). Since $\overrightarrow{Q}$ is concatenated to $\overrightarrow{S_p^m}(o')$ in forming $\overrightarrow{S_p^m}(o)$ and o is not a write, $last(\overrightarrow{S_p^m}(o), x) = last(\overrightarrow{Q}, x) = M_p(o)[x]$. Furthermore, any location y not occurring in Q also does not occur in Z_q', so $last(\overrightarrow{S_p^m}(o), y) = last(\overrightarrow{S_p^m}(o'), y)$. Since locations not in Z_q' are not changed, $M_p(o)[y] = M_p(o')[y] = last(\overrightarrow{S_p^m}(o'), y) = last(\overrightarrow{S_p^m}(o), y)$. Thus, $M(\overrightarrow{S_p^m}(o)) = M_p(o)$.

- $[Z_p(o)$ covers $S_p^m(o)]$.
  Let w be any write in $S_p^m(o) = H_p^m(o)$. Since o itself is not a write, w is either in $H_q^m(o_q)$ or $H_p^m(o')$, or equivalently by the definition of Q, in Q or $H_p^m(o')$. We will show there is a write in $Z_p(o) = O(Z_q') \cup O(Z')$ that covers w in either case.

  First suppose w is in Q. Since $Q \subseteq H_q^m(o_q) = S_q^m(o_q)$, $Z_q(o_q) = O(Z_q)$ covers Q, so there is some write w' (possibly w itself) in $O(Z_q)$ that covers w. If w' is in $O(Z_q')$ it is also in $Z_p(o)$, so suppose w' is not in $O(Z_q')$. Then, since w' was deleted by the second step of the receive operation, some write w'' in $Z_p(o')$ dominates w' and, by transitivity of the timestamp ordering, also dominates w. But, since both w and w'' are both writes, by Lemma 5.1 we have $w \underset{H^m}{\rightsquigarrow} w''$, implying $w \in H_p^m(o')$. But this conflicts with the choice of Q, so $w' \in O(Z_q')$. Thus, since $Z_p(o) = O(Z_q') \cup O(Z')$, there is a write ($w'$) in $Z_p(o)$ that covers w.

  Now suppose w is in $H_p^m(o')$. Then w is covered by some write, $w' \in Z_p(o')$. If w' is in $O(Z')$ it is also in $Z_p(o)$. Otherwise, if w' is not $O(Z')$, then it was deleted because some write w'' in $O(Z_q')$ covers w'. In this case, w'' is in $Z_p(o)$ and covers w. Thus in all cases, each write in $H_p^m(o) = S_p^m(o)$ is covered by a write in $Z_p(o)$, so $Z_p(o)$ covers $S_p^m(o)$.

$\square$

The theorem is now straightforward. Note that any execution of the Robust algoritm has both a message-causal and a causal execution history.

Theorem 5.4: *For every message-causal execution history $\overset{\rightsquigarrow}{H^m}$ of the Robust algorithm, the corresponding causal execution history $\overset{\rightsquigarrow}{H}$ satisfies causality.*

Proof: We will show $\overrightarrow{S_p^m} = (S_p^m, \underset{S_p^m}{\rightarrow})$ is a legal topological sort of $(H_p^m, \underset{H^m}{\rightsquigarrow})$, and that this implies $\overrightarrow{S_p^m}$ sequences $(H_p, \underset{H}{\rightsquigarrow})$. Let p be any process in $\mathcal{P}$, and let $o_1^p, o_2^p, \ldots$ be all the operations of p in H ordered by process order $(\underset{H}{\Rightarrow})$. Let $H_p^m = \bigcup_{i=1,2,\ldots} H_p^m(o_i^p)$, $S_p^m = \bigcup_{i=1,2,\ldots} S_p^m(o_i^p)$, and $\underset{S_p^m}{\rightarrow} = \bigcup_{i=1,2,\ldots} \underset{S_p^m(o_i^p)}{\longrightarrow}$.

Every operation, $o \in S_p^m$, occurs in $S_p^m(o_i^p)$ for some i. Since $S_p^m(o_i^p) = H_p^m(o_i^p)$ by Lemma 5.3, $S_p^m \subseteq H_p^m$. Similarly, every operation o' in H_p^m occurs in $H_p^m(o_j^p) = S_p^m(o_j^p)$ for some j, implying $H_p^m \subseteq S_p^m$. So $S_p^m = H_p^m$.

Now showing that $\overrightarrow{S_p^m}$ is a legal sequential poset respecting $\overset{\rightsquigarrow}{H^m}$ establishes that $\overrightarrow{S_p^m}$ is a legal topological sort of $(H_p^m, \underset{H^m}{\rightsquigarrow})$. It takes at most three operations to demonstrate a violation of one of the properties of a legal sequential poset or that $\overset{\rightsquigarrow}{H^m}$ is not respected. Suppose o, o' and o'' are three distinct elements of S_p^m that can be used to demonstrate that $\overrightarrow{S_p^m}$ is not a sequential poset or does not respect $\overset{\rightsquigarrow}{H^m}$. Since for all integers $i, j > 0$, $i \leq j$ implies $S_p^m(o_i) \subseteq S_p^m(o_j)$, there is some least i such that $o, o', o'' \in S_p^m(o_i)$. Since $\overrightarrow{S_p^m}(o_i)$ is a legal topological sort of $\overset{\rightsquigarrow}{H_p^m}(o_i)$ by Lemma 5.3, it cannot be that o, o', o'' can be used in $\overrightarrow{S_p^m}(o_i)$ to refute legality, transitivity, non-relatedness or that $\overset{\rightsquigarrow}{H^m}$ is not respected. Thus, there must be some $j > i$ such that $\overrightarrow{S_p^m}(o_j)$ adds pairs that violate one of the properties. But because $S_p^m(o_i) \subseteq S_p^m(o_j)$, $\overrightarrow{S_p^m}(o_j)$ also has this violation, contradicting Lemma 5.3.

All that remains to be shown is that $\overset{\rightsquigarrow}{H^m}$ respects $\overset{\rightsquigarrow}{H}$ and that $H_p \subseteq H_p^m$, since this implies $\overrightarrow{S_p^m}$ sequences $(H_p, \underset{H}{\rightsquigarrow})$. To see that $\overset{\rightsquigarrow}{H^m}$ respects $\overset{\rightsquigarrow}{H}$ note that for any two operations $o, o' \in H$, $o \underset{H}{\mapsto} o'$ implies that $o \underset{H^m}{\rightsquigarrow} o'$, because o is either a write by the same process execution o' or a write by another process that was received by a chain of messages.

We now show that $H_p \subseteq H_p^m$. Let o be any operation in H_p, so that o is either an operation of p or is a write known to some correct process. If o is an operation of p, it occurs in $H_p^m(o_i^p)$ for some i, and certainly occurs in H_p^m. Otherwise, o is a write known to some correct process, say q, with operations $o_1^q, o_2^q, \ldots$ in process order. Let o_i^q be the first operation at which q knows of o. After this point q knows of o, and every message sent by q contains a triple that covers o. Since the q-to-p channel eventually delivers one of these messages, o will be included in $H_p^m(o_j)$ for some j. Thus, $H_p \subseteq H_p^m$, and the theorem holds since $\overrightarrow{S_p^m}(o)$ is legal sequencing of $(H_p, \underset{H}{\leadsto})$.

□

When introducing the Robust algorithm, we mentioned that messages required $O(mn^2)$ bits where $m = |\mathcal{M}|$, $n = |\mathcal{P}|$, and where we assume memory values and timestamp components need only a constant number of bits each (thus, a vector timestamp needs $O(n)$ bits). We now establish that claim by showing that at most n triples per location are needed in Z.

Lemma 5.5: *For any execution of the Robust algorithm, and any operation o by process p acting on location x, the set $Z[x]$ of p after executing o contains at most one triple written by each process.*

Proof: Formally, an induction on operations similar to that done in Lemma 5.3 is needed, but we continue informally since this result is easy to see. Read and send operations do not affect the Z set. A write operation to location x deletes all other writes to x, obviously maintaining the required condition. The only operation that could violate the condition is a receive.

Let o be a receive operation, and let Z refers to the set of triples at the beginning of the operation, Z_{new} be the set at the end, and Z_q, Z'_q and Z' be the sets referred to during the execution of o. By the inductive assumption, any process r has at most one write triple to location x in Z and at most one in Z_q. Suppose that r has two distinct writes, w_1 and w_2, to location x corresponding to triples in Z_{new}. Since both writes are by the same process, there timestamps are related by $\underset{H^m}{\leadsto}$, and so without loss of generality, we may assume $ts(w_1) < ts(w_2)$. Since $Z'_q \subseteq Z_q$ and $Z' \subseteq Z$, neither $O(Z'_q)$ nor $O(Z')$ can contain both writes by the induction hypothesis. Thus, since $Z_{new} = Z' \cup Z'_q$, one of the writes is in $O(Z'_q)$ and the other is in $O(Z')$.

Suppose w_1 is in $O(Z'_q)$, implying w_2 is in $O(Z')$, and hence in $O(Z)$. But this is impossible because w_1 is dominated by w_2, and the triple of w_1 would be deleted when Z'_q was formed.

On the other hand, suppose w_1 is in $O(Z')$, so that w_2 is in $O(Z'_q)$. This is also impossible because w_1 is covered by w_2, and the triple of w_1 would be deleted when Z' was formed. Thus, the Z_{new} cannot contain two writes by the same process. □

6 Concluding Remarks

We have introduced a type of memory called *causal memory*, which preserves the orderings of events defined by causality. Because different applications may require that different causal relationships be preserved, we can actually define a class of causal memories, one of which is presented in detail in this paper. We can define all of these memories using a simple formal model.

We show that greater concurrency is possible with causal memory than with atomic or sequentially consistent memories. This is evident from the simple implementation given in Section 4, which allows a process to complete its read or write operations without requiring communication with other processes (newly written values are sent in messages to other processes, but the sender can continue execution while the messages are in transit). In contrast, it can be easily shown that any implementation of an atomic or sequentially consistent memory must force a process to sometimes block execution until communication completes with one or more other processes. For example, in their implementation of sequentially consistent memory, Afek et al. [2] allow asynchrony by queuing updates, but they require an expensive atomic broadcast to complete when a process reads a variable with a queued update. By demonstrating that an application in which processes interact heavily can be programmed in an identical manner on both causal and atomic memories (other applications are discussed in a related paper [3]), we believe that causal memory provides a reasonable and viable target architecture for programming distributed and parallel applications.

Our implementations of causal memory can use a large amount of storage. (Note that the simple algorithm requires unbounded space, as can be seen by considering a computation in which messages on a single channel, say pq, are delayed, while writes continue and all other channels run normally. The *pending* set of process q will grow without bound.) Perhaps a large amount of space is the price paid for wait-free causal memory based on message passing, but we would like to have clear lower bound results. Some interesting open questions are

- Can bounded timestamps [10] (that are consistent with causality) be implemented in a wait-free manner using message passing?
- Can the upper bound on space established for the Robust algorithm be improved (ignoring the problem of unbounded timestamps)? (We have developed "hybrid" algorithms that blend the ideas of the Simple and Robust algorithm, but have not improved on the space bound of the Robust algorithm.)

In the absence of concrete results, it may be interesting to explore the performance of variants of the Robust algorithm under reasonable assumptions about delays in message passing.

In this paper, we have only considered one type of causal memory. However, there exist varieties of causal memory which are both stronger and weaker than the causal memory implemented by the Simple and Robust algorithms. For example, a weaker causal memory can be defined using *weak causality*. An execution history $\overset{\rightsquigarrow}{H}$ satisfies weak causality if, for each process $p \in \mathcal{P}$ and each memory location $x \in \mathcal{M}$, there is a legal sequential poset $\overrightarrow{S_p^x}$ of the read and write operations of p to location x and the write operations of all other processes to x that respects $\overset{\rightsquigarrow}{H}$. In fact, the linear solver example presented earlier still runs correctly even when the memory correctness is defined using weak causality. Our future work will address the relationships of various types of memories and characterizations of problems that can be solved with such memories. The goal of our practical work is to demonstrate that weak memories can be feasibly programmed and implemented to achieve overall performance gains.

References

[1] Sarita V. Adve and Mark D. Hill. Weak ordering — a new definition. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 2–14, 1990.

[2] Yehuda Afek, Geoffrey Brown, and Michael Merritt. A lazy cache algorithm. In *Proceedings of the 1989 ACM Symposium on Parallel Algorithms and Architectures*, pages 209–223, June 1989.

[3] Mustaque Ahamad, Phillip W. Hutto, and Ranjit John. Implementing and programming causal distributed shared memory. In *11th International Conference on Dist. Comput.*, May 1991.

[4] Henri E. Bal and Andrew S. Tanenbaum. Distributed programming with shared data. In *Proceedings of the IEEE 1988 International Conference on Computer Languages*, pages 82–91, October 1988.

[5] J. K. Bennett, J. B. Carter, and W. Zwaenepoel. Adaptive software cache management for distributed shared memory architectures. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, May 1990.

[6] J. K. Bennett, J. B. Carter, and W. Zwaenepoel. Munin: Distributed shared memory based on type-specific memory coherence. In *Proceedings of the 2nd ACM Symposium on Principles and Practice of Parallel Programming*, pages 168–177, March 1990.

[7] Kenneth Birman, Andre Shiper, and Pat Stephenson. Lightweight causal and atomic group multicast. Technical Report 91-1192, Department of Computer Science, Cornell University, February 1991. To appear in *ACM Transactions on Computer Systems*.

[8] Roberto Bisiani, Andreas Nowatzyk, and Mosur Ravishankar. Coherent shared memory on a distributed memory machine. In *Proceedings of the 1989 International Conference on Parallel Processing*, volume I, pages 133–141, 1989.

[9] Patrick M. Clancey and Joan M. Francioni. Distribution of pages in distributed shared memory. In *Proceedings of the 1990 International Conference on Parallel Processing*, volume II, pages 258–265, August 1990.

[10] Danny Dolev and Nir Shavit. Bounded concurrent time-stamp systems are constructible. In *Proceedings of the Twenty-First ACM Symposium on Theory of Computing*, pages 454–466, May 1989.

[11] Michel Dubois, Christoph Scheurich, and Faye Briggs. Synchronization, coherence, and event ordering in multiprocessors. *IEEE Computer*, 21(2):9–22, February 1988.

[12] Colin Fidge. Timestamps in message-passing systems that preserve the partial ordering. In *Proceedings of the Eleventh Australian Computer Science Conference*, University of Queensland, Australia, 1988.

[13] Brett D. Fleisch and Gerald J. Popek. Mirage: A coherent distributed shared memory design. In *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles*, pages 211–224, December 1989.

[14] Kourosh Gharachorloo, Daniel Lenoski, James Laudon, Phillip Gibbons, Anoop Gupta, and John Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 15–26, 1990.

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

[16] Phillip W. Hutto and Mustaque Ahamad. Slow memory: Weakening consistency to enhance concurrency in distributed shared memories. In *Proceedings of the 10th International Conference on Distributed Computing Systems*, pages 302–311, 1990.

[17] R. E. Kessler and M. Livny. An analysis of distributed shared memory algorithms. In *Proceedings of the 9th International Conference on Distributed Computing*, pages 498–505, June 1989.

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

[19] Leslie Lamport. How to make a multiprocessor computer that correct executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9):690–691, September 1979.

[20] Leslie Lamport. On interprocess communication; part I: Basic formalism. *Distributed Computing*, 1(2):77–85, 1986.

[21] Leslie Lamport. On interprocess communication; part II: Algorithms. *Distributed Computing*, 1(2):86–101, 1986.

[22] Kai Li and Paul Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems*, 7(4):321–359, November 1989.

[23] Richard J. Lipton and Jonathan S. Sandberg. PRAM: A scalable shared memory. Technical Report 180-88, Princeton University, Department of Computer Science, September 1988.

[24] Richard J. Lipton and D. N. Serpanos. Uniform-cost communication in scalable multiprocessors. In *Proceedings of the 1990 International Conference on Parallel Processing*, pages I429–I432, August 1990.

[25] Friedemann Mattern. Time and global states of distributed systems. In Michel Cosnard, Patrice Quinton, Yves Robert, and Michel Raynal, editors, *Proceedings of the International Workshop on Parallel and Distributed Algorithms*. North-Holland, October 1988.

[26] J. Misra. Axioms for memory access in asynchronous hardware systems. *ACM Transactions on Programming Languages and Systems*, 8(1):142–153, January 1986.

[27] Umakishore Ramachandran, Mustaque Ahamad, and M. Yousef Khalidi. Coherence of distributed shared memory: Unifying synchronization and data transfer. In *Proceedings of the 18th International Conference on Parallel Processing*, pages 160–169, August 1989.