Efficient Bounded Timestamping from Standard Synchronization Primitives

BENYAMIN BASHARI, University of Calgary, Canada
ALI JAMADI, University of Calgary, Canada
PHILIPP WOELFEL, University of Calgary, Canada

Bounded timestamps [9, 19] allow a temporal ordering of events in executions of concurrent algorithms. They are a fundamental and well studied building block used in many shared memory algorithms. A concurrent timestamp system keeps track of m timestamps, which is usually greater or equal to the number of processes in the system, n. A process may, at any point, obtain a new timestamp, and later determine a total order of all process's most recent timestamps. Known timestamp algorithms do not scale well in the number of processes. Getting a new timestamp takes at least a linear number of steps, and a lower bound by Israeli and Li [19] implies that each timestamp needs to be represented by at least $\Omega(m)$ bits.

We introduce a slightly different semantics for timestamping, where there is no fixed timestamp value associated with an event. A process can execute operation `updateTS()` to update its latest timestamp, associating it with the (linearization) point of that operation, and `isEarlier(`p`,`q`)` to determine the temporal order of the latest `updateTS()` operations executed by processes p and q. Since no static timestamp value is returned by `updateTS()`, the lower bound of Israeli and Li does not apply.

We present efficient linearizable and wait-free implementations of these methods using a single bounded fetch-and-add object and $O(n^2)$ bounded compare-and-swap objects, which are available on standard hardware. The step complexity of each method call is constant, and base objects need only store $O(\log n)$ bits.

CCS Concepts: • **Theory of computation → Shared memory algorithms**.

Additional Key Words and Phrases: timestamping, shared memory, concurrency, distributed algorithms

1 INTRODUCTION

Timestamp algorithms are an important tool for synchronizing processes in asynchronous systems. They play a key role in solving many fundamental problems in concurrent algorithms, such as mutual- and k-exclusion [3, 14, 22, 25], consensus [1], obtaining snapshots of shared memory [16], and register constructions [17, 27]. Using a timestamp algorithm, processes in a system can determine a temporal ordering of events. A timestamp object typically provides two main functionalities: assigning a new timestamp to an event, and returning the temporal ordering of some events based on their timestamps.

Early work, starting with Lamport's logical clocks [23], assumed that timestamps can be chosen from an infinite totally ordered domain. Israeli and Li [19] considered bounded timestamp systems, where each timestamp is chosen from a finite domain, even though processes can infinitely often update their timestamps. (But there is only a finite number of "active" timestamps, typically one per process.) Since then many concurrent bounded timestamp algorithms have been devised [9–11, 15, 17, 20].

All of these algorithms have the property that the timestamps are *immutable*: Once a process assigns a timestamp to an event, the value of that timestamp remains the same until it is updated and assigned to a new event. This allows for easy comparison of active timestamps, by applying an appropriate irreflexive and antisymmetric "dominance" relation defined on all timestamps, which needs to be transitive only on the set of active timestamps.

Authors' addresses: Benyamin Bashari, University of Calgary, Calgary, Canada, benyamin.bashari@ucalgary.ca; Ali Jamadi, University of Calgary, Calgary, Canada, ali.jamadi@ucalgary.ca; Philipp Woelfel, University of Calgary, Calgary, Canada, woelfel@ucalgary.ca.

Assuming that the order of events is uniquely determined by these timestamp values, a lower bound by Israeli and Li [19] implies that at least $2^{\Omega(m)}$ timestamps are necessary, where m is the maximum number of active timestamps. Generally, m is at least as large (and often equal to) the number of processors in the system, n. Thus, employing such algorithms requires memory words of size at least n. This is generally not a realistic assumption, unless large memory words are simulated from smaller ones, which is very inefficient. (A common assumption for shared memory algorithms is that memory words can store $\Theta(\log n)$ bits.) Moreover, even if large memory words are assumed, these bounded timestamp algorithms exhibit a high step complexity of at least $\Omega(n)$ for each operation.

In this paper, we consider a slightly different semantics for timestamping, where timestamps are not immutable. Instead, the timestamps obtained by two processes, p and q, may change as a result of other processes getting new timestamps. But the *relative order* of these two timestamps does not change. Because of this, the lower bound of Israeli and Li does not apply, and we can hope for much more efficient timestamping algorithms.

To have a natural and simple interface, we consider the following sequential specification. A *mutable timestamp system* (short: MTS) maintains n timestamps, one for each process. It supports two operations, `updateTS()` and `isEarlier()`. Method `updateTS()` by process p updates p's timestamp, but does not return anything. Method `isEarlier(`p, q`)` takes as parameter two process IDs, p and q, and returns True either if p called `updateTS()` at least once and q called `updateTS()` at least once after p's latest `updateTS()` call, or if neither p nor q has called `updateTS()` and p's ID is less than q's ID. Otherwise, it returns False.

Observe that an MTS object has consensus number infinity, as processes can agree on the ID of one of the processes participating in a consensus protocol (i.e., solve name-consensus): A process calls `updateTS()`, and then determines and outputs the ID of the process that called `updateTS()` first, using $n-1$ `isEarlier()` calls. Hence, to implement a linearizable MTS, we need strong atomic operations, such as `CAS()` (compare-and-swap). Conventional immutable timestamp algorithms have weaker than linearizable semantics, and can be implemented from registers. It is easy to implement an efficient MTS on a system supporting unbounded `FAA()` (fetch-and-add) in addition to `CAS()`, if memory words have infinite size. But assuming unbounded word size defeats the purpose of a bounded timestamp system. Therefore, it is a natural question, whether there exists an efficient linearizable MTS implementation on a system with reasonably bounded word size, and using only commonly available atomic operations. We answer this question affirmatively, assuming a system with word-size $w = O(\log n)$ that supports (bounded) `FAA()` and `CAS()` operations. (Overflows caused by `FAA()` operations are allowed to cause arbitrary value changes.)

Theorem 1. *There is a linearizable implementation of an n-process MTS on a system supporting* `FAA()` *and* `CAS()`*, assuming $O(n^2)$ memory words of size $w = O(\log n)$, such that each* `updateTS()` *and* `isEarlier()` *operation terminates within a constant number of its own steps.*

Instead of `FAA()`, our actual algorithm employs a bounded object that provides an operation to fetch and increment modulo φ for some fixed integer φ. We will give a simple algorithm that implements such a mod φ-`FAI()` operation from a single bounded `FAA` object, provided that the word-size is at least $\log(n \cdot \varphi)$ bits. This algorithm may be of independent interest.

Contrary to many conventional timestamp systems, the MTS does not directly support a `scan()` operation that returns a total order of all timestamps. While adding such a `scan()` may be possible, this is beyond the scope of this work. However, for many applications such a `scan()` is not necessary. Often pairwise comparisons suffice, or the algorithm can be modified to use pairwise comparisons [3, 7, 16, 22].

It is also easy to extend the semantics of our MTS to allow $k > 1$ active timestamps. To that end, we allow any process p to call `updateTS(`i`)` with a parameter $i \in \{0, \ldots, k-1\}$. An `isEarlier(`$(p,i),(q,j)$`)` call then determines the temporal order of p's i'th timestamp and q's j'th timestamp. Such an extended semantics can easily be obtained using a regular kn process MTS, by allowing each of the n processes to assume k different IDs. This way we obtain an algorithm with constant step complexity, but using $O((kn)^2)$ memory words of size $O(\log(kn))$. One application of this extended specification is a recent efficient single-writer snapshot algorithm by Bashari and Woelfel [7]. That algorithm uses a single unbounded FAI object in addition to CAS and registers. Applying some minor modification to the algorithm, that FAI object can be replaced with an MTS with $k = O(n)$ active timestamps per process (see Section 7 for more details).

2 RELATED WORK

Most work on timestamping considers systems that support only atomic read and write operations. Unless otherwise mentioned, this is what we assume in this section.

Lamport defined the "happens-before" order, where event A by process p happens-before event B by process q if A can cause or influence B [23]. He also defined logical clocks that assign integer values, called timestamps or labels, to events, such that the order of timestamps is consistent with the happens-before order of the associated events. Fidge and Mattern introduced the notion of vector clocks [13, 24]. Here, every timestamp is a vector of n integers. A timestamp T is considered smaller than a timestamp T', if every component of T is smaller or equal to the corresponding component of T'. The authors present algorithms that guarantee that the order of timestamps is consistent with the happens-before order of events.

In the above algorithms the size of timestamps may grow without a bound. Israeli and Li [19] considered the *bounded timestamping problem*, where timestamp sizes are bounded, provided that at any point only a bounded number of timestamps, m, are active (i.e., may still be compared or accessed at a later point). They also presented a sequential algorithm for this problem, and proved that timestamps of size $\Omega(m)$ bits are sufficient and necessary (even without concurrency). The lower bound applies to sequential systems supporting two operations, `getTimestamp()` and `scan()`. A `getTimestamp(i)` call, for $i \in \{0, \ldots, m-1\}$, returns a value, called timestamp, and a `scan()` call returns the temporal order of the latest `getTimestamp(j)` calls for all $j \in \{0, \ldots, m-1\}$. The lower bound assumes, that this order is uniquely determined by the return values of the corresponding `getTimestamp()` operations (and thus, timestamps are immutable).

Dolev and Shavit [9] presented the first bounded concurrent timestamp algorithm. Assuming that n is the number of processes in the system, their implementation requires $O(n^2 \log n)$ shared memory steps for a `scan()` operation, and $O(n)$ shared memory steps for assigning a timestamp to an event. They assume registers of size $\Theta(n)$ bits, which matches Israeli and Li's lower bound for timestamps. Gawlick, Lynch, and Shavit [15] presented a bounded timestamping system that requires $O(n \log n)$ steps for scanning and assigning timestamps, but uses registers of size $\Theta(n^2)$. Israeli and Li [20] improved the step complexity to $O(n)$. Dwork and Waarts [11] presented a new primitive, called *traceable use abstraction*, which allows each process to determine which variables from a private pool are in use by other processes. They demonstrated how such a primitive can be used to solve the bounded timestamp problem with linear step complexity, by allowing processes to safely recycle variables from their own pool. Their system has a specification that differs slightly from the one presented by Dolev and Shavit. It allows the order of timestamps to be determined by the system state (e.g., by information stored in shared variables), and not only by timestamp values. Even though the lower bound of Israeli and Li does not apply in this case, the algorithm uses registers of size $\Theta(n \log n)$ bits. Other algorithms implementing this specification improved the number of shared memory registers [17] and process-local space [26].

Dwork, Herlihy, Plotkin, and Waarts [10] presented a bounded timestamp system with the same sequential specification as the one by Dolev and Shavit. It maintains linear step complexity, and reduce register size to $O(n)$.

Ellen, Fatourou, and Ruppert [12] proved that $\Omega(\sqrt{n})$ registers are required for obstruction-free unbounded timestamping, assuming only atomic reads and writes. This was later improved to $\Omega(n)$ [18].

3 MODEL AND PRELIMINARIES

We consider the standard asynchronous shared memory model, where n processes communicate by performing atomic operations on shared objects, each of which can store $w = O(\log n)$ bits. For this paper, we assume the system provides registers, compare-and-swap (CAS), and fetch-and-add (FAA) objects. All these objects support `read()` operations, and a register supports also `write()` operations. A CAS objects supports the operation `CAS`(old, new), which changes the value of the object to new, if it is old, and otherwise does not change the object's value. In either case it returns the previous value of the object (or sometimes a Boolean value reporting whether it did change the object's value). An FAA object stores an integer in $\{0, \ldots, 2^w - 1\}$, and supports the operation `FAA`(d), which increases the value of the object by the given integer d, provided that the sum is in $\{0, \ldots, 2^w - 1\}$, and then returns the previous value of the register. We assume that in case of an overflow, i.e., if the sum is outside this range, the behavior of the `FAA()` operation can be arbitrary. (Our algorithm ensures that this never happens.)

In order to avoid the ABA problem, our algorithm uses load-linked/store-conditional (LL/SC) instead of CAS objects. Such an object supports the operations `LL()`, which returns the value of the object, and `SC`(x), which succeeds and stores x in the object, provided that the calling process previously called `LL()`, and no other successful `SC()` operation has been performed (by any process) since then. Otherwise, the operation fails. The operation returns True if successful and False, otherwise. Moreover, the LL/SC object also supports an operation `VL()`, which returns True if an `SC()` operation executed in its place by the same process at the same time would succeed, and False otherwise. Our algorithm uses $O(n)$ LL/SC objects, each storing $O(\log n)$ bits. LL/SC objects are generally not available on hardware. Therefore, we employ a linearizable implementation of Jayanti and Petrovic [21], which uses $O(n)$ registers and CAS objects of size $O(\log n)$ bits to simulate one of our LL/SC objects with constant step complexity. This leads to the overall space complexity claimed in Theorem 1.

We will also need a mod φ-fetch-and-increment (mod φ-FAI) object. Such an object stores an integer in $\{0, \ldots, \varphi - 1\}$, and supports the operations `read()` and `FAI()`. The latter operation increments the value of the object by 1 modulo φ, and returns the previous value. Even though it is possible that `FAA()` operations on modern hardware "wrap around" on overflow, i.e., perform arithmetic modulo 2^w, this may not always be so, and the exact overflow behavior is often not well documented. Moreover, we need to increment modulo φ for a value that does not divide 2^m. Therefore, in Section 6 we will present an implementation of a linearizable mod φ-FAI object, for any $\varphi \in \{0, \ldots, \lfloor 2^w/n \rfloor\}$, from a single bounded FAA object. The algorithm is simple and efficient: each `FAI()` operation requires only two operations on the bounded FAA object.

4 ALGORITHM

4.1 High Level Description

In our MTS implementation, each process maintains an active timestamp, which is a value from a bounded domain D. We use an array TS of n LL/SC objects, where TS$[p]$ stores process p's active timestamp, in addition to some auxiliary information. An irreflexive and antisymmetric *dominance*

order $\prec$ on D is used to decide the temporal order of the corresponding `updateTS()` operations. In order to keep the size of D polynomial in n, the active timestamp of a process p may need to be reassigned a different value, if other processes repeatedly get new timestamps.

We will partition D into three clusters, with numbers 0, 1, and 2. For a positive integer $\delta = O(n^3)$, which is specified precisely in section 4.2, each cluster $i \in \{0, 1, 2\}$ comprises $n + \delta$ timestamps (i, j), where $-n \leq j < \delta$. Timestamp (i', j') *dominates* timestamp (i, j), in short $(i, j) \prec (i', j')$, if either $i = i'$ and $j < j'$, or $i = (i' - 1) \bmod 3$.

4.1.1 *Updating Timestamps.*

There is always one *active cluster* from which new timestamps are chosen. We use a global mod (3δ)-`FAI` object, Counter, whose value determines the active cluster, and also the next timestamp a process is trying to choose from that cluster. Specifically, the active cluster is $\lfloor \text{Counter}/\delta \rfloor$. Once a cluster i is active, new timestamps from that cluster are chosen in increasing (but not necessarily consecutive) order starting with $(i, 0)$. The "negative" side of the cluster containing timestamps (i, j) with $j < 0$ is reserved for later use.

Reassigning Timestamps. Ideally, a process can obtain a new timestamp by performing Counter.`FAI()`, and using the result of that increment operation. The main problem occurs, when the active cluster changes. E.g., suppose a process p increments Counter from 2δ to $2\delta + 1$, and thus obtains the first timestamp from cluster 2, i.e., $(2, 0)$, while the active timestamp of some process $q \neq p$ is in cluster 0. Then the dominance order between p's and q's timestamps does not match the temporal order of their `updateTS()` operations. To deal with this, we make sure that all active timestamps from cluster 0 are moved to cluster 1, before any process can get a new timestamp from cluster 2. (The idea of using values modulo 3 and assigning timestamps in such a way that one "cluster" is always empty originates from Israeli and Li's timestamping system [19].)

Whenever we move a timestamp from cluster 0 to cluster 1, we have to make sure that the dominance order of active timestamps is not affected. For that we use the reserved negative side of cluster 1: While cluster 1 is still active, all active timestamps from cluster 0 are reassigned to the negative side of cluster 1. Moreover, two active timestamps $(0, i)$ and $(0, j)$, where $i < j$, are moved to $(1, i')$ and $(1, j')$ such that $i' < j' < 0$. We also reassign timestamp $(0, j)$ to cluster 1 before we reassign timestamp $(0, i)$ to ensure that their dominance order remains preserved during this reassignment process. Timestamps that were obtained from cluster 1 before this reassignment, and timestamps that processes obtain during this reassignment, are located on the non-negative side of cluster 1. Therefore, the dominance order of those timestamps and the ones moved from cluster 0 to cluster 1, matches the temporal order of the corresponding `updateTS()` operations.

Distributing Work. Reassigning a single timestamp from cluster 0 to cluster 1 (or, more generally from cluster i to cluster $(i + 1) \bmod 3$) involves, roughly, taking a partial snapshot of the TS array, determining the index q, such that TS$[q]$ stores the dominant timestamp in cluster 0, and replacing the value of that array entry with a new value that corresponds to the appropriate timestamp in cluster 1. (Using LL/SC operations, we can avoid that outdated values get written to the TS array.) In order to keep individual step complexity constant, the work needed for reassigning all timestamps from one cluster to the next needs to be shared by the processes performing `updateTS()` and `isEarlier()` operations. I.e., during each `updateTS()` and `isEarlier()` operation a process performs a constant number of steps contributing to this reassignment process.

Helping. Recall that in its `updateTS()` operation, a process p performs `FAI()` on Counter in order to determine a new timestamp. After that, it has to update TS$[p]$ accordingly, using an `SC()` operation. Because its timestamp may in the meantime have been reassigned to a different cluster, this `SC()` operation may fail. (In fact, it is important that p's `SC()` does not succeed, as

otherwise the dominance order of p's active timestamp and other active timestamps might not be consistent with the order of the updateTS() operations.) Process p can repeatedly obtain a new timestamp from Counter and try to store it in TS[p], but obviously it may fail every time. However, if p fails three times, then over that interval TS[p] must be changed by other processes at least three times. This can only happen if all other processes complete at least $\Omega(\delta) = \Omega(n^3)$ updateTS() and isEarlier() calls. The idea is now that during each updateTS() method call and on average during one out of n isEarlier() calls, some process q helps some other process r that has a pending updateTS() call, by repeatedly determining a new timestamp (using FAI() on Counter) and trying to update TS[r] accordingly, until some process has updated TS[r] with a new timestamp. (This will happen if at least one process q helps r a constant number of times.) Thus, our algorithm guarantees that in an interval in which $\Omega(n^3)$ updateTS() and isEarlier() calls occur, process p's timestamp will be successfully updated, either by p itself, or by some other process.

To indicate that it needs help, process p changes an *announce bit* in a separate single-writer register A[p]. Similarly, TS[p] stores an auxiliary bit (in addition to p's active timestamp), called p's *flag*. The idea is that if p's flag does not match the value of A[p], then p wants help, i.e., it is trying to update its timestamp. This allows a process q to determine if p wants help, and also to indicate that p's timestamp has been updated to a new one (i.e., p has received help) with the same successful SC() operation that stores that new timestamp in TS[p].

The Invalidation Phase. It is possible that, when trying to reassign timestamps to a new cluster, a process p loads an array entry TS[i], then falls asleep, and later uses the information it obtained earlier from TS[i] to determine the dominant timestamp in a cluster to reassign to the next cluster. Reassigning that dominant timestamp is not affected by any possible changes to TS[i] that may have happened in the meantime, and so p may base its decision on outdated information. To avoid that, we add an *invalidation phase* that invalidates all links processes may have with the TS array. This phase starts when the active cluster number changes from c to $(c+1) \bmod 3$ (and thus Counter mod δ becomes 0), and ends after $\zeta \cdot n$ increments of Counter, for some value $\zeta = O(n)$ that will be defined later. At the beginning of the invalidation phase, all active timestamps are in cluster c, and the phase ends before any of these timestamps is moved to cluster $(c+1) \bmod 3$. If, during the invalidation phase, a process reads a value i from Counter, it invalidates all other process's links to TS[i], where $i = \left\lfloor \frac{\text{Counter mod } \delta}{\zeta} \right\rfloor$, provided that no process has already done so during the ongoing invalidation phase. As a result, if some process performs an LL() and SC() pair of operations on an entry of the TS array, and both are separated by a complete invalidation phase, then the SC() fails.

Wrap-Around-Detection. It is necessary for processes to be able to detect if the cluster number has "wrapped around", which can happen once Counter has been incremented at least $2\delta + 1$ times. To avoid employing additional shared memory variables, we will sacrifice one of the processes, and repurpose its TS array entry. Specifically, we will not allow process $n-1$ to perform any updateTS() operation (clearly, this does not affect the asymptotic complexity analysis). Thus, if a process performs TS[$n-1$].LL(), and later TS[$n-1$].VL(), then the latter operation will fail once Counter has been incremented at least $2\delta + 1$ times, because at least one invalidation phase must have been completed since the LL() operation.

4.1.2 *Comparing Timestamps.*

To compare two timestamps in an isEarlier(*i,j*) call, a process p first checks if one of processes i or j is currently trying to update its timestamp, and if yes, helps it do so (by getting new timestamps and trying to store them in TS[i] and TS[j], respectively). Process p then announces the process

IDs it wants to compare. To do this, p performs an LL() and an SC() operation on a separate LL/SC object, LookupTable[p]. (The algorithm guarantees that the SC() operation succeeds.) Then p tries to obtain a consistent view (i.e., a snapshot) of TS[i] and TS[j], and determine the return value of the isEarlier(i,j) call according to the dominance order of the timestamps stored in those registers. The fact that p helped any pending updateTS(i) and updateTS(j) calls then ensures that the dominance order of TS[i] and TS[j] matches the relative order of the latest linearized updateTS(i) and updateTS(j) calls preceding the linearization point of p's isEarlier(i,j) operation.

We will now show how p can obtain a consistent view of TS[i] and TS[j]: It first sandwiches a TS[j].LL() operation in-between a TS[i].LL() and a TS[i].VL() operation. If the VL() operation succeeds, p has obtained (with its LL() operations) a consistent view of TS[i] and TS[j]. Otherwise, p repeats the above, just with i and j swapped. If this still does not yield a consistent view of TS[i] and TS[j], then both of TS[i] and TS[j] must have changed at least once.

This is a good scenario, as we will now explain. Clearly, at the point when the updateTS() call of some process z linearizes, its timestamp should be the latest one among all linearized updateTS() calls. So only if some process $m \neq z$ with a higher process ID has never called updateTS(), then m's timestamp must be ordered after z's. We now observe the following: If each of those two array entries changed at least once as a result of updateTS(i) and updateTS(j) calls, respectively, then both i and j have at least one linearized updateTS() call. Hence, if TS[i] and TS[j] change one more time because of updateTS(i) and updateTS(j) calls, respectively, then p's isEarlier(i,j) can return either Boolean value, because it overlaps with the linearization point of the second updateTS() calls. But is possible that one (or both) of these array entries change because they are being invalidated (see the description of the invalidation phase, above), or that i's and j's timestamps are reassigned to a new cluster. If such events happen a constant number of times (for a large enough constant), then many updateTS() operations must be executed during that time.

The idea is now the following: With each updateTS() operation, a process q chooses a different index $\ell \in \{0, \ldots, n-1\}$ and performs an LL() operation on LookupTable[ℓ], which returns a triple (i', j', res). If $res = \perp$, then process ℓ wants help completing an isEarlier(i', j') call. In that case, q calls isEarlier(i', j') itself, and attempts to store the result in LookupTable[ℓ].res using an SC() operation.

As a result, once process p has written $(i, j, \perp)$ to LookupTable[p], indicating that it wants help to complete an isEarlier(i, j) call, it takes at most $\Omega(n^2)$ updateTS() calls of other processes, until the result of some isEarlier(i, j) call gets written to LookupTable[p].res. Hence, if during its isEarlier(i,j) operation, process p does not itself manage to obtain a consistent view of TS[i] and TS[j] despite trying a (large enough) constant number of times, then such consistent view will be provided by some other process.

4.2 Low Level Description

We will now describe the algorithm in detail. See Figures 1 and 2 for the pseudocode.

Shared Variables and Initial Configuration. We use an array of single bit (single-writer) registers, A[0..$n-1$]. Process i uses A[i] as its *announce bit*, to let other processes know that it wants to update its timestamp. Initially, all n bits are 0. We also employ an array TS[0..$n-1$] of LL/SC objects, where TS[i] stores a quadruple with the following components in order: cluster, index, flag, and inv. Components cluster and index store the cluster and index number, respectively, of process i's active timestamp. Component flag stores a bit that can be compared to $A[i]$, to determine if process i needs help completing an updateTS() operation. Lastly, inv stores an element in $\{0, 1, 2\}$. Every time a process performs a successful SC() operation on TS[i], it sets TS[i].inv to the last active cluster number it observed. All entries of the TS array are initially $(0, \perp, 0, 2)$. We use an

array LookupTable[0..$n-1$] of LL/SC objects, where each entry stores a triple with the following components in order: x, y, and res. Components x and y store process IDs, and res stores either $\perp$ or a Boolean value. All entries of the LookupTable array are initially $(0, 0, \text{True})$. Processes use this array to announce the process IDs they want to compare during an `isEarlier()` call. Also, each process has a variable helpID with global scope, which is initially 0. During each `updateTS()` call, a process attempts to assist the process with ID helpID, and updates helpID to a different process ID. The shared variable Counter is a mod 3δ-FAI, where the value of δ is given below.

Constants. Throughout the paper we assume that the number of processes, n, is fixed. We define the following:

$$\zeta = 6n + 2 \qquad \eta = \zeta \cdot n = 6n^2 + 2n$$
$$\mu = 6n^3 + 6n^2 + 2 \qquad \gamma = 3n^3 + 4n$$
$$\delta = \eta + \mu + \gamma = 9n^3 + 12n^2 + 6n + 2$$

During an execution, the algorithm cycles through three clusters, each of which comprises three phases : 1. invalidation phase, 2. move phase, and 3. update-only phase. The constants η, μ, and γ are the number of Counter increments that occur during an invalidation phase, a move phase, and an update-only phase, respectively. Additionally, $\delta = \eta + \mu + \gamma$ is the total number of Counter increments in each active cluster.

Finally, the algorithm uses an operator "$\ll$" (see lines 49 and 54). $\mathsf{TS}[i] \ll \mathsf{TS}[j]$ returns True if one the following conditions is true, and otherwise returns False.

- $\mathsf{TS}[i].\mathsf{index} = \mathsf{TS}[j].\mathsf{index} = \perp \wedge i < j$ (none of processes i and j have performed an `updateTS()` call),
- $\mathsf{TS}[i].\mathsf{index} \neq \perp \wedge \mathsf{TS}[j].\mathsf{index} = \perp$ (process i has performed at least one `updateTS()` call, and j has not performed any)
- $(\mathsf{TS}[i].\mathsf{cluster}, \mathsf{TS}[i].\mathsf{index}) \prec (\mathsf{TS}[j].\mathsf{cluster}, \mathsf{TS}[j].\mathsf{index})$ and $\mathsf{TS}[i].\mathsf{index} \neq \perp$ and $\mathsf{TS}[j].\mathsf{index} \neq \perp$ (both, i and j, have performed at least one `updateTS()` call and j's active timestamp dominates i's).

Function updateTS(). When process i executes this function, it first announces that it wants to update its timestamp in lines 2-3. It does so by reading $\mathsf{TS}[i]$.flag (i.e., the flag component of its own timestamp) and setting its announce bit to the complement of the value read. As a consequence, $\mathsf{TS}[i].\mathtt{flag} \neq \mathsf{A}[i]$. Then, in line 4, process i calls `helpUpdateTS(`i`)`. Before this call completes, either i or some other process assisting i must have obtained and written a new timestamp to $\mathsf{TS}[i]$.

Subsequently, in line 5, process i performs a `helpSystem()` call, in which it attempts to help another process complete its pending `updateTS()` or `isEarlier()` call. The details of this method are explained later. Performing one `helpSystem()` call during each `updateTS()` call guarantees that, by performing sufficiently many `updateTS()` calls, process i helps every process at least once. Finally, in line 6, process i calls `helpActiveCluster()`. This function handles invalidating and reassigning active timestamps, and is explained later.

Function helpUpdateTS(). Consider a `helpUpdateTS(`p`)` call by process q. If p has announced that it wants to update its timestamp, then q must first try to help p. To that end, it repeats the following up to three times: First, in line 9 it performs an `LL()` operation on $\mathsf{TS}[p]$. Then it reads p's announce bit in line 10 and compares that with $\mathsf{TS}[p]$.flag. If the two are the same, then p does not need help, so q returns in line 11. Note that q obtains $\mathsf{TS}[p]$.flag and $A[p]$ in two separate shared memory steps, but since both store bits, the result of the comparison is correct at some point between these two steps.

```
1  Function updateTS():
2      ts ← TS[myID].LL()
3      A[myID].write(¬ts.flag)
4      helpUpdateTS(myID)
5      helpSystem()
6      helpActiveCluster()
7  Function helpUpdateTS(id):
8      repeat 3 times
9          ts ← TS[id].LL()
10         a ← A[id].read()
11         if a = ts.flag then return
12         c ← Counter.FAI()
13         if TS[id].SC(⌊c/δ⌋, c mod δ, a, ⌊c/δ⌋) then
               return
14 Function helpSystem():
15     helpUpdateTS(helpID)
16     (x, y, res) ← LookupTable[helpID].LL()
17     if res = ⊥ then
18         c ← isEarlier(x, y)
19         LookupTable[helpID].SC(x, y, c)
20     helpID ← (helpID + 1) mod n
21 Function helpActiveCluster():
22     conditionalReset()
23     perform κ steps from helpMoveTS()
24 Function conditionalReset():
25     repeat 2 times
26         c ← Counter.read()
27         if c mod δ ≥ η then return
28         i ← ⌊(c mod δ)/ζ⌋
29         ts ← TS[i].LL()
30         c ← Counter.read()
31         if c mod δ ≥ η then return
32         if ts.inv = ⌊c/δ⌋ then return
33         TS[i].SC(ts.cluster, ts.index, ts.flag, ⌊c/δ⌋)
34 Function isEarlier(x, y):
35     flag ← TS[x].LL().flag
36     if A[x].read() ≠ flag then helpUpdateTS(x)
37     flag ← TS[y].LL().flag
38     if A[y].read() ≠ flag then helpUpdateTS(y)
39     helpActiveCluster()
40     LookupTable[myID].LL()
41     LookupTable[myID].SC(x, y, ⊥)
42     LookupTable[myID].LL()
43     repeat 6 times
44         xTS ← TS[x].LL()
45         yTS ← TS[y].LL()
46         if TS[x].VL() then
47             res ← xTS ≪ yTS
48             LookupTable[myID].SC(x, y, res)
49             return res
50         xTS ← TS[x].LL()
51         if TS[y].VL() then
52             res ← xTS ≪ yTS
53             LookupTable[myID].SC(x, y, res)
54             return res
55     LookupTable[myID].SC(x, y, True)
56     return LookupTable[myID].LL().res
```

Fig. 1. MTS algorithm part 1

Now suppose that q reads different values from TS$[p]$.flag and $A[p]$. In that case it continues to execute lines 12-13. Here, q performs an FAI() operation on Counter and uses the return value to compute a new timestamp for p. Process q then attempts to store the newly created timestamp into TS$[p]$ by performing an SC() operation. In the same SC() operation, q also updates TS$[p]$.flag to the value it read from $A[p]$. If the operation succeeds, then q returns in line 13, and otherwise it repeats the whole procedure (up to three times).

57 **Function** helpMoveTS():
58 localTS[$n-1$] ← TS[$n-1$].LL()
59 $c \leftarrow$ Counter.read()
60 **if** $c \bmod \delta \notin [\eta, \eta+\mu-1]$ **then return**
61 **for** $i \in \{0,\ldots,n-2\}$ **do** localTS[i] ← TS[i].LL()
62 $c \leftarrow$ Counter.read()
63 **if** $c \bmod \delta \notin [\eta, \eta+\mu-1]$ **then return**
64 **for** $i \in \{0,\ldots,n-1\}$ **do**
65 **if** $\left(\text{localTS}[i].\text{cluster} = (\lfloor \frac{c}{\delta} \rfloor - 1) \bmod 3\right) \wedge \neg(\text{TS}[i].\text{VL}())$ **then return**
66 **if** $\neg$TS[$n-1$].VL() **then return**
67 activeCluster ← $\{x \in \{0,\ldots,n-1\} \mid \text{localTS}[x].\text{cluster} = \lfloor \frac{c}{\delta} \rfloor\}$
68 oldCluster ← $\{x \in \{0,\ldots,n-1\} \mid \text{localTS}[x].\text{cluster} = (\lfloor \frac{c}{\delta} \rfloor - 1) \bmod 3\}$
69 **if** oldCluster = $\emptyset$ **then return**
70 **Let** j ∈ oldCluster **such that** localTS[j].index is *maximal*
71 **Let** i ∈ activeCluster **such that** localTS[i].index is *minimal*
72 **if** localTS[j].index ≠ $\perp$ **then** newIndex ← min(localTS[i].index,0) − 1 **else** newIndex ← $\perp$
73 TS[j].SC($\lfloor \frac{c}{\delta} \rfloor$, newIndex, localTS[j].flag, $\lfloor \frac{c}{\delta} \rfloor$)

Fig. 2. MTS algorithm part 2

If q's SC() in line 13 is successful, then as a result TS[p].flag = A[p]. Therefore, all processes (including p) can observe that p does not need help anymore updating its timestamp.

We prove that if q's SC() operation in line 13 fails three times, then some other process has helped p receive a new timestamp. Therefore, once p has announced that it wants to update its active timestamp, it takes at most one complete helpUpdateTS(p) call by any process until p is assigned a new timestamp.

Function helpSystem(). The purpose of this function is to help other processes in the system. Every time a process q executes this function, it calls helpUpdateTS(helpID) (line 15). This way, q helps the process with ID helpID if needed. Additionally, in lines 16-17, process q checks if the value of LookupTable[helpID].res is equal to $\perp$. If yes, then the process with ID helpID, has a pending isEarlier() call. Thus, q will perform an isEarlier(LookupTable[helpID].x, LookupTable[helpID].y) call and attempt to store the result back into LookupTable[helpID] (lines 18-19). Process q's attempt to write to LookupTable[helpID] fails only if that entry has already been updated by some other process. Lastly, q changes helpID to (helpID + 1) mod n (line 20). This ensures that q attempts to help every process over a period in which it completes n helpSystem() calls.

Function helpActiveCluster(). This function helps maintain the following two invariants.

Invariant 1: After a cluster-k invalidation phase, and as long as the active cluster remains the same, any successful TS[i].SC() operation updates TS[i].cluster to k. The conditionalReset() call in line 22 helps to maintain this invariant.

Invariant 2: Throughout the cluster-k update-only phase, no active timestamp is in cluster $(k-1)$ mod 3. Performing κ steps of a helpMoveTS() call in line 23 during a cluster-k move phase, for a large enough constant κ (such that a complete helpMoveTS() call requires at most $n \times \kappa$ steps) contributes to maintaining this invariant. Observe that using a large enough constant κ is sufficient

for the algorithm to function properly, because the `helpMoveTS()` function requires $\Theta(n)$ steps to complete. Thus, the step complexity of `helpActiveCluster()` is constant.

Function isEarlier(). During an `isEarlier(`i, j`)` call process p first checks if process i or j is trying to update its active timestamp. To do this, p compares i's announce bit to i's flag, and similarly j's announce bit to j's flag (lines 35-38). If either i or j is updating its active timestamp, then p ensures that the new timestamp is stored in the TS array by calling `helpUpdateTS(`i`)` or `helpUpdateTS(`j`)`, respectively. This is necessary to ensure that p's `isEarlier()` call can linearize.

Recall that during `helpUpdateTS()` a process may increment Counter, and the value of Counter determines the currently active phase of the algorithm. Hence, p must contribute to the progress of the currently active phase by calling `helpActiveCluster()` (line 39).

In lines 40-41, process p sets LookupTable[p] to $(i, j, \perp)$ in order to announce that it wants to compare i and j. Then, by executing the loop in line 43, p attempts up to six times to obtain a snapshot (xTS, yTS) of (TS[i], TS[j]), as described in Section 4.1.2. If p succeeds with that, then in line 48, or 53 p resets its announcement by performing a LookupTable[p].`SC()` operation that sets LookupTable[p].res to the result of $xTS \ll yTS$. That same result p then returns in line 49, or line 54.

If after six iterations of the loop in line 43, process p has not succeeded in capturing a snapshot, then it attempts to reset its announcement by setting LookupTable[p].res to True with an `SC()` operation (line 55). Then process p calls *Lookupsp*.`LL()` to determine the value of *Lookupsp*.res, which it returns (line 56). If, while p is executing the loop, some process q sees p's announcement, performs an `isEarlier(`i,j`)` call, and updates LookupTable[p].res, then p's attempt to reset its announcement will fail. In this case, p's `isEarlier()` call returns the same result as q's `isEarlier()` call. If, on the other hand, p's attempt to reset its announcement succeeds, then the algorithm guarantees that both i and j have performed sufficiently many `updateTS()` calls, such that p's response can be arbitrary.

In any case, before returning a response, p executes a LookupTable[p].`SC()` operation that sets LookupTable[p].res to a non-$\perp$ value, effectively resetting p's announcement. Since this `SC()` fails only if some other process has already updated LookupTable[p].res with the result of an `isEarlier()` call, LookupTable[p].res $\neq \perp$ immediately after this operations. This ensures that no outdated values get stored in LookupTable[p], and that p's `SC()` operation in line 41 always succeeds.

Function conditionalReset(). In a `conditionalReset()` call, process p repeats the following up to two times: in lines 26-27, p reads Counter to determine the currently active phase. If this is not the invalidation phase, then p returns in line 27. Otherwise, for $i = \left\lfloor \frac{\text{Counter mod } \delta}{\zeta} \right\rfloor$, p performs a TS[i].`LL()` operation in line 29. Since during the invalidation phase the value of Counter mod δ is between 0 and $\eta - 1 = n \cdot \zeta - 1$, we have $0 \leq i \leq n - 1$. Then in lines 30-31, process p reads Counter, again, determines the currently active cluster, k, and checks the algorithm's current phase. If the algorithm is not in an invalidation phase, p returns in line 31. Otherwise, in line 32, p compares TS[i].inv to k, and returns if TS[i].inv is equal to k. This helps ensure that during an invalidation phase every active timestamp is invalidated at most once, so that a process that attempts to update an active timestamp, is guaranteed to succeed after a constant number of tries. Lastly, in line 33, p performs a TS[i].`SC()` operation that sets TS[i].inv to the active cluster number. If this operation is not successful, p repeats the whole procedure.

The algorithm ensures that if a process performs an `LL()` and subsequent successful `SC()` operation on TS[i'] while cluster k is continously active, then the `SC()` operation sets TS[i'].inv to k. Therefore, even if p's TS[i].`SC()` operation in the second iteration of the loop fails, TS[i].inv = k

immediately after that SC(), because some other process must have performed an LL() and subsequent successful SC() operation, while cluster k was active.

Function helpMoveTS(). This function reassigns active timestamps from a previously active cluster to the currently active cluster. It only has an effect when executed during a move phase. In fact, we must ensure that if a helpMoveTS() call is invoked in a cluster-k interval, and before TS$[n-1]$ gets invalidated, or if the active cluster changes, during its execution, it does not change any process's active timestamp.

During its helpMoveTS() call, process p first loads TS$[n-1]$ in line 58. Then, in lines 59-60, it reads Counter in order to determine the currently active phase. If this is not the move phase, then p returns in line 60. Otherwise, p continues, and in line 61, loads all indices $0, \ldots, n-2$ of the TS array. Then in lines 62-63, process p reads Counter again, to determine the current phase. If the algorithm is not in a move phase, p returns in line 63.

Otherwise, in line 62 process p reads Counter into a local variable c. Thus, $k = \lfloor c/\delta \rfloor$ is the active cluster number at that point. Then, in lines 64-65, p executes a VL() operation on every entry of the TS array that was in cluster $(k-1) \bmod 3$ at the point when p first loaded it (in lines 58 and 61). If any of p's VL() operations returns False, then p returns in line 65. Otherwise, p continues to execute a TS$[n-1]$.VL() operation in line 66 and returns in the same line, if this VL() returns False.

In lines 67-70, process p uses its local view of the TS array (obtained in lines 58 and 61) to find the process, j, such that TS$[j]$ stores the dominant active timestamp of cluster $(k-1) \bmod 3$. Hence, process j has the next active timestamp that needs to be reassigned during the ongoing cluster-k move phase. Similarly, in line 71, process p finds the process, i, such that TS$[i]$ stores the dominated active timestamp of cluster k. In line 72, process p, uses its local view of the TS array to calculate a new index newIndex for j's active timestamp. If the index of TS$[j]$ is $\perp$, then newIndex is also $\perp$. Otherwise, newIndex is the largest negative integer less than the index of TS$[i]$. Lastly, in line 73, process p executes a TS$[j]$.SC() operation to update j's active timestamp to $(k,$ newIndex$)$. Process p's SC() fails only if some other process has reassigned j's active timestamp to cluster k.

5 CORRECTNESS OVERVIEW

In this section we give an overview for a correctness proof (see Appendix A for the full proofs). We first explain the connection between the number of Counter increments and helpActiveCluster() calls. We then discuss the purpose of the invalidation and move phases, and analyze the interplay between a process's flag and announce bit. Lastly, we will give an overview of our linearizability proof.

The following relates the number of helpActiveCluster() calls a process p completes to the number of times it increments Counter.

Lemma 2. *During any interval in which process p increments* Counter *i times, it completes at least* $\left\lceil \frac{i}{6} \right\rceil - 1$ helpActiveCluster() *calls.*

Recall that at any point during an execution, the currently active cluster is $\lfloor \frac{\text{Counter}}{\delta} \rfloor$. The value of Counter at point t is denoted C_t, and $cl_t = \lfloor \frac{C_t}{\delta} \rfloor$ is the active cluster at point t. A *cluster-k* interval, where $k \in \{0, 1, 2\}$, is an interval throughout which cluster k is active (i.e., $\lfloor \frac{\text{Counter}}{\delta} \rfloor = k$ throughout the interval). We define $idx_t = C_t \bmod \delta$. Hence, for any point t during a cluster-k interval, if $idx_t \in [0,\ \eta - 1]$, then the algorithm is in the cluster-k invalidation phase. If $idx_t \in [\eta,\ \mu + \eta - 1]$, then the algorithm is in the cluster-k move phase. Lastly, if $idx_t \in [\mu + \eta,\ \delta - 1]$, then the algorithm is in the cluster-k update-only phase.

The main goal of the cluster-k invalidation phase is to ensure that after this phase, and while the active cluster remains unchanged, any successful TS[i].SC() operation will set TS[i].cluster to k.

Lemma 3. *Let $k \in \{0, 1, 2\}$, and let $[t, t']$ be a maximal cluster-k interval, such that $C_{t'} \bmod \delta > \eta$. Furthermore, let t^* be the first point during this interval, such that $idx_{t^*} = C_{t^*} \bmod \delta = \eta$. If process p executes a successful* TS[q].SC() *operation during $[t^*, t']$, then that* SC() *writes k to* TS[q].cluster.

Therefore, after the cluster-k invalidation phase, no process can be assigned a timestamp from cluster $(k-1) \bmod 3$. Hence, processes can safely change active timestamps from cluster $(k-1) \bmod 3$ to cluster k, and thus, the cluster-k move phase can begin.

The objective of that phase is to move up to n active timestamps from cluster $(k-1) \bmod 3$ to cluster k. The phase begins immediately after the cluster-k invalidation phase. Therefore, as discussed above, no process gets assigned a timestamp in cluster $(k-1) \bmod 3$, during the cluster-k move phase. Furthermore, at the beginning of the cluster-k move phase at most n active timestamps can be assigned to cluster $(k-1) \bmod 3$. Thus, immediately after this phase, no process's active timestamp is in cluster $(k-1) \bmod 3$.

Lemma 4. *Let k be an element in $\{0, 1, 2\}$, and I a cluster-k interval. If* Counter $\bmod \delta \geq \eta + \mu$ *at some point $t \in I$, then at that point, no active timestamp is in cluster $(k-1) \bmod 3$.*

At the start of the execution, cluster 0 is the active cluster, and initially processes are assigned timestamps in cluster 0 until $\left\lfloor \frac{\text{Counter}}{\delta} \right\rfloor = 1$ for the first time. At this point, cluster 1 becomes active. While this cluster remains active, any process executing an updateTS() call is assigned a timestamp in cluster 1. However, some processes might still have pending updateTS() calls that started before the active cluster changed, which means that they may be assigned timestamps from cluster 0 even after cluster 1 has become active. By Lemma 3, after the cluster-1 invalidation phase, no process can be assigned a timestamp from cluster 0, until the active cluster changes again. Immediately following the cluster-1 invalidation phase, the cluster-1 move phase begins. By Lemma 4, when this phase completes, no process remains assigned to cluster 0. Hence, at this point, all active timestamps are in cluster 1, and they remain in cluster 1 until the active cluster changes. By repeatedly, applying Lemmas 3 and 4 in the same way, we obtain the following.

Lemma 5. *For any $p \in \{0, \ldots, n-1\}$, at any point in a cluster-k interval* TS[p].cluster *is equal to k or $(k-1) \bmod 3$.*

While moving timestamps to the active cluster during a move phase, it is crucial to maintain the dominance relation between any pair of active timestamps. This is guaranteed by the following lemma.

Lemma 6. *An* SC() *executed in line 73 does not affect the dominance order of any entries of the* TS *array.*

Recall that a process p announces that it wants to update its timestamp by changing its announce bit to the complement of its flag. Additionally, while updating p's active timestamp, either p or some other process helping p, sets p's flag to the value of p's announce bit. Therefore, TS[p].flag $\neq$ A[p] only while p wants to update its timestamp.

Lemma 7 states three important properties of the algorithm. Part (a) states that, if some process p executes a helpUpdateTS(q) call (q may be the same as p), then before this call responds, some process updates q's active timestamp, provided that q wants to update its timestamp (i.e., TS[q].flag $\neq$ A[q]). Part (b) shows that during an interval in which p wants to update its timestamp, at most two successful TS[p].SC() operations can be executed. Lastly, part (c) shows that during an interval I, in which Counter is incremented $O(n^3)$ times, some process p executes at least n

complete `helpSystem()` calls. During each complete `helpSystem()` call, process p tries to help a different process update its timestamp or complete an `isEarlier()` call. Thus, any `updateTS()` or `isEarlier()` call that has "announced" that it wants help before the start of interval I, either finishes, or gets helped.

Lemma 7. *Let p be a process, and let t and t' be points in time such that $t < t'$.*

(a) Suppose process p calls `helpUpdateTS(`q`)` *at point t, and the function call responds at point t'. Let $t_r \in [t, t']$ be the first point during the* `helpUpdateTS(`q`)` *call at which p executes the* `read()` *operation in line 10. If* $\mathsf{TS}[q].\mathsf{flag} \neq \mathsf{A}[q]$ *at point t_r, then there exist a point $t^* \in [t_r, t']$ at which* $\mathsf{TS}[q].\mathsf{flag}$ *changes.*

(b) If $\mathsf{TS}[p].\mathtt{flag} \neq \mathsf{A}[p]$ *throughout $[t, t']$, then there are at most two successful* `SC()` *operations on* $\mathsf{TS}[p]$ *during $[t, t']$.*

(c) During an interval in which Counter *gets incremented $3n^3 + 6n^2 + 6n$ times, some process performs at least n complete* `helpSystem()` *calls.*

Using Lemma 7, we can describe the interplay between a process's announce bit and flag. Specifically, for any process p, the pair of bits $(\mathsf{A}[p], \mathsf{TS}[p].\mathsf{flag})$ changes cyclically as follows:

(1) Initially, both bits are zero, and they will remain zero until p executes an `updateTS()` call.
(2) When p executes line 3 during its `updateTS()` call, it causes $\mathsf{A}[p]$ to change, and as a result, $\mathsf{TS}[p].\mathsf{flag} \neq \mathsf{A}[p]$.
(3) During the same `updateTS()` call, p calls `helpUpdateTS(`p`)`. Before p finishes its `helpUpdateTS()` call, p or some other process helping p executes line 13, which causes p's announce bit and flag to become equal again.
(4) After the `updateTS()` call completes, the pair of bits $(\mathsf{A}[p], \mathsf{TS}[p].\mathsf{flag})$ is in the same state as in the beginning of this cycle, except that both bits have been flipped. The cycle repeats.

The interplay of process p's flag and announce bit indicates that during every `updateTS()` call by process p, there exists a point at which some process q (which may be the same as p) performs a successful $\mathsf{TS}[p].\mathtt{SC}()$ operation in line 13. We call this point the *publishing* point of p's `updateTS()` call. Immediately after this, the effect of p's `updateTS()` is reflected by the dominance order of the TS array entries. Prior to executing that $\mathsf{TS}[p].\mathtt{SC}()$ operation, process q executes an `FAI()` operation in line 12 to obtain a new timestamp for p. We call the point of this `FAI()` the linearization point of p's `updateTS()` call.

The *interpreted value* of the MTS object at point t is the lexicographical order of pairs (t_p, p), where p is a process ID, and t_p is obtained as follows: If none of p's `updateTS()` calls linearizes before t, then $t_p = \infty$, and otherwise t_p is the linearization point of p's latest `updateTS()` call satisfying $t_p < t$.

To prove that our algorithm is linearizable, we will show for each `isEarlier(`i, j`)` operation that there is a point during that operation at which the interpreted value of the MTS object is consistent with the return value of the operation. Thus, the `isEarlier(`i, j`)` operation can linearize at that point.

Lemma 8. *Consider an* `isEarlier(`i, j`)` *call that gets invoked and responds at points t and t', respectively. The response of the* `isEarlier()` *call is consistent with the interpreted value of the MTS object at some point $t^* \in [t, t']$.*

The main difficulty with proving this lemma is that the effect of an `updateTS()` operation does not get reflected in the interpreted value of the object at the linearization point of the operation, but only at the publishing point, which occurs later. The following lemma helps us deal with that.

Lemma 9. *Let i and j be two distinct processes, t a point in time, and t^* the latest point before t, such that the publishing point of any* `updateTS()` *call by i or j with linearization point before t^* occurs before t. Then the dominance order of* $\mathsf{TS}[i]$ *and* $\mathsf{TS}[j]$ *at point t is consistent with the interpreted value of the MTS object at point t^**

We now highlight the main ideas behind the proof of Lemma 8. Consider an `isEarlier(`i, j`)` call by process p that gets invoked and responds at points t and t', respectively. While executing lines 35-38, process p checks if either i or j has announced that it wants help completing an `updateTS()` call, and if yes, helps accordingly (as detailed in Section 4.2). This ensures that if either i or j executes an `updateTS()` operation with a linearization point before t, then the operation's publishing point occurs before p finishes line 38.

Let L_p be the interval spanning p's execution of the loop comprising lines 43-54. Suppose that p successfully obtains a snapshot of $\mathsf{TS}[i]$ and $\mathsf{TS}[j]$ at some point $t_s \in L_p$ (in the same way as described in Section 4.2). In this case, p returns a Boolean value that reflects the dominance order between $\mathsf{TS}[i]$ and $\mathsf{TS}[j]$ at point t_s. As discussed above, the publishing points of i's and j's last `updateTS()` calls which linearized before t are before L_p. Therefore, by Lemma 9 p's response is consistent with the interpreted value of the MTS object at some point in $[t, t_s] \subset [t, t']$.

Now suppose p fails to capture a snapshot while executing the loop. In this case, both of $\mathsf{TS}[i]$ and $\mathsf{TS}[j]$ change at least six times during L_p (as explained in Section 4.2). We can prove that if none of these changes is due to an `updateTS()` call published during L_p, then the active cluster wraps around at least once during L_p. Hence, by Lemma 7(c), during some interval $I_q \subsetneq L_p$, some other process q executes `isEarlier(`i, j`)`, and writes the response of this `isEarlier()` call to LookupTable[p].res. Then in line 56, process p returns the value q stored in LookupTable[p].res. Therefore, with a simple induction on the number of `isEarlier()` calls, we can prove that the response of p's `isEarlier()` call is consistent with the interpreted value of the MTS object at some point in $I_q \subsetneq L_p \subsetneq [t, t']$.

Lastly, suppose that p fails to capture a snapshot while executing the loop, and that a long enough interval has not passed to guarantee that some other process has updated LookupTable[p]. During L_p each of i and j must execute at least one complete `updateTS()` call during the first 3 iterations of p's loop, and another one during the second 3 iterations. It it not hard to see that then the response of p's `isEarlier(`i, j`)` call can be arbitrary: Let t_i and t_j be the linearization points of i's and j's `updateTS()` calls, respectively, during the second 3 iterations of the loop. Since each of i and j completed at least one `updateTS()` call before $\min\{t_i, t_j\}$, at point t_i process i has the timestamp dominating all other ones (in the linearization order), and at point t_j process j has it. Thus, if p's `isEarlier(`i, j`)` call returns False, we can linearize this operation immediately after t_i, and otherwise immediately after t_j.

Lemma 8 implies that our MTS object is linearizable. It is easy to see that the step complexity of the algorithm is constant, since both, `updateTS()` and `isEarlier()`, comprise a constant number of shared memory steps. Finally, according to Section 4.2, our algorithm uses one mod φ-`FAI` object, where $\varphi = O(n^3)$, and $O(n)$ LL/SC objects and registers of size $O(\log n)$, each.

Corollary 10. *The algorithm presented in Figures 1 and 2 is a linearizable MTS object with constant step complexity using one* mod φ-`FAI` *object, where $\varphi = O(n^3)$, and $O(n)$ LL/SC objects and registers of size $O(\log n)$, each.*

Theorem 1 follows immediately from this, and the fact that we can implement an LL/SC object from $O(n)$ CAS objects [21], and a mod φ-`FAI` object from a bounded FAA object of size $O(\log(\varphi n))$ bits (see the next section).

6 BOUNDED FAI FROM BOUNDED FAA

In this section, we present our mod φ-FAI implementation, which uses a single bounded FAA object. We assume that the FAA object can store values in $\{0, \ldots, B-1\}$ (i.e., $B = 2^w$ assuming a word-size of w), and that $n \times \varphi < B$.

The algorithm is quite simple (see Figure 3): We define a threshold, $T = n(m-1)$. To increment the mod φ-FAI object, a process first reads the object. If the value returned is smaller than the threshold, then the process performs Counter.FAA(1), and otherwise Counter.FAA($-\varphi + 1$). Then it returns the result of the FAA() operation.

Clearly, as long as the FAA() operations cause no overflows, this is linearizable. We prove that if a process executes FAA(x) at a point when Counter has value v, then $v + x \in \{0, \ldots, B-1\}$ (see Appendix B). This immediately yields the following result.

Theorem 11. *There exists an algorithm with constant step complexity, that implements a linearizable* mod φ*-FAI from a bounded FAA object that can store values in* $\{0, \ldots, B-1\}$*, provided that* $B > \varphi \cdot n$.

```
74 Function FAI():
75   tmp ← Counter.read()
76   if tmp ≥ (φ · n − n) then tmp ← Counter.FAA(1 − φ)
77   else tmp ← Counter.FAA(1)
78   return tmp mod φ
```

Fig. 3. Bounded mod φ − FAI

7 SNAPSHOTS WITH BOUNDED WORD-SIZE

Bashari and Woelfel [7] presented a single-writer snapshot algorithm that supports three operations, Update(), Scan(), and Observe(). The object stores an array of size n, and process p can write value x to the p-th component by calling Update(x). To obtain a snapshot, process p can call Scan(). But contrary to a conventional snapshot object [2, 6], such a Scan() call does not return anything. Instead, process p can later call Observe(i) to determine the value of the i-th component of the array at the point of p's latest preceding scan. Conventional snapshot algorithms usually provide Scan() methods that return the entire state of the snapshot object (e.g., an n-component array). To achieve this efficiently, it is often assumed that the entire state of the array fits into a single memory word. This is not a realistic assumption. In the solution of Bashari and Woelfel, a memory word only needs to be asymptotically large enough to store a single array component and some auxiliary information. However, that auxiliary information contains a counter value, which increases linearly in the number of operations executed on the object, and thus is not bounded.

The algorithm uses a single-writer predecessor data structure $Pred_p$ for each process p, constructed from a single-writer balanced search tree. Process p can insert and remove key-value pairs (with keys from a totally ordered universe) into and from $Pred_p$. Any process can determine the key-value pair with the largest key in $Pred_p$ that is smaller than a given key.

The algorithm also employs an unbounded fetch-and-increment (FAI) object F to keep track of the order of Update() and Scan() operations. To update its snapshot component with value x, process p increments F from f to $f+1$, and then inserts the key-value pair (f, x) into $Pred_p$. To perform a Scan(), a process can read F into a variable f'. A subsequent Observe(i) operation then

only needs to return the value of the predecessor of f' in $Pred_i$. (Additional tricks, such as helping, are necessary to make this a linearizable solution; but these are not affected by our modification.)

The algorithm ensures that each predecessor data structure stores at any point at most m elements, where $m = O(n)$. This is achieved by periodically recycling "unreachable" key-value pairs.

The fact that unbounded counter values obtained from the FAI object F need to be stored, is the only reason why the algorithm requires shared memory words of unbounded size. But the only purpose of F is to associate `Update()` and `Scan()` operations with counter values that can later be used to determine the temporal order of these operations. Hence, it seems natural to replace the FAI with a bounded timestamp object.

Specifically, we can replace F with a variant of our MTS object, where each process maintains $k = O(m \cdot n)$ timestamps. Recall that a process can do that by simply simulating k distinct process IDs (e.g., $kn, \ldots, (k+1)n-1$). Timestamps then act as keys in the predecessor data structure. Instead of incrementing or reading F in an `Update()` or `Scan()` operation, respectively, a process chooses one of its "free" IDs (i.e., one, for which no associated timestamp occurs in any predecessor data structure), and updates the timestamp associated with that ID. It then uses that ID in place of the key when inserting a key-value pair into the predecessor data structure. Thus, in an `Observe(`i`)` operation, a process can determine the value of the latest preceding `Update()` of process i, by comparing timestamps instead of keys in $Pred_i$.

The problem of finding a "free" ID is essentially a memory reclamation problem, which is well understood, and several known solutions with constant step complexity can be applied here (e.g., [4, 5, 8]).

ACKNOWLEDGMENTS

We thank the anonymous reviewers, who provided insightful comments and suggestions. Support is gratefully acknowledged from the Natural Science and Engineering Research Council of Canada (NSERC) under Discovery Grant RGPIN-2019-04852, and the Canada Research Chairs program.

REFERENCES

[1] Karl Abrahamson. 1988. On Achieving Consensus Using a Shared Memory. In *Proceedings of the 7th ACM Symposium on Principles of Distributed Computing (PODC)*. 291–302. https://doi.org/10.1145/62546.62594

[2] Y. Afek, H. Attiya, D. Dolev, E. Gafni, M. Merritt, and N. Shavit. 1993. Atomic Snapshots of Shared Memory. *Journal of the ACM* 40, 4 (1993), 873–890.

[3] Yehuda Afek, Danny Dolev, Eli Gafni, Michael Merritt, and Nir Shavit. 1994. A Bounded First-in, First-Enabled Solution to the l-Exclusion Problem. *ACM Transactions on Programming Languages and Systems* 16 (1994), 939–953. https://doi.org/10.1145/177492.177731

[4] Zahra Aghazadeh, Wojciech Golab, and Philipp Woelfel. 2014. Making Objects Writable. In *Proceedings of the 33rd ACM Symposium on Principles of Distributed Computing (PODC)*. 385–395. https://doi.org/10.1145/2611462.2611483

[5] Zahra Aghazadeh and Philipp Woelfel. 2016. Upper bounds for boundless tagging with bounded objects. In *Proceedings of the 30th International Symposium on Distributed Computing (DISC)*. 442–457. https://doi.org/10.1007/978-3-662-53426-7_32

[6] James H. Anderson. 1993. Composite Registers. *Distributed Computing* 6, 3 (1993), 141–154. https://doi.org/10.1007/BF02242703

[7] Benyamin Bashari and Philipp Woelfel. 2021. An Efficient Adaptive Partial Snapshot Implementation. In *Proceedings of the 40th ACM Symposium on Principles of Distributed Computing (PODC)*. 545–555. https://doi.org/10.1145/3465084.3467939

[8] Guy E. Blelloch and Yuanhao Wei. 2020. LL/SC and Atomic Copy: Constant Time, Space Efficient Implementations Using Only Pointer-Width CAS. In *Proceedings of the 34th International Symposium on Distributed Computing (DISC)*. 5:1–5:17. https://doi.org/10.4230/LIPIcs.DISC.2020.5

[9] Danny Dolev and Nir Shavit. 1997. Bounded concurrent time-stamping. *SIAM Journal on Computing* 26, 2 (1997), 418–455. https://doi.org/10.1137/S0097539790192647

[10] Cynthia Dwork, Maurice Herlihy, Serge Plotkin, and Orli Waarts. 1999. Time-lapse snapshots. *SIAM Journal on Computing* 28, 5 (1999), 1848–1874. https://doi.org/10.1137/S0097539793243685

[11] Cynthia Dwork and Orli Waarts. 1999. Simple and efficient bounded concurrent timestamping and the traceable use abstraction. *Journal of the ACM* 46, 5 (1999), 633–666. https://doi.org/10.1145/324133.324161

[12] Faith Ellen, Panagiota Fatourou, and Eric Ruppert. 2008. The space complexity of unbounded timestamps. *Distributed Computing* 21, 2 (2008), 103–115. https://doi.org/10.1007/s00446-008-0060-6

[13] C. J. Fidge. 1988. Timestamps in Message-Passing Systems That Preserve the Partial Ordering. *Proceedings of the 11th Australian Computer Science Conference* 10, 1 (1988), 56–66.

[14] Michael J. Fischer, Nancy A. Lynch, James E. Burns, and Allan Borodin. 1989. Distributed FIFO allocation of identical resources using small shared space. *ACM Transactions on Programming Languages and Systems* 11, 1 (1989), 90–114. https://doi.org/10.1145/59287.59292

[15] Rainer Gawlick, Nancy A. Lynch, and Nir Shavit. 1992. Concurrent Timestamping Made Simple. In *Theory of Computing and Systems (ISTCS)*. 171–183. https://doi.org/10.1007/BFb0035176

[16] Rachid Guerraoui and Eric Ruppert. 2007. Anonymous and fault-tolerant shared-memory computing. *Distributed Computing* 20, 3 (10 2007), 165–177. https://doi.org/10.1007/s00446-007-0042-0

[17] Sibsankar Haldar and Paul Vitányi. 2002. Bounded concurrent timestamp systems using vector clocks. *Journal of the ACM* 49, 1 (2002), 101–126. https://doi.org/10.1145/505241.505246

[18] Maryam Helmi, Lisa Higham, Eduardo Pacheco, and Philipp Woelfel. 2014. The Space Complexity of Long-Lived and One-Shot Timestamp Implementations. *Journal of the ACM* 61, 1 (2014), 7:1–7:25. https://doi.org/10.1145/2559904

[19] Amos Israeli and Ming Li. 1987. Bounded Time-Stamps. In *Proceedings of the 28th Annual IEEE Symposium on Foundations of Computer Science (FOCS)*. 371–382. https://doi.org/10.1109/SFCS.1987.10

[20] Amos Israeli and Meir Pinhasov. 1992. A Concurrent Time-Stamp Scheme which is Linear in Time and Space. In *Distributed Algorithms*, Vol. 647. 95–109. https://doi.org/10.1007/3-540-56188-9_7

[21] Prasad Jayanti and Srdjan Petrovic. 2003. Efficient and Practical Constructions of LL/SC Variables. In *Proceedings of the 22ed ACM Symposium on Principles of Distributed Computing (PODC)*. 285–294. https://doi.org/10.1145/872035.872078

[22] Leslie Lamport. 1974. A New Solution of Dijkstra's Concurrent Programming Problem. *Commun. ACM* 17 (1974), 453–455. Issue 8. https://doi.org/10.1145/361082.361093

[23] Leslie Lamport. 1978. Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM* 21 (1978), 558–565. Issue 7. https://doi.org/10.1145/359545.359563

[24] Friedemann Mattern. 1989. Virtual Time and Global States of Distributed Systems. *Parallel and Distributed Algorithms* (1989), 215–226.

[25] Glenn Ricart and Ashok K. Agrawala. 1981. An optimal algorithm for mutual exclusion in computer networks. *Commun. ACM* 24 (1981), 9–17. Issue 1. https://doi.org/10.1145/358527.358537

[26] Vivek Shikaripura and Ajay D. Kshemkalyani. 2002. A Simple, Memory-Efficient Bounded Concurrent Timestamping Algorithm. In *Proceedings of the 13th Annual International Symposium on Algorithms and Computation (ISAAC)*. 550–562. https://doi.org/10.1007/3-540-36136-7_48

[27] Paul M. B. Vitanyi and Baruch Awerbuch. 1986. Atomic shared register access by asynchronous hardware. In *Proceedings of the 27th Annual Symposium on Foundations of Computer Science (SFCS '86)*. 233–243. https://doi.org/10.1109/SFCS.1986.11

A CORRECTNESS PROOF

It is easy to see that our implementation has constant step complexity:

Lemma 12. *Functions* `updateTS()` *and* `isEarlier()` *have constant step complexity.*

Proof. Both functions, `isEarlier()` and `updateTS()`, comprise a constant number of shared memory operations, and one call of `helpActiveCluster()`. The `helpActiveCluster()` function comprises a constant number of shared memory operations and one call of `helpMoveTS()`. Due to the for loops in lines 61 and 64, a `helpMoveTS()` call has a step complexity of $\Theta(n)$. In each `helpActiveCluster()` call, only a constant number of steps of a `helpMoveTS()` call are executed. Therefore, `updateTS()` and `isEarlier()` have constant step complexity. □

The remainder of this section is structured as follows. We first, prove Lemma 2. Next, we demonstrate the connection between a process's announce bit and its flag. We then investigate the algorithm's behavior during the invalidation and move phases. Additionally, we describe how the timestamps change throughout an execution, and how an active timestamp changes during an `updateTS()` call. We also discuss what causes the dominance relation between two active timestamps to change. Finally, we define a linearization point for each operation, such that ordering operations based on their linearization points yields a history that is consistent with the sequential specification.

Observation 13. *For each* `updateTS()` *or* `isEarlier()` *function call, a process p can increment* Counter *at most six times.*

Proof. The only shared memory operation in the algorithm that increments Counter is in line 12 of `helpUpdateTS()`. Because of the repeat loop spanning lines 8-13, during every call of this function, Counter can be incremented at most three times. An `updateTS()` call performs `helpUpdateTS()` at most twice, once in line 4, and once in line 15 of its `helpActiveCluster()` call in line 5. Therefore, for every `updateTS()` call, process p can increment Counter at most six times. Similarly, a complete `isEarlier()` call can call `helpUpdateTS()` at most twice, once in line 36 and once in line 38. Thus, during an `isEarlier()` call, process p can increment Counter at most six times as well. □

In the following lemma we show that between any two consecutive `helpActiveCluster()` calls of a process p, it can increment Counter at most 6 times.

Lemma 14. *Between any two consecutive* `helpActiveCluster()` *calls of process p, it can increment* Counter *at most 6 times.*

Proof. There are two shared memory operations in the algorithm that execute a call of `helpActiveCluster()`, line 6 in the `updateTS()` function and line 39 in the `isEarlier()` function. Therefore, after executing a call of `helpActiveCluster()` process p can either respond to an `updateTS()` call, or execute lines 40-56 and then respond to an `isEarlier()` call. Executing lines 40-56 does not increment Counter. Therefore, after responding to a `helpActiveCluster()` call process p does not increment Counter until it responds to its pending `updateTS()` or `isEarlier()` call. Furthermore, after responding to its `isEarlier()` or `updateTS()` call and before invoking another `helpActiveCluster()` call, p can either invoke `updateTS()`, and execute lines 2-5, or invoke `isEarlier()` and execute lines 35-38. By Observation 13, in both cases p can increment Counter at most 6 times. Thus, between any two consecutive `helpActiveCluster()` calls of p, it can increment Counter at most six times. □

Using Lemma 14 and observation 13, we can prove Lemma 2.

Lemma 2. *During any interval in which process p increments* Counter *i times, it completes at least $\lceil \frac{i}{6} \rceil - 1$* `helpActiveCluster()` *calls.*

Proof. From the beginning of the execution until the first time p executes a `helpActiveCluster()` call, it can either execute lines 2-5 during an `updateTS()` call, or execute lines 35-38 during an `isEarlier()` call. By Observation 13, in both cases p can increment Counter at most six times before executing a call of `helpActiveCluster()`. Executing a call of `helpActiveCluster()` does not increment Counter. Furthermore, By Lemma 14 between any two consecutive calls of `helpActiveCluster()` by p, it can increment Counter at most 6 times. Hence, after at most six Counter increments p executes a call of `helpActiveCluster()`. Therefore, in any interval in which p increments Counter i times, it must complete at least $\lceil \frac{i}{6} \rceil - 1$ `helpActiveCluster()` calls. □

Flag and Announce Bit Interplay

In this section we state some useful properties of $A[p]$ and $TS[p]$.flag for a process p, and describe the interplay between them.

Observation 15. *A successful* $TS[p]$.`SC()` *operation in line 33 or 73 does not change* $TS[p]$.flag.

Proof. First we prove the statement for a successful `SC()` executed in line 33. Assume that a successful `SC()` in line 33 changes $TS[p]$.flag. Let b be the flag component returned by the preceding `LL()` on $TS[p]$ in line 29. Since the `SC()` writes b to the flag component of $TS[p]$, at some point after the `LL()` in line 29 and before the `SC()` is executed, $TS[p]$.flag must change. This contradicts the assumption that the `SC()` is successful.

The proof for the case that the successful `SC()` is executed in line 73 is identical to the case where the `SC()` is executed in line 33, except that its preceding `LL()` happens in line 61 (if $p \neq n-1$), or in line 58 (if $p = n-1$). □

We now observe that a process's announce bit only changes due to a write operation in line 3, and its flag changes only due to successful `SC()` operation in line 13.

Observation 16. *Let p be a process.*

(a) If $A[p]$ changes at point t, then at that point process p executes a write operation in line 3.
(b) If $TS[p]$.flag changes at point t, then at that point some process executes an `SC()` *in line 13.*

Proof. Part (a) follows immediately from the fact that the only line in the only shared memory operation in the algorithm, which modifies the value of an array entry $A[p]$, is the write operation by p in line 3.

We now prove part (b). Assume $TS[p]$.flag changes at point t. The algorithm has three shared memory operations that can perform an `SC()` on $TS[p]$ in lines 13, 33 and 73. By Observation 15, an `SC()` in line 29 or 73 does not change $TS[p]$.flag. Therefore, only an `SC()` in line 13 can change $TS[p]$.flag. □

The following shows how the equality relation between a process's flag and announce bit changes, if either its announce bit or flag changes.

Lemma 17. *Let p be a process, and let t be a point in time at which a shared memory operation is executed.*

(a) If $A[p]$ changes at point t, then $TS[p]$.flag $\neq A[p]$ immediately after t.
(b) If $TS[p]$.flag changes at point t, then $TS[p]$.flag $= A[p]$ immediately after t.

Proof. Suppose the lemma is not true, and let t be the first point in time when one of the statements is not true. First assume part (a) is not true at this point. Suppose A[p] changes from $1-b$ to b at point t, and TS[p].flag = A[p] immediately after t. By Observation 16(a) at point t, process p executes the write operation on TS[p] in line 3. Let $t_{LL} < t$ be the point when p executes the preceding LL() in line 2. Then the flag component returned by this LL() is $1-b$. So TS[p].flag = $1-b$ at point t_{LL}. Furthermore, because A[p] is a single-writer register, and p is the only process that can write to it, A[p] = $1-b$ throughout $[t_{LL}, t]$. Thus, by the assumption that TS[p].flag = A[p] immediately after t, at some point $t^* \in (t_{LL}, t)$ process p's flag changes from $1-b$ to b. Then TS[p].flag changes at point t^*, and TS[p].flag $\neq$ A[p] immediately after t^*. Therefore, at point $t^* < t$ part (b) does not hold. This contradicts the assumption that t is the first point in time when one of the statements is not true — a contradiction.

Now assume at point t, part (b) is not true. Suppose TS[p].flag changes from $1-b$ to b at point t, and TS[p].flag $\neq$ A[p] immediately after t. By Observation 16(b), at point t, some process q executes a successful SC() on TS[p] in line 13. Let $t_{LL} < t$ be the point when q executes the preceding LL() in line 9. Since q's SC() is successful, TS[p].flag = $1-b$ throughout $[t_{LL}, t]$. Furthermore, since q does not execute the return statement in line 11, it holds A[p] = b at the point $t_{read} \in [t_{LL}, t]$, when q executes line 10. Therefore, by the assumption TS[p].flag $\neq$ A[p] immediately after t, process p's announce bit must change from b to $1-b$ at some point $t^* \in (t_{read}, t)$. Then A[p] changes at point t^*, and TS[p].flag = A[p] immediately after t^*. Therefore, at point $t^* < t$ part (a) does not hold — a contradiction. □

Lemma 17 and Observation 16 yield the following.

Corollary 18. *Let t be a point in time when a shared memory operation is executed.*

(a) If TS[p].flag $\neq$ A[p] *immediately before t, and* TS[p].flag = A[p] *immediately after t, then* TS[p].flag *changes at point t due to a process executing a successful* SC() *in line 13.*

(b) If TS[p].flag = A[p] *immediately before t, and* TS[p].flag $\neq$ A[p] *immediately after t, then* A[p] *changes at point t due to p executing a write operation in line 3.*

Invalidation Phase

In this section, we focus on the properties of the algorithm during an invalidation phase. We first show that with any successful TS[p].SC() operation that has a matching TS[p].LL() during the same active cluster, the inv component of TS[p] gets updated to the current active cluster number.

Lemma 19. *Consider a cluster-k interval. Suppose during this interval process p performs a* TS[q].LL() *operation at point t, and a successful* TS[q].SC() *operation at point $t' > t$. Then,* TS[q].inv = k, *immediately after t'.*

Proof. The algorithm has three shared memory operations that can perform an SC() on TS[q] in lines 13, 33 and 73. First, we prove the lemma for the case that the SC() is executed in line 13.

If p performs a successful SC() on TS[q] in line 13, its preceding LL() at point t_1, happens in line 9. Then, at point $t_r \in (t_1, t')$ process p reads C_{t_r} from Counter in line 12, and at point t', process p's TS[q].SC() operation sets $\lfloor C_{t_r}/\delta \rfloor$ to the inv component of TS[q]. By the assumption that at point $t < t'$ process p performs a TS[q].LL() operation, $t_1 \geq t$. Hence, $t_r \in (t, t')$. Since (t, t') is a cluster-k interval, $cl_{t^*} = \lfloor \frac{C_{t^*}}{\delta} \rfloor = k$ for any point $t^* \in (t, t')$. Therefore, p's TS[q].SC() operation writes $\lfloor C_{t_r}/\delta \rfloor = cl_{t_r} = k$ to the inv component of TS[q].

The proofs for the other two cases are identical except if the SC() is executed in line 73, its preceding LL() is executed in line 61 or in line 58, and Counter is read in line 62, and if the SC() is executed in line 33, its preceding LL() is executed in line 29, and Counter is read in line 30. □

Lemma 19 immediately yield the following.

Corollary 20. *For any $k \in \{0,1,2\}$ and a cluster-k interval $[t,t']$, if at some point $t^* \in [t,t']$ a successful* TS[p].SC() *operation sets* TS[p].inv $= k$, *then* TS[p].inv $= k$ *throughout* $[t^*,t']$.

The following lemma states during the invalidation phase of an active cluster, if there exists a long enough sub interval during which $\frac{\text{Counter mod } \delta}{\zeta} = i$ for some $i \in \{0,\dots,n-1\}$, then there exists a point where TS[i].inv $= k$.

Lemma 21. *For any $k \in \{0,1,2\}$ consider a cluster-k interval $[t,t']$ during which* Counter *gets incremented at least $6n+1$ times. If throughout this interval* Counter mod $\delta < \eta$ *and* $\frac{\text{Counter mod } \delta}{\zeta} = i$, *for some $i \in \{0,\dots,n-1\}$, then there exists a point $t^* \in [t,t']$ at which* TS[i].inv $= k$.

Proof. Suppose TS[i].inv $\neq k$ throughout $[t,t']$. Since during this interval Counter gets incremented at least $6n+1$ times, there exists a process p that increments Counter at least seven times. Hence, by Lemma 2, process p completes at least one helpActiveCluster() call in (t,t'), and thus, also a conditionalReset() call in line 22. Therefore, process p, during its conditionalReset() call, executes the loop spanning lines 25-33 up to two times. Let $t_1 \in (t,t')$ be the point when p executes line 26 in the first iteration the loop. Thus, p reads C_{t_1} from Counter. Then in line 27, process p computes $idx_{t_1} = C_{t_1} \bmod \delta$ and compares it to η. Moreover, $idx_{t_1} < \eta$, because of the assumption that Counter mod $\delta < \eta$ throughout $[t,t']$. Therefore, p does not execute the return statement in line 27. Then, in line 28, process p calculates $\lfloor \frac{C_t \bmod \delta}{\zeta} \rfloor$ which is i, because of the assumption that $\lfloor \frac{\text{Counter}}{\zeta} \rfloor = i$ throughout $[t,t']$. Thus, at some point $t_{LL} \in (t_1,t')$ process p performs a TS[i].LL() operation in line 29. Then in lines 30-31, process p repeats the same operations as in lines 26-27. Hence, at point $t_2 \in (t_{LL},t')$ process p reads C_{t_2} from Counter. Similarly, $idx_{t_2} < \eta$, since $t_2 \in [t,t']$. Thus, p does not execute the return statement in line 31 either. Then, in line 32, process p compares TS[i].inv to $cl_{t_2} = \lfloor \frac{C_{t_2}}{\delta} \rfloor$. Since t_2 is in $[t,t']$ which is a cluster-k interval, $cl_{t_2} = k$. Therefore, by the assumption that TS[i].inv $\neq k$ throughout $[t,t']$, process p does not execute the return statement in line 32. Hence, at some point $t_{SC} \in (t_2,t')$ process p performs a TS[i].SC() operation in line 33. This SC() must fail, because otherwise, by Lemma 19 at point $t_{SC} \in (t,t')$, process p changes TS[i].inv to k. Process p then repeats the loop another time, and with the same reasoning, p's TS[i].SC() operation in the second iteration the loop must fail as well. Hence, during (t,t') at least two successful TS[i].SC() operations must get executed.

Let q and r be the two (not necessarily distinct) processes that perform the first two of those successful TS[i].SC(), respectively. Furthermore, let t_q, and $t_r > t_q$ be the points when q and r perform their TS[i].SC(), respectively. Since r's SC() is successful, r performs its corresponding LL() at some point $t_{r'} \in (t_q,t_r)$. Furthermore, since $[t_{r'},t_r] \subsetneq (t,t')$, process r's LL() and SC() happen during the same cluster-k interval. Therefore, by Lemma 19, at point $t_r \in (t,t')$ process r writes k to the inv component of TS[i]. This contradicts the assumption that TS[i].inv $\neq k$ throughout (t,t'), because $t_r \in (t,t')$. □

We now show that during the invalidation phase of an active cluster, a long enough sub-intervals exists for any $i \in \{0,\dots,n-1\}$ such that $\lfloor \frac{\text{Counter mod } \delta}{\zeta} \rfloor = i$, and Counter gets incremented at least $6n+1$ times.

Lemma 22. *For any cluster $k \in \{0,1,2\}$, consider a maximal cluster-k interval $[t,t']$. If t' is the first point after t such that $idx_{t'} = C_{t'} \bmod \delta = \eta$, then for any process ID $i \in \{0,\dots,n-1\}$ there exists an interval $[t_1,t_2] \subsetneq [t,t')$ such that throughout it* Counter mod $\delta < \eta$, $\lfloor \frac{\text{Counter mod } \delta}{\zeta} \rfloor = i$, *and* Counter *gets incremented at least $6n+1$ times.*

Proof. Since $[t, t']$ is a maximal cluster-k interval, it begins when cluster k becomes active. Thus, $idx_t = C_t \bmod \delta = 0$. Recall that Counter is a mod 3δ-FNI object, and $\delta = \eta + \mu + \gamma > \eta$. Since t' is the first point after t when Counter mod $\delta = \eta = n \cdot \zeta$, during $[t, t']$ the value of Counter never decreases, and is incremented $n \cdot \zeta$ times. For any $i \in \{0, \ldots, n-1\}$ consider an interval $[t_i, t_{i'})$, such that $idx_{t_i} = C_{t_i} \bmod \delta = i \cdot \zeta$ and $t_{i'}$ is the first point after t_i, such that $idx_{t_{i'}} = C_{t_{i'}} \bmod \delta = (i+1) \cdot \zeta$. Observe that, throughout this interval, $\lfloor \frac{idx_{t_i}}{\zeta} \rfloor \leq \lfloor \frac{\text{Counter mod } \delta}{\zeta} \rfloor < \lfloor \frac{idx_{t_{i'}}}{\zeta} \rfloor$. Therefore, $\lfloor \frac{\text{Counter mod } \delta}{\zeta} \rfloor = i$ throughout $[t_i, t_{i'})$, because $\lfloor \frac{idx_{t_i}}{\zeta} \rfloor = \lfloor \frac{i \cdot \zeta}{\zeta} \rfloor = i$, and $\lfloor \frac{idx_{t_{i'}}}{\zeta} \rfloor = \lfloor \frac{(i+1) \cdot \zeta}{\zeta} \rfloor = i + 1$. Furthermore, observe that the maximum value of $idx_{t_{i'}} = n \cdot \zeta = \eta$ (for $i = n-1$), and the minimum value of $idx_{t_i} = 0 \cdot \zeta = 0$ (for $i = 0$). Therefore, for any $i \in \{0, \ldots, n-1\}$ the interval $[t_i, t_{i'})$ is a sub-interval of $[t, t')$, since during $[t, t')$, the value of Counter mod δ goes from 0 to $\eta - 1 = n \cdot \zeta - 1$. Lastly, since Counter mod δ never decreases during $[t_i, t_{i'}) \subsetneq [t, t']$, during this interval Counter is incremented $C_{t_{i'}} \bmod \delta - C_{x_1} \bmod \delta - 1 = (i+1) \cdot \zeta - i \cdot \zeta - 1 = 6n + 1$. □

Lemmas 21 and 22 immediately yield the following.

Corollary 23. *For any $k \in \{0, 1, 2\}$ and a maximal cluster-k interval $[t, t']$ such that t' is the first point during the interval when* Counter mod $\delta = \eta$*, for each process ID $i \in \{0, \ldots, n-1\}$ there exists a point during $[t, t')$ at which* TS$[i]$.inv $= k$.

Now we show that during a cluster-k invalidation phase, for every process p, there exists a point when a successful SC() operation sets the inv component of p's active timestamp to k.

Lemma 24. *For any $k \in \{0, 1, 2\}$, consider a maximal cluster-k interval $[t, t']$ such that t' is the first point during the interval when* Counter mod $\delta = \eta$*. Then for each $p \in \{0, \ldots, n-1\}$, there exists a point $t_p \in [t, t')$ at which a successful* TS$[p]$.SC() *operation changes* TS$[p]$.inv *to k.*

Proof. Suppose the statement is not true. During an execution, let $[t_1, t_2]$ be the first maximal cluster-k interval such that t_2 is the first point during the interval when Counter mod $\delta = \eta$, and throughout $[t, t')$ no successful TS$[p]$.SC() operation that changes TS$[p]$.inv to k is performed. Since $[t_1, t_2]$ is a maximal cluster-k interval, it begins when cluster k becomes active. Thus, $idx_{t_1} = C_{t_1} \bmod \delta = 0$. The interval $[t_1, t_2]$ is either within the maximal cluster-0 interval that starts at the beginning of the execution, or it happens after the first time cluster-1 becomes active.

First consider the case that interval $[t_1, t_2]$ happens during the first maximal cluster-0 interval. Since the beginning of the execution is when cluster 0 becomes active, point t_1 is at the beginning of execution. Therefore, TS$[p]$.inv $= 2$ at point t_1, since TS$[p]$.inv is initialized to 2. Since by Corollary 23, there exists a point $t^* \in [t_1, t_2)$, at which TS$[p]$.inv $= 0$, there exist a point $t^{**} \in [t_1, t^*] \subset [t_1, t_2]$ at which a successful TS$[p]$.SC() operation changes TS$[p]$.inv() to 0 — a contradiction.

Now consider the case that $[t_1, t_2]$ starts at some point after the first time cluster-1 becomes active. By the assumption that $[t_1, t_2]$ is the first maximal cluster-k interval during which the statement is not true, the statement holds during the maximal cluster-$(k-1) \bmod 3$ interval, $[t_0, t_1)$ that happens immediately before t_1. Therefore, there exists a point $t^* \in [t_0, t_1)$ at which a successful TS$[p]$.SC() operation sets TS$[p]$.inv $= (k-1) \bmod 3$. Thus, by Corollary 20, TS$[p]$.inv $= (k-1) \bmod 3$ immediately before t. Lastly, since by Corollary 23, there exists a point $t_k \in [t_1, t_2)$ at which TS$[p]$.inv $= k$, there must exists a point $t^{**} \in [t_1, t_k]$ at which a successful TS$[p]$.SC() operations changes TS$[p]$.inv to k — a contradiction. □

The following statement is a direct result of Lemmas 19 and 24.

Corollary 25. *For any $k \in \{0, 1, 2\}$, consider a cluster-k interval $[t, t']$, such that $C_t \bmod \delta \geq \eta$. Then for every $p \in \{0, \ldots, n-1\}$,* TS$[p]$.inv $= k$ *throughout $[t, t']$.*

The following statement shows that if at some point during a cluster-k interval a successful SC() operation sets TS[p].$Inv = k$, then as long as the cluster k remains active, no successful TS[p].SC() operation in line 33 is executed.

Lemma 26. *For any $k \in \{0, 1, 2\}$, consider a cluster-k interval $[t, t']$. If at some point $t^* \in [t, t']$ a successful* TS[p].SC() *operation changes* TS[p].inv *to k, then during $(t^*, t']$ no successful* TS[p].SC() *operation in line 33 is executed.*

Proof. Suppose at point $t_{SC} \in (t^*, t']$, process q performs a successful TS[p].SC() operation in line 33 (p and q may be the same). Let $t_{LL} < t_{SC}$ be the point when q executes the preceding LL() in line 29. Since q's SC() is successful, $t_{LL} > t^*$. By Corollary 20, TS[p].inv $= k$ at point $t_{LL} \in [t^*, t_{SC}]$. Furthermore, let $t_r \in (t_{LL}, t_{SC})$ be the point in time when q executes the read operation in line 30, and reads C_{t_r} from Counter. Since $t_r \in (t_{LL}, t_{SC}) \subset [t, t']$ and $[t, t']$ is a cluster-k interval, $\lfloor \frac{C_{t_r}}{\delta} \rfloor = k$. Therefore, TS[$p$].inv $= \lfloor \frac{C_{t_r}}{\delta} \rfloor = k$, and process q, in line 32 executes a return statement. This contradicts the assumption that at point t_{SC} process q executes a TS[p].SC() operation in line 33. □

Lemma 26 and corollary 25 immediately yield the following corollary. The corollary states that if the algorithm is in a move phase or an update-only phase, no successful SC() in line 33 gets executed.

Corollary 27. *Suppose at point t a successful* TS[p].SC() *operation is performed in line 33. Then $idx_t = C_t \bmod \delta < \eta$.*

Furthermore, Lemma 26 directly result following, which states that during a cluster-k invalidation phase, for each active timestamp, at most one successful SC() is executed in line 33.

Corollary 28. *For any $k \in \{0, 1, 2\}$, consider a maximal cluster-k interval $[t, t']$, such that t' is the first point during the interval when* Counter $\bmod \delta = \eta$. *Then for any $p \in \{0, \ldots, n-1\}$, during $[t, t')$ at most one successful* TS[p].SC() *operation in line 33 gets executed.*

Recall that the main purpose of a cluster-k invalidation phase is to guarantee that during the following cluster-k move phase and cluster-k update only phase, no process is assigned a timestamp in cluster-$(k-1) \bmod 3$. Lemma 3 states that the algorithm has guarantees this property, and the following observation is used to prove it.

Observation 29. *Suppose $[t, t']$ is a cluster-k interval for some $k \in \{0, 1, 2\}$. Suppose during this interval p performs a* TS[q].LL() *operation at point t_{LL}, and then a successful* TS[q].SC() *operation at point $t_{SC} > t_{LL}$ in line 13, or line 73. Then* TS[q].cluster $= k$, *immediately after t_{SC}.*

Proof. The proof for the case that p performs the SC() in line 13 is similar to the proof where the SC() is performed in line 73. Below we write the proof for the first case and note the differences in parentheses.

If p performs a successful SC() in line 13 (73), its preceding LL() at point t_{LL}, happens in line 9 (61 or line 58 if $q = n-1$). Let $t_r \in (t_{LL}, t_{SC})$ be the point in time when p reads C_{t_r} from Counter in line 12 (62). Then, at point t_{SC}, process p writes $\lfloor \frac{C_{t_r}}{\delta} \rfloor = cl_{t_r}$ to TS[q].cluster. Since $t_r \in (t_{LL}, t_{SC}) \subseteq [t, t']$, by the assumption that $[t, t']$ is a cluster-k interval, $cl_{t_r} = k$. □

Lemma 3. *Let $k \in \{0, 1, 2\}$, and let $[t, t']$ be a maximal cluster-k interval, such that $C_{t'} \bmod \delta > \eta$. Furthermore, let t^* be the first point during this interval, such that $idx_{t^*} = C_{t^*} \bmod \delta = \eta$. If process p executes a successful* TS[q].SC() *operation during $[t^*, t']$, then that* SC() *writes k to* TS[q].cluster.

Proof. Suppose the lemma is not true and let $t_{SC} \in [t^*, t']$ be the first point in time when p performs a successful TS[q].SC() operation that sets TS[q].cluster to $z \neq k$. The algorithm has three shared memory operations that can perform an SC() on TS[q] in lines 13, 33 and 73. By Corollary 27, no successful SC() in line 33 gets executed during $[t^*, t']$. Therefore, p's SC() must have been executed in line 13 or line 73. By Observation 29, process p's preceding LL() must have happened at some points $t_{LL} < t$, since otherwise p's SC() would write k to TS[q].cluster. Furthermore, by Lemma 24, there exists a point $t^* \in [t, t^*) \subsetneq (t_{LL}, t_{SC})$ when a successful TS[q].SC() operation is executed. This contradicts the assumption that p's SC() at point t_{SC} is successful. □

Lastly, observe that the conditionalReset() calls do not affect any active timestamp's cluster or index.

Observation 30. *If p performs a successful* TS[q].SC() *operation in line 33, then the* SC() *does not change* TS[q].cluster *or* TS[q].index.

Proof. Assume that a successful SC() in line 33 changes the cluster or index component of TS[q]. Since the SC() sets the same cluster and index components that are returned by the preceding LL() in line 29, TS[q].cluster or TS[q].index must change at some point between when the LL() and the SC() are executed. This contradicts the assumption that the SC() is successful. □

Move Phase

We continue by analyzing the properties of the algorithm during a move phase.

We define TS_t to be the value of the TS array at point t, and we denote the set of all active timestamps in cluster k at point t with $\mathsf{TS}_{k,t}$. More formally, $\mathsf{TS}_{k,t} = \{s \in \mathsf{TS}_t \mid s.\text{cluster} = k\}$.

The following lemma states that during a cluster-k move phase, if some process p in line 58 performs TS[$n-1$].LL(), and TS[n-1].cluster $\neq (k-1) \bmod 3$ at that point, then p's subsequent TS[$n-1$].VL() operation in line 66 returns True.

Lemma 31. *Let $k \in \{0, 1, 2\}$, and let $[t, t']$ be a cluster-k interval such that* Counter mod $\delta \in [\eta, \eta + \mu - 1]$ *throughout the interval. Suppose at point $t_{LL} \in [t, t']$ process p performs an* LL() *on* TS[$n-1$] *in line 58, and* TS[$n-1$].cluster $\neq (k-1) \bmod 3$ *at that point. If p performs the subsequent* TS[$n-1$].VL() *operation at some point during $(t_{LL}, t']$, then its* VL() *returns* True.

Proof. Suppose at some point $t_{\text{VL}} \in (t_{LL}, t']$ process p preforms the TS[$n-1$].VL() operation in line 66, and it returns False. Hence, at some point $t^* \in (t_{LL}, t_{\text{VL}})$, another process q, performs a successful SC() on TS[$n-1$]. There are three shared memory operations in the algorithm that can modify the TS array: Lines 13, 33 and 73. Since process $n-1$ never calls the updateTS() function, process $n-1$ never executes line 3. Thus, by Lemma 17(a), throughout the execution A[$n-1$] does not change. Hence, by corollary 18, throughout the entire execution, TS[$n-1$].flag = A[$n-1$]. Therefore, any process executing a helpUpdateTS($n-1$) call, executes the return statement in line 11, and thus, no process performs an SC() on TS[$n-1$] in line 13. Furthermore, since Counter mod $\delta \geq \eta$ throughout $[t, t']$ by Corollary 27 during $(t_{LL}, t_{VL}) \subsetneq [t, t']$ no successful SC() in line 33 is executed. Hence, at point t^* process q performs a successful SC() on TS[$n-1$] in line 73. Let $t_{qLL} < t^*$ be the point when process q performs its corresponding LL() on TS[$n-1$] (in line 58). Since in line 73, process q executes the SC() operation on TS[$n-1$], it must add $n-1$ to its local variable oldCluster in line 68. Therefore, TS[$n-1$].cluster = $(k-1) \bmod 3$ at point t_{qLL}. Furthermore, TS[$n-1$].cluster $\neq (k-1) \bmod 3$ throughout $[t_{LL}, t^*]$, because TS[$n-1$].cluster $\neq (k-1) \bmod 3$ at point t_{LL}, and by Lemma 3 any successful TS[$n-1$].SC() operation during $[t_{LL}, t^*]$ sets TS[$n-1$].cluster to k. Thus, $t_{qLL} < t_{LL}$. Therefore, there exists a point during $[t_{qLL}, t_{LL}] \subsetneq [t_{qLL}, t^*]$ when a successful SC() on TS[$n-1$] changes its cluster component. This contradicts the assumption that q's SC() at point t^* is successful. □

The following lemma shows that every complete helpMoveTS() that is executed during a cluster-k a move phase, moves at least one active timestamp from cluster $(k-1) \bmod 3$ to cluster k, provided that there exists at least one active timestamp in cluster $(k-1) \bmod 3$ at the invocation of the helpMoveTS() call. Intuitively, this means if at least n complete and separate helpMoveTS() calls happen during a move phase then all timestamps from previous cluster get reassigned to the active cluster.

Lemma 32. *For any $k \in \{0,1,2\}$, consider a cluster-k interval $[t,t']$. Suppose process p starts and finishes a* helpMoveTS() *call at point $t_s \in [t,t')$ and $t_f \in (t_s,t']$, respectively, such that throughout the interval the value of* Counter mod δ *is in $[\eta, \eta+\mu-1]$. If $\left|\mathsf{TS}_{(k-1) \bmod 3,s}\right| > 0$, then $\left|\mathsf{TS}_{(k-1) \bmod 3,s}\right| > \left|\mathsf{TS}_{(k-1) \bmod 3,f}\right|$, and $\left|\mathsf{TS}_{k,f}\right| > \left|\mathsf{TS}_{k,s}\right|$.*

Proof. Suppose the statement is not true. Therefore, either $\left|\mathsf{TS}_{(k-1) \bmod 3,s}\right| \leq \left|\mathsf{TS}_{(k-1) \bmod 3,f}\right|$ or $\left|\mathsf{TS}_{k,f}\right| \leq \left|\mathsf{TS}_{k,s}\right|$. First, observe that since $[t_s,t_f] \subset [t,t']$ is a cluster-k interval, and Counter mod $\delta \geq \eta$ throughout $[t_s,t_f]$, then by Lemma 3, during this interval, if any process performs successful SC() on an entry of the TS array, it writes k to that entry's cluster component. Therefore,

$$\begin{array}{l}\text{during } [t_s,t_f], \text{ no successful SC() on an active timestamp}\\ \text{assigned to cluster } (k-1) \bmod 3 \text{ is performed.}\end{array} \qquad (*)$$

While executing the helpMoveTS() during $[t_s,t_f]$, first, in line 58, process p performs an TS$[n-1]$.LL(). Then in lines 59-60, process p reads c from Counter, and checks that $c \bmod \delta$ is in $[\eta, \eta+\mu-1]$. Since Counter mod $\delta \in [\eta, \eta+\mu-1]$ throughout $[t_s,t_f]$, process p does not execute the return statement in line 60. Then in line 61, process p performs an LL() on every entry in the TS array, except TS$[n-1]$. Let t_1 be the point immediately after p completes executing line 61. Hence, during $[t_s,t_1]$, process p preforms an LL() on every entry in the TS array. Process p then, by executing lines 62-63, reads c' from Counter, and checks that $c' \bmod \delta \in [\eta, \eta+\mu-1]$. Again, since Counter mod δ is in $[\eta, \eta+\mu-1]$ throughout $[t_s,t_f]$, process p does not execute the return statement in line 63. By executing lines 64-65, process p performs a VL() on every active timestamp i that was in cluster $(k-1) \bmod 3$, when p executed the TS$[i]$.LL() during $[t_s,t_1]$. Therefore, since by $(*)$ during $[t_s,t_f]$ no successful SC() on an active timestamp assigned to cluster $(k-1) \bmod 3$ is executed, p does not execute the return statement in line 65. Then, in line 66 process p performs a TS[n-1].VL() operation, and execute a return statement if TS[n-1].VL() returns False. If TS[*n-1*].cluster $= (k-1) \bmod 3$ when p performed the TS[n-1].LL() operation in line 58, then by $(*)$, The TS[n-1].VL() operation in line 66 returns True. Also, If TS[*n-1*].cluster $= (k-1) \bmod 3$ when p performed the TS[n-1].LL() operation, then by Lemma 31, the TS[n-1].VL() operation in line 66 returns True. Thus, either way, p does not execute the return statement in line 66. By executing line 68, process p stores the index i, of every TS$[i]$ where TS$[i]$.cluster $= (\lfloor \frac{c'}{\delta} \rfloor - 1) \bmod 3$. Since $[t_s,t_f]$ is a cluster-k interval, $\lfloor \frac{c'}{\delta} \rfloor = k$. Hence, by $(*)$, and the assumption that $\left|\mathsf{TS}_{(k-1) \bmod 3,s}\right| > 0$, process p stores at least one index in oldCluster. Hence, oldCluster $\neq$, and p does not execute the return statement in line 69. Then by executing line 70, process p assigns some process ID $q \in$ oldCluster to the variable j. Therefore, at some point $t_{SC} \in (t_1,t_f]$, process p performs a TS$[q]$.SC() operation in line 73. Let $t_{LL} \in [t_s,t_1]$ be the point when p performs the TS$[q]$.LL() operation while executing lines 58-61. Since by $(*)$, TS$[q]$ does not change throughout $[t_{LL},t_{SC}]$, at point t_{SC} process p's SC() on TS$[q]$ is successful. Thus, at point t_{SC} process p changes TS$[q]$.*Cluster* from $(k-1) \bmod 3$ to $\lfloor \frac{c'}{\delta} \rfloor = k$. As such, $\left|\mathsf{TS}_{(k-1) \bmod 3,s}\right| > \left|\mathsf{TS}_{(k-1) \bmod 3,t_{SC}}\right|$, and $\left|\mathsf{TS}_{k,t_{SC}}\right| > \left|\mathsf{TS}_{k,s}\right|$. Finally, $\left|\mathsf{TS}_{(k-1) \bmod 3,f}\right| \leq \left|\mathsf{TS}_{(k-1) \bmod 3,t_{SC}}\right|$, and $\left|\mathsf{TS}_{k,f}\right| \geq \left|\mathsf{TS}_{k,t_{SC}}\right|$, since by Lemma 3, any successful SC() operation on any index i of the TS array, sets TS$[i]$.*Cluster* $= k$. Therefore, $\left|\mathsf{TS}_{(k-1) \bmod 3,s}\right| > \left|\mathsf{TS}_{(k-1) \bmod 3,f}\right|$, and $\left|\mathsf{TS}_{k,f}\right| > \left|\mathsf{TS}_{k,s}\right|$ — a contradiction. □

Using Lemma 2 it is easy to see the relation between the number of Counter increments during a move phase and the number of complete `helpMoveTS()` calls.

Lemma 33. *For any $k \in \{0, 1, 2\}$, consider a cluster-k interval $[t, t']$. If $idx_t = C_t \bmod \delta = \eta$, and t' is the first point after t such that $idx_{t'} = C_{t'} \bmod \delta = \eta + \mu$, then during $[t, t')$, some process executes at least n complete* `helpMoveTS()` *calls.*

Proof. Since Counter is mod3δ-FNI object, where $\delta > \eta + \mu$, throughout $[t, t']$ the value of Counter never decreases. Furthermore, during this interval Counter is incremented $idx_{t'} - idx_t = \mu$ times. Therefore, Counter is incremented at least $\mu - 1$ times during $[t, t')$. Thus, there exists a process p that increments Counter at least $\left\lceil \frac{\mu-1}{n} \right\rceil$ during $[t, t')$. Since $\mu = 6n^3 + 6n^2 + 2$, during $[t, t')$ process p increments Counter at least $6n^2 + 6n + 1$ times. Therefore, by Lemma 2, during this interval p executes at least $\left\lceil \frac{6n^2+6n+1}{6} \right\rceil - 1 = n^2 + n$ complete `helpActiveCluster()` calls. Hence, p executes $\kappa \cdot (n^2 + n)$ steps of `helpMoveTS()` function calls. Since only one these calls can be invoked before t, and a complete `helpMoveTS()` call requires at most $\kappa \cdot n$ steps, p performs $\lfloor \frac{\kappa \cdot (n^2+n)}{\kappa \cdot n} \rfloor - 1 = \lfloor \frac{\kappa \cdot n \cdot (n+1)}{\kappa \cdot n} \rfloor - 1 = n$ complete `helpMoveTS()` calls in $[t, t')$. □

Lemma 4. *Let k be an element in $\{0, 1, 2\}$, and I a cluster-k interval. If* Counter $\bmod \delta \geq \eta + \mu$ *at some point $t \in I$, then at that point, no active timestamp is in cluster $(k-1) \bmod 3$.*

Proof. Let t_1 be the latest point before t such that $idx_{t_1} = C_{t_1} \bmod \delta = \eta$, and let $t_2 < t$ be the first point after t_1 such that $idx_{t_2} = C_{t_2} \bmod \delta = \eta + \mu$. It is trivial to see that the interval $[t_1, t_2]$ is a cluster-k interval. By Lemma 33, there exists a process p such that during $[t_1, t_2) \subsetneq [t_1, t]$ process p executes n complete `helpMoveTS()` calls. Hence, by Lemma 32, during each of p's `helpMoveTS()` calls, if there exists at least one active timestamp in cluster $(k-1) \bmod 3$ when p invokes the `helpMoveTS()` call, then before the `helpMoveTS()` call responds, at least one of those active timestamps is moved to cluster k. Furthermore, since $[t_1, t]$ is a cluster-k interval, and Counter $\bmod \delta \geq \eta$ throughout $[t_1, t]$, during this interval any successful `SC()` on an entery in the TS array, sets its cluster to k. Thus, during this interval no active timestamp i, is changed such that i.cluster is set to $(k-1) \bmod 3$. Therefore, since at point t_1 there can be at most n active timestamps in cluster $(k-1) \bmod 3$, after n complete `helpMoveTS()` calls (at point t_2), no active timestamp is in $(k-1) \bmod 3$. Lastly, since during $[t_2, t] \subsetneq [t_1, t]$ no active timestamp gets reassigned to cluster $(k-1) \bmod 3$, at point t no process has an active timestamp assign to cluster $(k-1) \bmod 3$. □

The next statement follows directly from Lemmas 3 and 4. Simply put, it states that if at the beginning of the move phase during an active cluster k, no active timestamp is in cluster $(k-2) \bmod 3$, then during the following update-only phase, all active timestamps are in cluster k.

Corollary 34. *For any $k \in \{0, 1, 2\}$, consider a maximal cluster-k interval $[t, t']$, such that $idx_t = C_t \bmod \delta = \eta$, and $idx_{t'} \geq \eta + \mu$. Let t^* be the first point after t such that $idx_{t'} = \eta + \mu$. If $\left|\mathsf{TS}_{(k-2) \bmod 3, t}\right| = 0$, then for any $i \in \{0, \ldots, n-1\}$, $\mathsf{TS}[i]$.cluster $= k$ throughout $[t^*, t']$.*

The following lemma states that a successful `SC()` in line 73 happens only if the algorithm is in a move phase.

Lemma 35. *Suppose at some point t, a process p executes a successful $\mathsf{TS}[q]$.`SC()` in line 73. Then $idx_t = C_t \bmod \delta \in [\eta, \eta + \mu - 1]$.*

Proof. Suppose process p in line 73 performs a successful `SC()` on $\mathsf{TS}[q]$ at some point t such that $C_t \bmod \delta = idx_t \notin [\eta, \eta + \mu - 1]$. Let $t_{LL} < t$ be the point when p performs the preceding `LL()` on $\mathsf{TS}[q]$ in line 61 or line 58, and let $t_{read} \in (t_{LL}, t)$ be the point when p subsequently

performs the read() operation in line 62. Since p does not execute the return statement in line 63, $idx_{t_r} = C_{t_r} \bmod \delta \in [\eta,\ \eta+\mu-1]$. Then in line 70, process p must assign q to the variable j. Hence, TS[q].cluster = $(\lfloor \frac{C_{t_r}}{\delta} \rfloor - 1) \bmod 3 = (cl_{t_r} - 1) \bmod 3$, at point t_{LL}. Since $idx_{t_r} \in [\eta,\ \eta+\mu-1]$, and $idx_t \notin [\eta,\ \eta+\mu-1]$, there exists at least one point during $[t_r, t]$ at which, Counter mod $\delta \geq \eta+\mu$. Let $t^* \in [t_r, t]$ be the first of such points. Since by Lemma 4 at point t^* no process is assigned to cluster $(cl_{t_r} - 1) \bmod 3$, there exists a point in $[t_{LL}, t^*] \subset [t_{LL}, t]$ when a successful SC() on TS[q] change its cluster number. This contradicts the assumption that at point t process p's SC() is successful. □

Similar to the invalidation phase, in the following lemma we show that during a cluster-k move phase, for every process i at most one successful TS[i].SC() operation gets executed in line 73

Lemma 36. *For any $k \in \{0,1,2\}$, consider a cluster-k interval $[t, t']$ such that* Counter mod $\delta \in [\eta,\ \eta+\mu-1]$ *throughout this interval. Then for each $i \in \{0,\ldots,n-1\}$, at most one successful* TS[i].SC() *operation is executed in line 73, during $[t, t']$.*

Proof. Suppose during $[t, t']$, two successful SC() operations on TS[i] are executed in line 73. Let p and q be the two, not necessarily distinct processes that execute these successful SC() operations at points t_p and t_q, respectively, such that $t_p < t_q$. Since process q's SC() at point t_q is successful, at some point $t_{LL} \in (t_p, t_q)$ process q must have performed a preceding LL() on TS[i] in line 61, or in line 58. Furthermore, since $[t, t']$ is a cluster-k interval, and Counter mod $\delta \geq \eta$ throughout $[t, t']$, by Lemma 3 during $[t_p, t_{LL}] \subsetneq [t, t']$ any successful SC() on TS[i] sets TS[i].cluster to k. Hence, TS[i].cluster = k at point t_{LL}. Therefore, after q subsequently executes line 68 process i is not added to the local variable oldCluster. Since in line 73, process q performs an SC() on an index $j \in$ oldCluster of the TS array, at point t_q process q performs an SC() on TS[j] for some $j \neq i$. — a contradiction. □

The following lemma states that if a process p that executes a successful SC() in line 73, during a cluster-k move phase, then p has performed an LL() on all entries of the TS array, during the same active cluster k.

Lemma 37. *For any $k \in \{0,1,2\}$, consider a maximal cluster-k interval $[t, t']$ Suppose at some point $t_{SC} \in [t, t')$, process p performs a successful* SC() *on* TS[q] *in line 73. Then, during an interval $I \in [t, t_{SC})$ process p performs a* TS[*n-1*].LL() *operation in line 58, and an* LL() *operation on all other entries of the* TS *array in line 61.*

Proof. Suppose process p performs a successful SC() in line 73 at point t_{SC}, during a helpMoveTS() call, and performs the preceding LL() on TS[$n-1$] in line 58, at some point $t^* < t$. Let $t_c > t^*$ be the point when process p reads C_{t_c} from Counter in line 62. Since in line 49 process p does not execute the return statement, $idx_{t_c} \in [\eta,\ \eta+\mu-1]$. Also, since process p does not execute the return statement in line 66, there exists a point $t_{VL} > t_c$ when p's VL() on TS[$n-1$] in line 66 returns True.

First suppose $t_c > t$. Let t_1 be the first point after t such that $idx_{t_1} = C_{t_1} \bmod \delta = \eta$. Then, $t < t_1 \leq t_c < t_{VL}$, since $idx_{t_c} \in [\eta,\ \eta+\mu-1]$. Therefore, during $[t, t_1] \subsetneq [t^*, t_{VL}]$, by Lemma 24, at least one successful SC() is performed on TS[$n-1$]. This contradicts the assumption that at point t_{VL}, process p's VL() returns True.

Now suppose $t_c \leq t$. Let $t_q < t_c$ be the point in time when process p performs an LL() on TS[q] in line 61 or line 58 if $q = n-1$. Since $t_{SC} > t$ and by Lemma 35, $idx_{t_{SC}} \in [\eta,\ \eta+\mu-1]$, there exists a point $t_1 \in [t, t_{SC})$ such that $idx_{t_1} = \eta$. Therefore, during $[t, t_1] \subsetneq [t_{LL}, t_{SC}]$ by Lemma 24 at least one successful SC() is performed on TS[q]. This contradicts the assumption that at point t_{SC}, process p's SC() on TS[q] is successful.

Hence, p's preceding TS[$n-1$].LL() operation in line 58, must happen at some point $t_{nLL} \in [t, t_{SC})$. Then at some point after t_{nLL}, during the same helpMoveTS() call, process p executes line 61, and performs an LL() on every entry of TS array except TS[$n-1$]. Therefore, during the interval $[t_{nLL}, t_{SC})$, process p performs a TS[n-1].LL() operation in line 58, and an LL() operation on all other entries of the TS array in line 61. □

Timestamp Invariants

In the following two statements we show how all active timestamps change during an execution.

We first show that during any active cluster k no successful SC() can assign a timestamp in cluster $(k-2) \bmod 3$ to a process.

Lemma 38. *For any $k \in \{0,1,2\}$, consider a cluster-k interval $[t, t']$. Let $t_{SC} \in [t, t']$ be a point in time at which a successful* TS[q].SC() *operation changes* TS[q].cluster. *Then* TS[q].cluster $\neq (k-2) \bmod 3$ *immediately after t_{SC}.*

Proof. Suppose at point $t_{SC} \in [t, t']$ process p performs a successful SC() on TS[q] changes its cluster component to $(k-2) \bmod 3$. There are three shared memory operations in the algorithm that can modify the TS array: Lines 13, 33 and 73. Since by Observation 30, a successful SC() executed in line 33 does not change the cluster component of any active timestamp, p must have executed the SC() in line 13 or line 73. The proof for the two cases are identical, we prove for the first case, and note any differences with the proof of the second case in parentheses.

Suppose at point t_{SC}, process p executes line 13 (73). Let $t_{LL} < t_{SC}$ be the point when p performs its corresponding LL() in line 9 (61 or line 58 if $q = n-1$). Furthermore, let $t_r \in (t_{LL}, t_{SC})$ be the point in time when p rs C_{t_r} from Counter in line 12 (62). Then, $cl_{t_r} = \lfloor \frac{C_{t_r}}{\delta} \rfloor = (k-2) \bmod 3$, since p writes $\lfloor \frac{C_{t_r}}{\delta} \rfloor$ to TS[q].cluster. Since point t is in a cluster-k interval, $cl_t = \lfloor \frac{C_t}{\delta} \rfloor = k$. Therefore, $\lfloor \frac{\text{Counter}}{\delta} \rfloor$ must change at least twice during $[t_r, t]$. Hence, there exists two points $t_1, t_2 \in (t_r, t)$ where t_1 is the first point after t_r, such that $cl_{t_1} = (k-1) \bmod 3$, and t_2 is the first point after t_1, such that $idx_{t_2} = C_{t_2} \bmod \delta = \eta$. Lastly, by Lemma 24 there exists a point in $[t_1, t_2] \subsetneq (t_{LL}, t_{SC})$ when a successful SC() on TS[q] is performed. This contradicts with the assumption that at point t_{SC}, process p's SC() is successful. □

Now, we prove that during an execution all active timestamps are always assigned to two consecutive clusters.

Lemma 5. *For any $p \in \{0, \ldots, n-1\}$, at any point in a cluster-k interval* TS[p].cluster *is equal to k or $(k-1) \bmod 3$.*

Proof. Let t_0 be the point in time that the execution begins. The statement is initially true, since $cl_{t_0} = \lfloor \frac{C_{t_0}}{\delta} \rfloor = 0$ and every process has an active timestamp in cluster zero.

Let t_1 be the first point in time after t_0 when the active cluster changes. Hence, $cl_{t_1} = \lfloor \frac{C_{t_1}}{\delta} \rfloor = (cl_{t_0} + 1) \bmod 3 = 1$, and $cl_{t^*} = cl_{t_0}$ for any $t^* \in [t_0, t)$. Furthermore, let t_2 be the first point after t_1 when the active cluster changes again. Thus, $cl_{t_2} = \lfloor \frac{C_{t_2}}{\delta} \rfloor = (cl_{t_1} + 1) \bmod 3 = 2$.

By Lemma 38, during $[t_0, t_1)$ no process is assigned to cluster $(cl_{t_0} - 2) \bmod 3 = 1$. Hence, during $[t_0, t_1)$ the invariant holds. Let $t_i \in [t_0, t_1)$ be a point in time such that $idx_{t_i} = C_{t_i} \bmod \delta = \eta$ (end of the invalidation phase during active cluster 0) Let $t_m \in [t_i, t_1)$ be the first point such that $idx_{t_m} = C_{t_m} \bmod \delta > \eta + \mu$ (end of the move phase during active cluster 0). Since $\left|\text{TS}_{1,t_i}\right| = 0$, by Corollary 34 at point t_m all active timestamps are assigned to cluster $cl_{t_i} = 0$. Furthermore, since during $[t_m, t_1)$, by Lemma 3 any successful SC() on any active timestamp sets its cluster number to $cl_{t_0} = 0$, at point t_1 all processes are assigned to cluster $cl_{t_0} = 0$. Therefore, immediately

before the start of the new active cluster 1 all active timestamps are assigned to cluster 0. With this configuration, identical arguments can be made to prove that the invariant holds during $[t1, t_2)$, and immediately before t_2, the MTS object is in the similar configuration to its configuration immediately before t_1. Meaning immediately before t_2 all active timestamps are assigned to cluster 1. □

UpdateTS() function

In this section, we will analyze the properties of the `updateTS()` function.

The following observation states that during an interval in which a process executes n `helpSystem()` calls, for every process q at least one `helpUpdateTS(`q`)` gets executed.

Observation 39. *Suppose during an interval $[t, t']$ a process p performs n complete* `helpSystem()` *calls. Then for each $q \in \{0, \ldots, n-1\}$, there exists a point $t^* \in [t, t']$ when p executes a* `helpUpdateTS(`q`)` *call.*

Proof. By executing lines 15 and 20 during each `helpSystem()` call, p performs a complete `helpUpdateTS()` for a different process ID, going through all IDs in a round-robin fashion. Therefore, after n complete `helpSystem()` calls, process p has performed a complete `helpUpdateTS(`q`)` for every $q \in \{0, \ldots, n-1\}$. □

The following two statements relate announce bit changes to `helpSystem()` calls, and `helpUpdateTS()` calls.

Lemma 40. *Consider an interval $[t, t']$ during which process p's announce bit changes i times. Then, p performs at least $i-1$ complete* `helpSystem()` *calls.*

Proof. Since during $[t, t']$ process p's announce bit changes i, by Observation 16(a) during this interval p performs the `write()` operation in line 3 of the `updateTS()` function i times. Since p must complete the `updateTS()` call before calling it again, p executes lines 3-5 at least $i-1$ times. Therefore, p completes the `helpSystem()` call (in line 5) at least $i-1$ times. □

Lemma 41. *Suppose* Counter *gets incremented i times during an interval $[t, t']$. Then, there exists two not necessarily distinct processes p and q such that during $[t, t']$, process p performs at least $\left\lceil \frac{\left\lceil \frac{\lfloor i/3 \rfloor}{n} \right\rceil - 2}{n} \right\rceil$ complete* `helpUpdateTS(`q`)` *calls, each of which increment* Counter *at least once.*

Proof. Since Counter can only be increment at most three times during a `helpUpdateTS()` call, and the only shared memory operation in the algorithm that increments Counter is in line 12, at least $\lfloor \frac{i}{3} \rfloor$ such calls overlap with the interval $[t, t']$. Therefore, there exists some process p such that it performs at least $\left\lceil \frac{\lfloor i/3 \rfloor}{n} \right\rceil$ of these `helpUpdateTS()` function calls. Since at most two of p's `helpUpdateTS()` calls can be not completely in $[t, t']$, during $[t, t']$ process p performs at least $\left\lceil \frac{\lfloor i/3 \rfloor}{n} \right\rceil - 2$ complete `helpUpdateTS()` function calls such that each of them increments Counter at least once. Lastly, there must exist a process ID q (p and q are not necessarily distinct) such that during $[t, t']$, process p performs at least $\left\lceil \frac{\left\lceil \frac{\lfloor i/3 \rfloor}{n} \right\rceil - 2}{n} \right\rceil$ complete `helpUpdateTS(`q`)` calls, each of which increment Counter at least once. □

The next lemma states that during an interval I, if three successful `TS[`p`].SC()` operations get executed, and none of them is due to p' timestamp being updated in line 13, then Counter get incremented $O(n^3)$ times during I.

Lemma 42. *Suppose during an interval I, three successful* SC() *operations are performed on* TS[p] *and none of them are executed in line 13. Then* Counter *is incremented at least $3n^3 + 6n^2 + 6n$ times during I.*

Proof. Consider the shortest possible interval I', during which three successful SC() operations are performed on TS[p] and none of them are executed in line 13 There are three shared memory operations in the algorithm that can modify the TS array: Lines 13, 33 and 73. By the assumption that none of the three successful SC() operations is performed in line 13, during I' two of the successful SC() operations must be performed in the same line.

Case 1: Suppose at least two of the successful SC() operations are performed in line 33. Let $t_1 \in I'$ be the point when the first successful SC() on TS[p] in line 33 is executed, and let $t_2 \in I'$ be the point when the last such operation is performed. Then, by Corollary 27, these SC() operations can only happen if the algorithm is in an invalidation phase. Furthermore, by Corollary 28, these SC() operations cannot happen during the same invalidation phase. Therefore, the shortest interval $[t_1, t_2]$ happens when t_1 is at the end of an invalidation phase (i.e., $idx_{t_1} = C_{t_1} \bmod \delta = \eta - 1$), and t_2 is at the beginning of the next invalidation phase (i.e., $idx_{t_2} = C_{t_2} \bmod \delta = 0$). Thus, Counter is incremented at least $\delta - \eta = \eta + \mu + \gamma - \eta = \mu + \gamma$ times during $[t_1, t_2]$. Lastly, $\mu + \gamma > 3n^3 + 6n^2 + 6n$, since $\mu = 6n^3 + 6n^2 + 2$ and $\gamma = 3n^3 + 4n$.

Case 2: Suppose at least two of the successful SC() operations are performed in line 73. Let $t_1 \in I'$ be the point when the first successful SC() on TS[p] in line 73 gets executed, and let $t_2 \in I'$ be the point when the last such operation is performed. Then, by Lemma 35 these SC() operations can only happen when the algorithm is in a move phase. Furthermore, by Lemma 36, these SC() operations cannot happen during the same move phase. Therefore, the shortest interval $[t_1, t_2]$ happens when t_1 is at the end of a move phase (i.e., $idx_{t_1} = C \bmod \delta = \eta + \mu$), and t_2 is at the beginning of the next move phase (i.e., $idx_{t_2} = \eta$). Thus, Counter is incremented at least $\delta - (\eta + \mu) + \eta = (\eta + \mu + \gamma) - (\eta + \mu) + \eta = \gamma + \eta$ times during $[t_1, t_2]$. Lastly, $\gamma + \eta = 3n^3 + 6n^2 + 6n$, since $\eta = 6n^2 + 2n$ and $\gamma = 3n^3 + 4n$.

Therefore, Counter is incremented at least $3n^3 + 6n^2 + 6n$ times during any interval in which three successful TS[p].SC() operations gets executed, such that none of them is executed in line 13. □

We now restate and prove Lemma 7.

Lemma 7. *Let p be a process, and let t and t' be points in time such that $t < t'$.*

(a) Suppose process p calls helpUpdateTS(q) *at point t, and the function call responds at point t'. Let $t_r \in [t, t']$ be the first point during the* helpUpdateTS(q) *call at which p executes the* read() *operation in line 10. If* TS[q].flag $\neq$ A[q] *at point t_r, then there exist a point $t^* \in [t_r, t']$ at which* TS[q].flag *changes.*

(b) If TS[p].flag $\neq$ A[p] *throughout $[t, t']$, then there are at most two successful* SC() *operations on* TS[p] *during $[t, t']$.*

(c) During an interval in which Counter *gets incremented $3n^3 + 6n^2 + 6n$ times, some process performs at least n complete* helpSystem() *calls.*

Proof. Suppose the lemma is not true. Let t' be the first point in time such that there exists an interval $[t, t']$ during which one of the parts is not true.

First suppose process p's helpUpdateTS(q) call starts and finishes at points t and t', respectively. Let $t_r \in [t, t']$ be the first point during helpUpdateTS(q) call when p performs the read() operation in line 10. For the purpose of contradiction, assume TS[q].flag $\neq$ A[q] at point t_r and TS[q].flag does not change during $[t_r, t']$ (i.e., part (a) is not true). Since TS[q].flag $\neq$ A[q], process p does not execute the return statement in line 11. Furthermore, p's SC() in line 13 must fail, because otherwise TS[q].flag would change. Therefore, during $[t, t']$ process p's SC() on TS[q]

fails three times, once in each iteration of the repeat loop. Hence, there should be at least three successful SC() on TS[q] during (t_r, t'). Let $t^* \in (t_r, t')$ be the point when the third successful SC() is executed. Since TS[p].flag $\neq$ A[p] during $[t_r, t^*] \subseteq [t, t']$, at point $t^* < t'$ part (b) does not hold. This contradicts the assumption that t' is the first point in time when one of the statements is not true.

Now assume TS[p].flag $\neq$ A[p] during $[t, t']$, and there are three successful SC() operations in this interval (i.e., part (b) is not true). By Lemma 17(b), none of the SC() operations is executed in line 13. Therefore, by Lemma 42, Counter is incremented at least $3n^3 + 6n^2 + 6n$ times during $[t, t']$. Thus, by part (c) there exists a process q that performs at least n complete helpSystem() calls. Hence, by Observation 39, there exists an interval $[t_s, t_f] \subseteq (t_1, t_2)$ when process q performs a complete helpUpdateTS(p) call. Since, TS[p].flag $\neq$ A[p] throughout $[t_s, t_f] \subseteq (t, t')$, at point $t_f < t'$ part (a) does not hold. This contradicts the assumption that t' is the first point in time when one of the statements is not true.

Lastly, consider the case where Counter is incremented $3n^3 + 6n^2 + 6n$ times during $[t, t']$ and no process performs at least n complete helpSystem() calls (i.e, part (c) is not true). Since Counter is incremented $3n^3 + 6n^2 + 6n$ times during $[t, t']$, by Lemma 41, there exists two processes p and q (p and q might be the same process) such that during $[t, t']$ process p performs at least $n + 2$ complete helpUpdateTS(q) calls. Let t_s^i be the points when p starts the i-th helpUpdateTS(q) call and let t_f^i the point when p finishes that call. Furthermore, Let $t_r^i \in [t_s^i, t_f^i]$ be the first point during the i-th helpUpdateTS(q) call when p performs the read() operation in line 10. Since during this helpUpdateTS(q) call process p increments Counter at least once, TS[q].flag $\neq$ A[q] at point t_r^i. Otherwise, p performs the return statement in line 11 and would not increment Counter. Therefore, by the assumption that until t' all the statement are true, and by Part (a), there exists a point $t_e^i \in [t_r^i, t_f^i]$ when q's flag change and TS[q].flag = A[q] immediately after t_e^i. Otherwise, at point $t_f^i < t'$ Part (a) does not hold. Therefore, during each separate interval $[t_r^i, t_e^{i+1}]$ for $i \in [0, n{+}1]$, there exists at least one point, t^* when a shared memory operation is executed such that TS[q].flag = A[q] immediately before t^* and TS[q].flag $\neq$ A[q] immediately after t^*. By Corollary 18(b) at those points process q's announce bit changes. Since during $[t, t']$ there are at least $n + 1$ such points, q's announce bit changes at least n. Therefore, by Lemma 40 during $[t, t']$ process q performs at least n complete helpSystem() calls. This contradicts the assumption that during $[t, t']$ no process performs at least n complete helpSystem() calls. □

Lemma 43. *Let p be a process, then throughout the execution of the algorithm, the pair (*A[p], TS[p].flag*) changes in the following way (see also Figure 4).*

(1) Initially, TS[p].flag *equals* A[p]*, and they will remain equal until p executes line 3.*
(2) Process p executes line 3 during an updateTS() *call. As a result of that,* A[p] *changes, and immediately after that* TS[p].flag $\neq$ A[p].
(3) During the same updateTS() *call, p calls* helpUpdateTS(p)*. Before p finishes its* helpUpdateTS() *call, its announce bit and flag become equal again.*

Proof. Part (1) follows immediately from Corollary 18(b), part (2) from Lemma 17(a), and part (3) from Lemma 7a. □

Lemma 44. *Consider an* updateTS() *call by process q that starts and finishes at points t_s and t_f, respectively. Suppose process p performs a successful* SC() *on* TS[q] *in line 13 at point $t_{SC} \in [t_s, t_f]$. Let $t^* < t_{SC}$ be the point when p performs the preceding* FAI() *operation in line 12. Then t^* is in the (t_s, t_f) interval.*

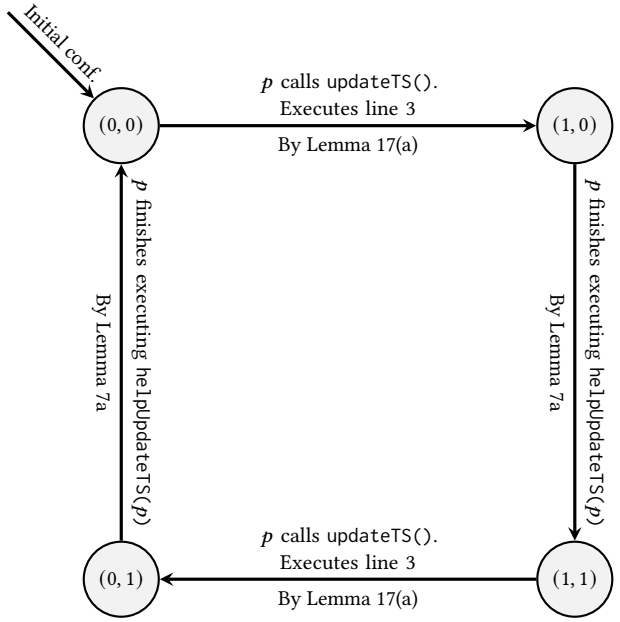

Fig. 4. A diagram illustrating how a process's flag and announce bit changes during any execution

Proof. Let $t_{LL} < t^*$ be the point when process p performs the LL() in line 9, corresponding to its SC() at point t_{SC}. Also, let $t_r < t^*$ be the point when p performs the preceding read() operation in line 10. Thus, $t_{LL} < t_r < t^*$. Since p's SC() is successful, TS[q] does not change during (t_{LL}, t_{SC}). Furthermore, TS[p].flag $\neq$ A[p] at point t_r, because p does not execute the return statement in line 11. By Lemma 43, TS[p].flag is not equal to A[p], only during an updateTS() call by process q. Therefore, t_r must be after the invocation of an updateTS() call by process q. Hence, $t_s < t_r < t^*$. Also, since TS[p].flag = A[p] after the updateTS() call responds, by Lemma 17(b), TS[q] changes at some point before t_f. Thus, $t_{SC} < t_f$, because TS[q] does not change throughout $[t_{LL}, t_{SC}]$. Therefore, $t^* \in [t_s, t_f]$. □

Dominance relation between active timestamps

There are three functions in the algorithm that might write to the TS array: helpUpdateTS(), helpMoveTS(), and conditionalReset(). In this section we will show that only a helpUpdateTS() call can affect the dominance order between any pair of active timestamp.

We first show that during any cluster-k interval, if a process gets assigned a timestamp in cluster k, its active timestamp remains in cluster k, as long as the active cluster does not change.

Lemma 45. *For any $k \in \{0, 1, 2\}$, consider a cluster-k interval $[t, t']$. Suppose* TS[q].cluster = k *at some point $t^* \in [t, t']$. Then* TS[q].cluster = k *throughout $[t^*, t']$.*

Proof. Suppose at some point $t_x \in (t^*, t']$ process p performs a successful SC() on TS[q] such that TS[q].cluster $\neq k$. By Lemma 5 throughout $[t^*, t'] \subset [t, t']$ all process are assigned to cluster k or $(k-1)$ mod 3. Hence, p's TS[q].SC() operation at point t_x must change TS[q].cluster to $(k-1)$ mod 3. By Lemma 3 any successful SC() at some point t_s such that $idx_{t_s} = C_{t_s} \bmod \delta \geq \eta$ writes k to TS[q].cluster. Hence, $idx_{t_x} = C_{t_x} \bmod \delta < \eta$. There are three shared memory operation in the algorithm that modify the TS array: Lines 13, 33 and 73. By Observation 30 a successful SC()

in line 33 does not change any active timestamp's cluster component. Furthermore, since $idx_{t_x} < \eta$, by lemma 35, at point t_x, no successful SC() in line 73 is executed. Hence, at point t_x process p must have executed the SC() in line 13. Let $t_{LL} < t_x$ be the point when process p performs the preceding TS[q].LL() operation in line 9, and let $t_r \in (t_{LL}, t_x)$ be the point when process p reads C_{t_r} from Counter by executing the FAI() operation in line 12. Then $cl_{t_r} = \lfloor \frac{C_{t_r}}{\delta} \rfloor = (k-1) \bmod 3$, since at point t_x process p writes $\lfloor \frac{C_{t_r}}{\delta} \rfloor$ to TS[q].cluster. Therefore, at point t_r, the cluster component of TS[q] is not k, because by Lemma 5, at this point TS[q].cluster is $cl_{t_r} = (k-1) \bmod 3$ or $(cl_{t_r} - 1) \bmod 3 = (k-2) \bmod 3$. Since $t_x > t^*$ and $t_{LL} < t_r$ there exists at least one point during $[t_{LL}, t_x]$ when a successful SC() on TS[q] changes its cluster number. This contradicts the assumption that at point t_x process p's SC() is successful. □

In Lemma 37, we have shown that a process p that performs a successful SC() in line 73 obtains its local view of the TS array (lines 58-66) during the same active cluster. The following statement extends the former and shows that the local view that process p obtains is consistent with all active timestamps in the previously active cluster at some point while p executes lines 58-66.

Lemma 46. *For any $k \in \{0, 1, 2\}$, consider a cluster-k interval I Suppose at some point $t_{SC} \in I$, process p, while performing a* helpMoveTS() *call, executes a successful* SC() *operation in line 73. Let L_p be the interval spanning p's execution of lines 58-66 during the same* helpMoveTS() *call. Then there exists a point $t^* \in L_p$ when* Counter mod $\delta \geq \eta$, *and for every $q \in \{0, \ldots, n-1\}$ if* $\mathsf{TS}_{t^*}[q]$.cluster $= (k-1) \bmod 3$, *then* localTS[q] $= \mathsf{TS}_{t^*}[q]$.

Proof. Suppose at point $t_{SC} \in I$, process p performs a successful SC() in line 73 and the lemma is not true. By Lemma 37, the interval L_p is in the same maximal cluster-k interval as t_{SC}. During L_p, process p performs an LL() on all entries of the TS array and stores their values in the localTS array (lines 58 and 61). Let $t_r \in L_p$ be the point when p executes the read() operation in line 62. Since p does not execute the return statement in line 63, Counter mod $\delta \in [\eta, \mu + \eta - 1]$. Let $[t_1, t_2] \subsetneq L_p$ be the interval spanning p's execution of the loop comprising lines 64-65. Since during $[t_r, t_2] \subsetneq L_p$ the active cluster does not change, the value of Counter mod δ during this interval never decreases. Therefore, Counter mod $\delta \geq \eta$, throughout $[t_1, t_2]$. Furthermore, during $[t_1, t_2]$, process p performs a VL() in line 65, on every active timestamp that was in cluster $(k-1) \bmod 3$ when p previously preformed an LL() on it. Since p does not execute the return statement in line 65, all of p's VL() operations during $[t_1, t_2]$, return True. Hence, there exists a point $t^* \in [t_1, t_2]$ when Counter mod $\delta \geq \eta$ and localTS[q] $= \mathsf{TS}_{t^*}[q]$ for every $q \in \{0, \ldots, n-1\}$ such that localTS[q].cluster $= (k-1) \bmod 3$. By the assumption that the lemma is not true, at point t^*, there must exist an index i such that localTS[i].cluster $\neq (k-1) \bmod 3$ and TS[i].cluster $= (k-1) \bmod 3$.

Let $t_{LL} < t_1$ be the point when process p performs an LL() on TS[i]. Since L_p is a cluster-k interval, and TS[i].cluster $\neq (k-1) \bmod 3$ at point $t_{LL} \in L_p$, then by Lemma 5, TS[i].cluster $= k$ at point t_{LL}. Therefore, by Lemma 45, TS[i].cluster remains k throughout $[t_{LL}, t_2]$. This contradicts the assumption that TS[i].cluster $= (k-1) \bmod 3$ at point $t^* \in [t_1, t_2] \subsetneq [t_{LL}, t']$. □

The following two lemmas show that TS[p].index $= \perp$ at any point t if and only if process p has not performed any updateTS() call until point t.

Lemma 47. *Suppose at point t a successful* SC() *is executed on* TS[p] *in line 13. Then throughout the rest of the execution* TS[p]index $\neq \perp$.

Proof. Suppose TS[p].index $= \perp$ at some point $t^* > t$. Since line 13 cannot set TS[p].index $= \perp$, immediately TS[p].index $\neq \perp$ after t. Therefore, there exists at least one point after t such that a successful SC() on TS[p] sets its index to $\perp$. Let $t^{**} \leq t^*$ be the first of these points, and let q the

process that performs the SC(). Therefore, TS[p].index $\neq \perp$ throughout (t, t^{**}). There are three shared memory operations in the algorithm that can modify the TS array: Lines 13, 33 and 73. Since any successful SC() in line 13 cannot set TS[p].index to $\perp$, at point t^{**} a successful SC() happens in either line 33 or line 73. An SC() in either of these lines set TS[p].index to $\perp$ only if TS[p].index $= \perp$ at the point of their preceding LL() operation. Thus, their preceding LL() operation must happen before t, because TS[p].index $\neq \perp$ throughout $[t, t^{**}]$. This contradicts the assumption that at point t^{**}, process p's SC() on TS[p] is successful. □

Lemma 48. *For any point t and any process q, if form the beginning of execution until the point t, process p does not perform any* updateTS() *call, then* TS[p].index $= \perp$ *at point t.*

Proof. Let t_0 be the beginning of execution. Suppose p does not call updateTS() throughout $[t_0, t]$ and TS[p].index $\neq \perp$ at point t. Initially TS[p].index $= \perp$. Therefore, there exists at least one point during $[t_0, t]$ when a successful SC() on TS[p] changes TS[p].index. Let $t^* \in [t_0, t]$ be the first of these points and let q be the process that performs the SC(). Thus, TS[p].index $= \perp$ throughout $[t_0, t^*)$. There are three shared memory operations in the algorithm that can modify the TS array: Lines 13, 33 and 73. Since p never calls the updateTS() function, p never executes line 3. Thus, by Lemma 17(a), A[*n-1*] does not change throughout $[t_0, t]$. Hence, by corollary 18, throughout the entire execution, TS[q].flag $=$ A[q]. Therefore, no process performs a successful SC() on TS[p] in line 13. Furthermore, by Observation 30 a successful SC() in line 33 does not change any active timestamp's index. Therefore, at point t^*, process q performs a successful SC() in line 73. However, q in line 73 sets TS[q].index $\neq \perp$ only if TS[p].index $\neq \perp$ at the point of q's preceding LL() on TS[p] (see line 72). This is a contradiction because TS[p].index $= \perp$ throughout $[t_0, t^*)$. □

As a result of Lemmas 47 and 48, it is easy to see that for any pair of process (i, j) as long as at least one of them has not invoked an updateTS() call, a successful SC() in line 73 does not affect the order of dominance between TS[i] and TS[j]. Thus, for the statements below until Lemma 6, we consider processes that have obtained at least one timestamp (i.e., TS[i].index $\neq \perp$).

The following two observations show that TS[p].index is less than zero if and only if the last successful TS[p].SC() operation that changed TS[p].index happens in line 73.

Observation 49. *Suppose at point t process p performs a successful* SC() *on* TS[q] *in line 73. If* TS[q].index $\neq \perp$ *immediately before t, then at point t process p sets* TS[q].index *to a negative value.*

Proof. By executing line 73, process p writes the value of its newIndex local variable to TS[q].index. This value is last set when p executes the preceding operations in line 72. By executing line 72, process p assign some value $i \leq -1$ to newIndex variable if TS[q].index $\neq \perp$ at the point of p's preceding LL() operations preceding on TS[q]. Thus, we only have to show that TS[q].index $\neq \perp$ at the point of p's preceding LL(). This is trivially true based on the assumptions that TS[q].index $\neq \perp$ immediately before t, and p's SC() at point t is successful. □

Observation 50. *Suppose at point t process p performs a successful* SC() *on* TS[q] *that changes its index to a negative value. Then p's* SC() *at this point is executed in line 73.*

Proof. There are three shared memory operations in the algorithm that can modify the TS array: Lines 13, 33 and 73. A successful SC() in line 13 cannot write a negative value to the index component of any active timestamp. Furthermore, by Observation 30, a successful SC() in line 33 does not change the index component any active timestamp. Hence, only a successful SC() in line 73 can change the index component of process q's timestamp to a negative value. □

The following statement is the last lemma we need to prove that a helpMoveTS() call does not change the order of dominance between active timestamp. It states that a successful SC() in line 73, changes an active timestamp in the previously active cluster to a timestamp in the currently active cluster, such that it remains dominated by every active timestamp in the active cluster.

Lemma 51. *For any $k \in \{0, 1, 2\}$, consider a maximal cluster-k interval $[t, t']$. Suppose at point $t_{SC} \in [t, t']$, process p performs a successful* SC() *in line 73 on* TS$[q]$. *Furthermore, suppose* TS$[q]$.index $\neq \perp$ *immediately before t. Let r be the process that has the active timestamp with the minimum index i, among all active timestamps in cluster k immediately before t_{SC}. Then at point t_{SC}, process p writes some value less than i to* TS$[q]$.index.

Proof. Suppose at point t_{SC} process p performs a successful SC() on TS$[q]$ in line 73, and writes some value $z \geq i$ to TS$[q]$.index. By Observation 49, $z < 0$. Hence, TS$[j]$.cluster $= k$ and TS$[j]$.index $= i < 0$ immediately before t_{SC}. Let t_1 be the point when process p executes line 58 during the same helpMoveTS() call that it executes the SC() at point t_{SC}. Furthermore, let $t_2 \in (t_1, t_{SC})$ be the point when process p is poised to execute line 67. By Lemma 46, at some point $t^* \in [t_1, t_2]$, the value of Counter mod $\delta \geq \eta$, and process p has a local view of the TS array that is consistent with $\mathsf{TS}_{(k-1) \bmod 3, t^*}$. Moreover, by Lemma 37, $[t_1, t_2]$ happens during the same maximal cluster-k interval as t_{SC}. Therefore, Counter mod $\delta \geq \eta$ throughout $[t^*, t_{SC}]$. Thus, by Lemma 3, no process gets assigned to cluster $(k-1)$ mod 3 during $[t^*, t_{SC}]$. By executing line 70 during the same helpMoveTS() call, process p selects TS$[q]$ as the most dominant active timestamp in cluster $(k-1)$ mod 3. Thus, TS$[q]$ dominates every other active timestamp in cluster $(k-1)$ mod 3 at point t^*. Furthermore, TS$[q]$ dominates every other active timestamp in cluster $(k-1)$ mod 3 during $[t^*, t_{SC}]$, since TS$[q]$ does not change during $[t^*, t_{SC}]$ and no process gets assigned to cluster $(k-1)$ mod 3 during this interval.

Let $t_r < t^*$ be the point when p performs the preceding LL() on TS$[r]$ in line 61. Then, there must exist a point $t_y \in (t_r, t_{SC})$, when some process q performs a successful SC() on TS$[r]$ that changes its index to i. Otherwise, process p by executing line 72, sets minIndex $< i$, and at point t_{SC} changes TS$[q]$.*index* to minIndex.

By Observation 50, at point t_y process s performs the SC() in line 73. Then by Lemma 46, at some point $t^{**} < t_y$ process s has a local view of the TS array that is consistent with $\mathsf{TS}_{(k-1) \bmod 3, t^{**}}$. With the same reasoning as above, TS$[r]$ dominates every other active timestamp in cluster $(k-1)$ mod 3 throughout $[t^{**}, t_y]$. Therefore, the interval $[t^{**}, t_y]$ must be completely before $[t^*, t_{SC}]$. Hence, $t_y < t^*$. By Lemmas 5 and 37, TS$[r]$.cluster is either k or $(k-1)$ mod 3 at point t_r.

First, suppose at point t_r, process p reads TS$[r]$.cluster $= k$ by executing the LL() on TS$[r]$. Then by Lemma 45, TS$[r]$.cluster $= k$ throughout $[t_r, t']$. This is a contradiction since TS$[r]$.cluster must be $(k-1)$ mod 3 throughout $[t^{**}, t_y]$, and $[t^{**}, t_y]$ overlaps $[t_r, t']$ (because $t_y > t_r$).

Now suppose at point t_r, process p reads TS$[r]$.cluster $= (k-1)$ mod 3 by executing the LL() on TS$[r]$. Therefore, there exists a point $t_{VL} \geq t^*$ when process p performs a VL() on TS$[r]$ in line 65. Since p does not execute the return statement in line 65, at point t_{VL} process p's VL() on TS$[r]$ must return True. This is a contradiction, since t_r is the last point when process p performs an LL() on TS$[r]$ and $t_y \in (t_r, t_{VL})$. □

In the following lemma we prove that a helpMoveTS() function call does not change the order of dominance between active timestamps.

Lemma 6. *An* SC() *executed in line 73 does not affect the dominance order of any entries of the* TS *array.*

Proof. Suppose at point t_{SC} process p performs a successful SC() in line 73 on TS$[i]$ such that it changes the dominance relation between TS$[i]$ and TS$[j]$. Let TS$[i]^-$ denote the value of TS$[i]$

immediately before t_{SC}, and $\mathsf{TS}[i]^+$ denote the value of $\mathsf{TS}[i]$ immediately after it. By Lemma 35, p's SC() happens in during a cluster-k move phase (i.e., $idx_{t_{SC}} = C_{t_{SC}} \bmod \delta \in [\eta, \eta + \mu - 1]$). Let $t < t_{SC}$ be the latest point such that $idx_t = \eta$. By Lemmas 37 and 46, at some point $t^* \in [t, t_{SC}]$ process p has a local view of the TS array that is consistent with $\mathsf{TS}_{(k-1) \bmod 3, t^*}$. Process p, in line 70, selects $\mathsf{TS}[i]$ as the dominant active timestamp assigned to cluster $(k-1) \bmod 3$. Thus, $\mathsf{TS}[i].\mathsf{cluster} = (k-1) \bmod 3$, and $\mathsf{TS}[i].\mathsf{index}$ is larger than the index component of any other active timestamp in cluster $(k-1) \bmod 3$ at point t^*. Furthermore, $\mathsf{TS}[i]^-.\mathsf{cluster} = (k-1) \bmod 3$, because $\mathsf{TS}[i]$ does not change during $[t^*, t_{SC}]$. Also, by Lemma 3, $\mathsf{TS}[i]^+.\mathsf{cluster} = k$, since $idx_{t_{SC}} \geq \eta$. If either $\mathsf{TS}[i]^-.\mathsf{index} = \perp$ or $\mathsf{TS}[j].\mathsf{index} = \perp$ it is trivial to see that the dominance relation between $\mathsf{TS}[i]$ and $\mathsf{TS}[j]$ is not affected. Hence, consider the case where $\mathsf{TS}[i]^-.\mathsf{index} \neq \perp$, and $\mathsf{TS}[j].\mathsf{index} \neq \perp$.

First suppose $\mathsf{TS}[j] \ll \mathsf{TS}[i]$ immediately before t_{SC}, and $\mathsf{TS}[i] \ll \mathsf{TS}[j]$ immediately after t_{SC}. By Lemma 5, at point t_{SC} all active timestamps are in cluster $(k-1) \bmod 3$ or k. Then, $\mathsf{TS}[j].\mathsf{cluster} = (k-1) \bmod 3$ immediately before t_{SC}, since $\mathsf{TS}[i]^-.\mathsf{cluster} = (k-1) \bmod 3$ and $\mathsf{TS}[i]$ dominates $\mathsf{TS}[j]$ immediately before t_{SC}. Furthermore, $\mathsf{TS}[j].\mathsf{cluster} = (k-1) \bmod 3$ immediately after t_{SC}, because $\mathsf{TS}[j]$ does not change at point t_{SC}. This contradicts the assumption that $\mathsf{TS}[i] \ll \mathsf{TS}[j]$ immediately after t_{SC}, since $\mathsf{TS}[i]^+.\mathsf{cluster} = k$, which dominates $\mathsf{TS}[j].\mathsf{cluster}$.

Now suppose $\mathsf{TS}[i] \ll \mathsf{TS}[j]$ immediately before t_{SC}, and $\mathsf{TS}[j] \ll \mathsf{TS}[i]$ immediately after t_{SC}. Again by Lemma 5, at point t_{SC} all active timestamps are in cluster $(k-1) \bmod 3$ or k. Then, $\mathsf{TS}[j].\mathsf{cluster} = k$ at point t^*, since at this point $\mathsf{TS}[i]$ dominates every active timestamp in cluster $(k-1) \bmod 3$. Furthermore, by Lemma 45, $\mathsf{TS}[j].\mathsf{cluster}$ remain k throughout $[t^*, t]$. Let i be the $\mathsf{TS}[j].\mathsf{index}$, immediately before t. Then, by Lemma 51 at point t process p sets $\mathsf{TS}[i].\mathsf{index} < i$. This contradiction the assumption that $\mathsf{TS}[j] \ll \mathsf{TS}[i]$ immediately after t, since $\mathsf{TS}[j].\mathsf{cluster} = \mathsf{TS}[i].\mathsf{cluster}$ and $\mathsf{TS}[i].\mathsf{index} < \mathsf{TS}[j].\mathit{Index}$. □

Now we can easily show that only a successful SC() in line 13 can affect the dominance order between active. The following lemma states this property.

Lemma 52. *Let t be the point when a shared memory operation is executed. If $\mathsf{TS}[i] \ll \mathsf{TS}[j]$ immediately before t, and $\mathsf{TS}[j] \ll \mathsf{TS}[i]$ immediately after t, then at point t a successful* SC() *in line 13, is executed on either $\mathsf{TS}[i]$ or $\mathsf{TS}[j]$.*

Proof. Since at point t the dominance relation between $\mathsf{TS}[i]$ and $\mathsf{TS}[j]$ changes, at point t the cluster or the index component of one of them must change. Hence, at point t a successful SC() is executed on either $\mathsf{TS}[i]$ or $\mathsf{TS}[j]$ that changes their cluster or index component. There are three shared memory operations in the algorithm that can modify the TS array: lines 13, 33 and 73. By Observation 30 a successful SC() in line 33 does not change the cluster or the index component of any active timestamp. Furthermore, by Lemma 6, a successful SC() executed in line 73 on either $\mathsf{TS}[i]$ or $\mathsf{TS}[j]$, does not change dominance order between $\mathsf{TS}[i]$ and $\mathsf{TS}[j]$. Hence, at point t a successful SC() on $\mathsf{TS}[i]$ or $\mathsf{TS}[j]$ must be executed in line 13. □

Linearizability

During an interval I, that spans an updateTS() call by process p, by Lemma 43, there exists a point when some process q (p and q may be the same) executes a successful $\mathsf{TS}[p]$.SC() operation in line 13. Furthermore, by Lemma 44, process q's preceding FAI() operation, during the same helpUpdateTS(p) call happens during I as well. Hence, the linearization point of an updateTS() call is between its invocation and response. In the remainder of this section, we provide useful statement to prove Lemma 8, and finally provide a complete proof for that lemma.

Let i and j be two processes. We say $i \prec j$ at point t, if i comes before j in the total order of the interpreted value of the MTS object at point t.

The following lemma shows that immediately after the publishing point t, of some process i's updateTS() call, TS[i] dominates every other active timestamp j, provided that $j \prec i$ at point t, and at this point j has no updateTS() which is linearized before t but not published before t.

Lemma 53. *For any $k \in \{0,1,2\}$, consider a maximal cluster-k interval $[t,t']$. Suppose at point $t_p \in [t,t']$, process p performs a successful* SC() *on* TS[i] *in line 13. For any $j \in \{0,\dots,n-1\}$, where $j \neq i$, if $j \prec i$ at point t_p, and at this point, j has no* updateTS() *call which is linearized before t_p but not published, then* TS[j] $\ll$ TS[i] *immediately after t_p.*

Proof. Suppose $j \prec i$ at point t and TS[i] $\ll$ TS[j] immediately after t_p. First observe that since $j \prec i$ at point t_p, and i has an updateTS() which is linearized before t_p, process j's last updateTS() call (before t_p) must linearize before i's. Furthermore, since at point t_p, process j has no updateTS() which is linearized before t_p but not published, by Lemma 47, at this point TS[j].index $\neq \perp$. Also, TS[i].index $\neq \perp$ immediately after t_p. Since $[t,t']$ is a maximal cluster-k interval, and any maximal cluster-k interval must begin when cluster k becomes active, $idx_t = C_t \bmod \delta = 0$.

Let $t_f < t_p$ be the point when p performs the preceding FAI() operation in line 12. Then at point t_p, process p writes $\lfloor \frac{C_{t_f}}{\delta} \rfloor = cl_{t_f} = r$ to TS[i].cluster and $C_{t_f} \bmod \delta = idx_{t_f} = i$ to TS[i].index. Since TS[i] $\ll$ TS[j] immediately after t_p, then either TS[j].cluster dominates r, or TS[j].cluster = r and TS[j].index $> i$. By Lemma 5, at point t_p all processes are assigned to cluster k or $(k-1) \bmod 3$. Therefore, immediately after t_p, either TS[i].cluster = TS[j].cluster = k and TS[i].index < TS[j].index, or TS[i].cluster = $(k-1) \bmod 3$ and TS[j].cluster = k.

Case 1: suppose TS[i].cluster = TS[j].cluster = k, and TS[j].index > TS[i].index ≥ 0 immediately after t_p. Let $t_q < t_p$ be the last point when some process q performs a successful SC() on TS[j] that changes TS[j].cluster or TS[j].index. Thus, q's SC() sets TS[j].cluster = k, and TS[j].index > 0. Hence, $t_q > t$, since by Lemma 5, TS[j].cluster $\neq k$ immediately before t.

There are three shared memory operations in the algorithm that can modify the TS array: Lines 13, 33 and 73. By Observation 30 a successful SC() in line 33 cannot change the cluster or index component of any active timestamp. Furthermore, by Observation 49, a successful SC() in line 73, writes a negative number to TS[j].index. Therefore, at point t_q, process q must perform the SC() in line 13.

Let $t_{LL} < t_q$ be the point when process q performs the corresponding LL() on TS[j] in line 9, and let $t_r \in (t_{LL}, t_q)$ be the point when q performs the FAI() operation in line 12. Thus, $cl_{t_r} = \lfloor \frac{C_{t_r}}{\delta} \rfloor = k$ and $idx_{t_r} = C_{t_r} \bmod \delta > i$, since at point t_q process q writes $\lfloor \frac{C_{t_r}}{\delta} \rfloor = cl_{t_r} = k$ and $C_{t_r} \bmod \delta = idx_{t_r}$ to TS[j].cluster and TS[j].index, respectively. Therefore, t_r is in a cluster-k interval. Furthermore, if t_r is not in the same maximal cluster-k interval as t_q, then during $[t_{LL}, t_q]$ the active cluster changes at least two times. This contradicts the assumption that at point t_q, process q's SC() is successful, because by Lemma 5, at least two successful TS[j].SC() operations happen during $[t_{LL}, t_q]$. Therefore, t_r must be in the same maximal cluster-k interval as t_q. Furthermore, $t_r > t_f$, since $idx_{t_r} > idx_{t_f} = i$. This contradicts the assumption that $j \prec i$ at point t_p, because t_r and t_f are the linearization points of j's, and i's last updateTS() calls, respectively, which are linearized before t_p.

Case 2: Suppose TS[i].cluster = $(k-1) \bmod 3$, and TS[j].cluster = k immediately after t_p. Then, $t_f < t$, since $r = (k-1) \bmod 3$. Therefore, by Lemma 24, $idx_{t_p} < \eta$. Let $t_q < t_p$ be the last point when some process q performs a successful SC() on TS[j] that changes TS[j].cluster, such that TS[j].cluster = k immediately after t_q. Then, $t_q \in [t, t_p]$, since by Lemma 5, TS[j].cluster $\neq k$ immediately before t.

There are three shared memory operations in the algorithm that can modify the TS array: Lines 13, 33 and 73. By Observation 30 a successful SC() in line 33 cannot change the cluster component of TS[j]. Furthermore, since $idx_{t_p} < \eta$, by Lemma 35 no successful SC() in line 73, happens during $[t, t_p]$. Hence, at point $t_q \in [t, t_p]$, process q performs the SC() in line 13.

Let $t_{LL} < t_q$ be the point when process q performs the corresponding LL() on TS[j] in line 9, and let $t_r \in (t_{LL}, t_q)$ be the point when q performs the FAI() operation in line 12. With the same reasoning as case 1, t_r happens during the same active cluster-k as t_p. Therefore, $t_r > t > t_f$. This contradicts the assumption that $j \prec i$ at point t_p, because t_r and t_f are the linearization points of j's, and i's last updateTS() calls, respectively, which are linearized before t_p. □

The following lemma extends Lemma 53.

Lemma 54. *Let i and j be any two processes. Suppose at point t, a successful* SC() *on* TS[i] *is performed in line 13 and $j \prec i$. Furthermore, suppose at point t, process j has no* updateTS() *call which is linearized before t and published after t. Let t' be the first point after t when $i \prec j$. Then* TS[j] $\ll$ TS[i] *throughout* (t, t').

Proof. By Lemma 53, TS[j] $\ll$ TS[i] immediately after the successful TS[i].SC() operation at point t. By Lemma 52, only a successful SC() operation executed in line 13, on either TS[i], or TS[j] can change the dominance relation order TS[i] and TS[j] Furthermore, since the first point after t when $i \prec j$ is t', process j has no updateTS() call which linearizes during (t, t'). Thus, during (t, t') no successful SC() on TS[j] is executed in line 13, since the first point after t when such SC() happens is after the linearization point of an updateTS() call by j. Lastly, by Lemma 53, during (t, t'), after every successful TS[i].SC() operation executed in line 13, TS[j] $\ll$ TS[i]. Therefore, TS[j] $\ll$ TS[i], throughout (t, t'). □

Lemmas 53 and 54 and the fact that initially the dominance relation between all active timestamps is consistent with the interpreted value of the MTS object, immediately yields Lemma 9.

Lemma 9. *Let i and j be two distinct processes, t a point in time, and t^* the latest point before t, such that the publishing point of any* updateTS() *call by i or j with linearization point before t^* occurs before t. Then the dominance order of* TS[i] *and* TS[j] *at point t is consistent with the interpreted value of the MTS object at point t^**

In the following lemma we show that the last updateTS() call by process i or j, that is linearized before some process p invokes a isEarlier(i, j) call, is also published before p executes lines 39.

Lemma 55. *Suppose at point t, process p starts a* isEarlier(i, j) *call. Let $t' > t$ be the point when p is poised to execute line 39. Furthermore, suppose at some point $t_{SC} > t'$, process q performs a successful* SC() *on* TS[i], *or* TS[j] *in line 13. Let $t_{fetch} < t_{SC}$ be the point when process q performs the preceding* FAI() *operation in line 12. Then $t_{fetch} > t$.*

Proof. Here we consider the case that at point t_{SC} process q performs a TS[i].SC() operation in line 13. The proof for the case that q performs a TS[j].SC() operation is identical.

Suppose at point t_{SC}, process q performs a successful TS[i].SC() operation, and at point $t_{fetch} < t$ process q performs the preceding FAI() operation in line 12. Let $t_{LL} < t_{fetch}$ be the point when process q performs the preceding LL() in line 9. Therefore, TS[i] does not change during $[t_{LL}, t_{SC}]$. Furthermore, let $t_r \in [t_{LL}, t_{fetch}]$ be when q reads the announce bit of i in line 10. Since q does not execute the return statement in line 11, TS[i].flag $\neq$ A[i] at point t_r. Therefore, by Lemma 17, the next component of these two that changes is TS[i].flag. Therefore, TS[i].flag $\neq$ A[i] throughout $[t_r, t_{SC}]$, since this change happens at point t_{SC}. Thus, during some interval $[t_1, t_2] \subset [t, t']$ process p by executing lines 35-38 performs a complete helpUpdateTS(i). Hence, by Lemma 7a there exists

some point $t^* \in [t_1, t_2]$ when TS[i] changes. This contradicts the assumption that q's SC() at point t_{SC} is successful, since $t^*[t_1, t_2] \subsetneq (t_{LL}, t_{SC})$. □

In the following observation we show that the response of an isEarlier(i, j) call that returns in either line 49, or 54, is consistent with the dominance order of TS[i] and j at some point during the execution of isEarlier(i, j) call.

Observation 56. *Suppose p starts and finishes an* isEarlier(i, j) *call at points t_s and t_f, respectively. Let $t' \in (t_s, t_f)$ be the point p is poised to execute line 39 during the same* isEarlier() *call. If at point t_f process p executes the return statement in line 49 or line 54, then there exists a point $t^* \in [t', t_f]$ when the dominance relation between* TS[i] *and* TS[j] *is the same as the response of p's* isEarlier(i, j) *call.*

Proof. The proof for the case when p executes the return statement in line 49 is similar to the case when p executes the return statement in line 54. Below we write the proof for first case and note the differences in parentheses.

Suppose at point t_f, process p executes the return statement in line 49 (line 54). Let $t_1 \in [t', t_f]$ be the point when p performs an LL() on TS[i] (TS[j]) in line 44 (line 45). Furthermore, let $t_2 > t_1$ be the point when p subsequently performs the VL() in line 46 (line 51). Therefore, TS[i] (TS[j]) does not change during $[t_1, t_2]$, because p's VL() operations returns true. Let $t^* \in [t_1, t_2]$ be the point when process p performs an LL() on TS[j] (TS[i]) in line 45 (line 50). Thus, p's local view of TS[i] and TS[j] is consistent with the values stored in the TS array at point t^*. Lastly, since p's return is based on the dominance relation between its local view of TS[i] and TS[j], its return is consistent with the dominance relation between TS[i] and TS[j] at point $t^* \in [t_1, t_2] \subsetneq [t, t']$. □

The following lemma states that when an isEarlier() call by process p responds, LookupTable[p] is not $\perp$.

Lemma 57. *Let t be a point at which an* isEarlier(i,j) *call by process p responds. Then* LookupTable[p].res $\neq \perp$ *at point t.*

Proof. Suppose the lemma is not true. Let t be the first point during the execution when process p's isEarlier(i, j) call responds, and LookupTable[p].res $= \perp$ at this point. Since initially LookupTable[p].res $\neq \perp$, there exists at least one point before t when a successful SC() sets it to $\perp$. Let $t^* < t$ be the last point before t, when some process performs a successful SC() on LookupTable[p] that sets LookupTable[p].res $= \perp$. There are five shared memory operations in the algorithm that can modify the LookupTable array: Lines 19, 41, 48, 53 and 55. However, a successful SC() in line 55 cannot set LookupTable[p].res to $\perp$, since it sets it to the constant value True. Neither a successful SC() in lines 48 and 53 can set LookupTable[p].res $= \perp$, since they use the result of a $\ll$ operation which is a Boolean value. Therefore, at point t^*, either process p executes a successful SC() in line 41, or some process q executes a successful SC() on LookupTable[p] in line 19.

First, suppose at point t^*, process p executes a successful SC() in line 41 during the same or a preceding isEarlier() call. Let $t^{**} \in (t^*, t]$ be the point when p completes that isEarlier() call. There are three return statements in the isEarlier() call that p can execute: lines 49, 54 and 56. Before executing the return statement in line 49, 54, or 56, process p executes an SC() on LookupTable[p] in line 48, 53, or 55, respectively, These SC() operations cannot set LookupTable[p].res to $\perp$. Thus, either t^* is not the last point before t when some process performs a successful SC() on LookupTable[p] that sets LookupTable[p].res $= \perp$, or at LookupTable[p].res $\neq \perp$ at point t. — a contradiction.

Now suppose at point $t^* < t$ some process q executes a successful SC() on LookupTable[p] in line 19 and sets LookupTable[p].res to $\perp$. Since q in line 19 write the results of its preceding

isEarlier(*i,j*) call in line 18 to LookupTable[p].res, at some point $t^{**} < t^*$ its isEarlier(*i,j*) returns $\perp$. There are three return statements in the isEarlier() function: Lines 49, 54 and 56. Since return statements in lines 49 and 54 return the result of a $\ll$ operation, an isEarlier() call that executes either of these return statements does not return $\perp$. Hence, at point t^{**}, process q executes the return statements in line 56 during its isEarlier(*i,j*) call. Since by executing line 56, process q performs an LL() on LookupTable[q] and return its res component, LookupTable[q].res = $\perp$ at point $t^{**} < t$. This contradicts the assumption that t is the first point in time when the lemma is not true. □

In the following observation, we show that the return value of an isEarlier() is never $\perp$.

Observation 58. *The return value of an* isEarlier() *call cannot be* $\perp$.

Proof. There are three return statements in the isEarlier() function: Lines 49, 54 and 56. Since the return statements in lines 49 and 54 use the result of a $\ll$ operation, an isEarlier() call that executes either of those return statements returns a Boolean value. Furthermore, any process p that executes the return statement in line 56 during an isEarlier() call returns the value of LookupTable[p].res at the point of its response. By Lemma 57 at that point LookupTable[p].res $\neq \perp$. Thus, the result of any isEarlier() call cannot be $\perp$. □

The following observation states that if at some point t, the value of LookupTable[p].res changes to $\perp$, then at point t, process p executes the SC() in line 41.

Observation 59. *Let t be a point in time when a shared memory operation is executed. If* LookupTable[p].res $\neq \perp$ *immediately before t and* LookupTable[p].res $= \perp$ *immediately after t, then at point t process p executes a successful* SC() *in line 41.*

Proof. There are five shared memory operations in the algorithm that can modify LookupTable[p]: Lines 19, 41, 48, 53 and 55. A successful SC() in line 55, sets LookupTable[p].res = True. Also, a successful SC() in line 53 or 48 sets LookupTable[p].res to the result of a $\ll$ operation which is a Boolean value. Lastly, a successful SC() in line 18 sets LookupTable[p].res to the result of a preceding isEarlier() call, which by Observation 58 cannot be $\perp$. Hence, only a successful SC() in line 41 performed by process p can set LookupTable[p].res to $\perp$. □

The following lemma shows the SC() in line 41 is guaranteed to be successful.

Lemma 60. *If process p perform an* SC() *in line 41, then its* SC() *is successful.*

Proof. Suppose at point t, process p performs an SC() in line 41 during an isEarlier(*i,j*) call, and p's SC() not successful. Let $t_{LL} < t$ be the point when p performs the preceding LL() on LookupTable[p] in line 40. Since p's SC() at point t is not successful, LookupTable[p] must change at least once during (t_{LL}, t) by some other process. Let $t^* \in (t_{LL}, t)$ be the first point after t_{LL} when some other process q performs a successful SC() on LookupTable[p]. Since the only line in the algorithm where q might perform an SC() on LookupTable[p] is line 19, at point t^* process q executes line 19. Let $t_q < t*$ be the point when q performs the preceding LL() on LookupTable[p] in line 16. Since before executing line 19 process q checks the condition in line 17, LookupTable[p].res = $\perp$ at point t_{LL}. Consider the last point $t_s < t_{LL}$ when LookupTable[p].res changed. By Observation 59 at that point p executes line 41 during a isEarlier() call. Since p's pending isEarlier() call at point t_s must finish before p can invoke another isEarlier() call. There exists a point $t_f \in (t_s, t_{LL})$ when p's pending isEarlier() call at point t_s responds. By Lemma 57, LookupTable[p].res $\neq \perp$ at point t_f. Thus, there exists a point $t^{**} \in (t_s, t_f)$ when LookupTable[p].res changes. Since we defined t_s to be the last time before t_q when LookupTable[p].res changes, $t_q < t^{**} < t_f < t_{LL} < t^*$. This contradicts the assumption that q's SC() on LookupTable[p] at point t^* is successful. □

The following lemma shows under what condition LookupTable[p] gets updated by another process helping p complete a `isEarlier(`i, j`)` call.

Lemma 61. *Suppose process p starts and finished an* `isEarlier(`i,j`)` *call at point t_s and t_f, respectively. Furthermore, suppose at point t_f process p executes the return statement in line 56. Let $t_a \in [t_s, t_f]$ be the point when p executes line 41. If no successful* `SC()` *on either* TS[i] *or* TS[j] *(or both) is executed in line 13 during $[t_a, t_f]$, then there exist a point $t^* \in (t_a, t_f)$ when some other process q performs a successful* `SC()` *on* LookupTable[p] *in line 19.*

Proof. For the purpose of contradiction suppose the lemma is not true, and at no point during (t_a, t_f) some other process q performs a successful `SC()` on LookupTable[p] in line 19. Let $t' < t_f$ be the point when p perform the `SC()` in line 55. Since during $(t_a, t') \subsetneq [t_s, t_f]$, process p does not execute either of the return statements in line 49 or line 54, both TS[i] and TS[j] must change at least six times (once in each iteration of the loop comprising lines 43-54). Furthermore, since none of these change is because of a successful `SC()` in line 13, by Lemma 42 during (t, t') the value of Counter is incremented at least $3n^3 + 6n^2 + 6n$ times. Thus, by Lemma 7(c) during (t, t') there exist a process q such that during (t, t'), process q performs n complete `helpSystem()` calls. With every `helpSystem()` call, by executing line 20 process q updates the helpID variable to another process ID in a round-robin fashion. Thus, after n complete `helpSystem()` calls, there exist an interval $[t_1, t_2] \subsetneq (t, t')$ where q executes lines 16-19 with helpID $= p$. Let $t_{LL} \in [t_1, t_2]$ be the point when q performs an `LL()` on LookupTable[p] in line 16. By Lemma 60, LookupTable[p].res $= \perp$ immediately after t. Since during (t, t') process p does not execute either of the `SC()` operations in line 48 or 53, process p does not change LookupTable[p] during (t, t'). Furthermore, the only other line in the algorithm where another process might perform an `SC()` on LookupTable[p] is in line 19. Thus, LookupTable[p] does not change during (t, t'). Therefore, at point $t_{LL} \in (t, t')$ process q reads LookupTable[p].res $= \perp$. Hence, the condition in line 17 is met. Thus, there exists a point $t^* \in [t_1, t_2]$ when process q executes the `SC()` in line 19. Lastly, since no successful `SC()` is executed on LookupTable[p] during $[t_{LL}, t^*] \subsetneq (t, t')$, process q's `SC()` on LookupTable[p] at point t^* is successful. This contradiction the assumption that at no point $t^* \in (t, t_f)$ some other process q performs a successful `SC()` on LookupTable[p] in line 19. □

The following lemma shows that if LookupTable[p] gets updated by some other process q, helping p complete an `isEarlier()` call, then q sets LookupTable[p].res to the result of an `isEarlier()` call executed during the interval in which, p is executing its `isEarlier()` call.

Lemma 62. *Suppose process p starts and finishes an* `isEarlier(`i, j`)` *call at points t_s and t_f, respectively. Furthermore, suppose at point $t_q \in (t_s, t_f)$, some process q, performs a successful* `SC()` *on* LookupTable[p] *in line 19. Let $t_{qLL} < t_q$ be the point when q performs the preceding* `LL()` *on* LookupTable[p] *in line 16. Then $t_{qLL} \in [t_s, t_q)$.*

Proof. Suppose p starts and finishes an `isEarlier(`i, j`)` call at points t_s and t_f, respectively, and at point $t_q \in (t_s, t_f)$ process q performs a successful `SC()` on LookupTable[p] in line 19. Let $t_{qLL} < t_q$ be the point when q performs the preceding `LL()` on LookupTable[p] in line 16. For the purpose of contradiction suppose $t_{qLL} < t_s$. Therefore, LookupTable[p].res $= \perp$ at point t_{qLL}, since before executing line 19 process q checks the condition in line 17. Let t_1 be the last point before t_{qLL} when LookupTable[p].res changes such that LookupTable[p].res $= \perp$ immediately after t_1. By Observation 59 at point t_1 process p executes line 41 during a preceding `isEarlier()` call. Let $t'_f < t_s$ be the point when that `isEarlier()` call finishes. By Lemma 57, LookupTable[p].res $\neq \perp$ at point t'_f. Thus, there exist a point $t^* \in (t_1, t'_f)$ when a successful `SC()` on LookupTable[p] changes LookupTable[p].res. Furthermore, $t_{qLL} \in (t_1, t^*)$, since t_1 is the last time before t_{qLL} when

a successful SC() changes LookupTable[p].res to $\perp$. This contradicts the assumption that q's SC() at point t_q is successful, since $t_{qLL} < t^* < t_q$. □

Lemma 8. *Consider an* isEarlier(i, j) *call that gets invoked and responds at points t and t', respectively. The response of the* isEarlier() *call is consistent with the interpreted value of the MTS object at some point $t^* \in [t, t']$.*

Proof. Suppose the lemma is not true, and let t' be the first point during the executing at which a isEarlier(i, j) call of some process p responds such that at no point during its execution, its response is consistent with the interpreted value. Let $t < t'$ be the point when p invokes this isEarlier(i, j) call. Furthermore, let $t_1 \in (t, t')$ be the point when p is poised to execute line 39 during the same isEarlier(i, j) call. First observe that by Lemma 55:

$$\text{If } i \text{ or } j \text{ has an updateTS() call that is linearized before } t, \text{ then the publishing point of that call is before } t_1. \tag{$*$}$$

There are three return statements in the isEarlier() function: lines 49, 54 and 56. Thus, at point t' process p must execute one of these three statements.

Case 1: suppose at point t' process p executes the return statement in line 49, or 54. By Observation 56 there exist a point $t^* \in [t_1, t']$ when p's responds is consistent with the dominance relation between i and j's active timestamp. Furthermore, by ($*$) there exists a point $t_x \in [t, t_1]$ when i has no pending updateTS() call that is linearized but not published. Similarly, there exists a point $t_y \in [t, t_1]$ when j has no pending updateTS() call that is linearized but not published. Without loss of generality assume $t_x < t_y$. Thus, there exists a point between $t_{xy} in [t_x, t_y]$ when neither i nor j has a pending updateTS() call that is linearized but not published. Therefore, by Lemma 9 the dominance relation between TS[i] and TS[j] at point t^* is consistent with the interpreted value of the MTS object at some point $t^{**} in [t_{xy}, t^*]$. This contradicts the assumption that at no point during p's isEarlier(i, j) call p's response is consistent with the interpreted value, since $t^{**} \in [t_{i,j}, t^*] \subsetneq [t, t']$.

Case 2: suppose at point t' process p executes the return statement in line 54. Let t_a be the point when p executes the SC() in line 41. Furthermore, let $t_{SC} \in (t_a, t')$ be the last point when some process executes a successful SC() on LookupTable[p]. Since the last time before t' when p might perform a successful SC() on LookupTable[p] is when p executes the SC() operation in line 55, at point t_{SC} either p executes the SC() in line 55 or some other process q execute a successful SC() on LookupTable[p].

Case 2a: suppose at point t_{SC} some process q performs a successful SC() on LookupTable[p]. The only line in the algorithm that q might perform an SC() on LookupTable[p] is line 19. Thus, at point $t_{SC} \in (t_a, t')$ process q executes a successful SC() in line 19. Let t_{qLL} be the point when q executes the preceding LL() operation on LookupTable[p] in line 16. By Lemma 62, $t_{qLL} \in [t, t_{SC})$. At point t_{SC} process q sets LookupTable[p].res to the response of its preceding isEarlier(i, j) call in line 18. By the assumption that p's isEarlier(i, j) call is the first isEarlier() call for which the lemma is not true, there exist a point $t^* \in (t_qLL, t_{SC})$ such that at t^* the response of q's isEarlier(i, j) call is consistent with the interpreted value of the MTS object. Since t_{SC} is the last time before t' when a successful SC() on LookupTable[p] is executed, at point t' process p returns the same response as q's isEarlier(i, j) call. This contradicts the assumption that at no point during $[t, t']$ process p's response is consistent with the object's interpreted value, since $t^* \in (t_qLL, t_{SC}) \subsetneq [t, t']$.

Case 2b: suppose at point t_{SC} process p's SC() in line 55 is successful. Thus, LookupTable[p].res = True immediately after t_{SC}. Since t_{SC} is the last point when a successful SC() on LookupTable[p] is executed before t', at point t' process p returns True. Let t_2 be the point when p executes the

LL() operation in line 42 during the same isEarlier() call. Since p does not execute either of the return statements in line 49, or 54, during $[t_2, t']$, both TS$[i]$ and TS$[j]$ each change at least six times (once in each iteration of the loop in line 43). Furthermore, during the first three iteration of the loop, by Lemma 61, if no successful SC() on either TS$[i]$ or TS$[j]$ is executed in line 13 during $[t, t_f]$, then there exist a point during (t_2, t_{SC}) when some other process q performs a successful SC() on LookupTable$[p]$ in line 19. Therefore, during the first three iteration of the loop, both TS$[i]$ and TS$[j]$ each have at least one successful SC() in line 13. Otherwise, p's SC() at point t_{SC} is not successful. Let t_{x_1} and t_{y_1} be the first points during the first three iteration of the loop when some process in line 13 performs a successful SC() on TS$[i]$ and TS$[j]$, respectively. Similarly, in the second three iteration of the loop both TS$[i]$ and TS$[j]$ each have at least one successful SC() in line 13. Let t_{x_2} and t_{y_2} be the first points during the second three iteration of the loop when some process in line 13 performs a successful SC() on TS$[i]$ and TS$[j]$, respectively. Thus, t_{x_1} and t_{y_1} are both before t_{x_2} and t_{y_2}. A successful SC() in line 13 is the publishing point of a linearized updateTS() call. By $(*)$ the linearization point of these updateTS() operations must be after t. Let $t^*_{x_1} \in (t, t_{x_1})$ and $t^*_{y_1} \in (t, t_{y_1})$ be the linearization point the updateTS() calls that are published at points t_{x_1} and t_{y_1}, respectively. Similarly, let $t^*_{x_2} \in (t_{x_1}, t_{x_2})$ and $t^*_{y_2} \in (t_{y_1}, t_{y_2})$ be the linearization point the updateTS() calls that are published at points t_{x_2} and t_{y_2}, respectively. If $t^*_{x_1} < t^*_{y_1}$, then $i \prec j$ at point $t^*_{x_1}$. Otherwise, $i \prec j$ at point $t^*_{y_2}$. This contradicts the assumption that at no point during p's execution its response is consistent with the interpreted value, since both $t^*_{y_2}$ and $t^*_{x_1}$ are during the interval (t, t'). □

Lemma 8 implies that our MTS object is linearizable, and by Lemma 12 its step complexity is constant. Finally, according to Section 4.2, our algorithm uses one mod φ-FAI object, where $\varphi = O(n^3)$, and $O(n)$ LL/SC objects and registers of size $O(\log n)$, each. This immediately yields Corollary 10, which in turn yields Theorem 1 (see Section 5).

B BOUNDED FAI OBJECT

Lemma 63. *Let t be a point in time when a shared memory operation is executed, and let v be the value of* Counter *at point t. Furthermore, let k^- and k^+ be the number of processes that immediately before t are poised to execute the* FAA *operations in line 76 and line 77, respectively. Then the algorithm holds the following invariant.*

$$v \in [(\varphi - 1)k^-,\ \varphi \times n - k^+]$$

Proof. Since at the beginning no process is poised to execute the FAA operation in line 76 or line 77, both k^+ and k^- are 0. Hence, the invariant is true initially, because Counter is initialized with value 0 and $0 \in [0, varphi \times n]$. There are three possible shared memory operations that can effect at least one of the terms in the invariant: the read operation in line 75, the FAA operation in line 76, or the FAA operation in line 77. Below, for each case, we prove that if the invariant holds before this operation, then the invariant still holds after the operation. Therefore, since the invariant is true initially, and it remains true after any of the shared memory operations, the invariant holds true at any point during an execution.

Case 1: Suppose at point t a process, p performs the FAA operation in line 77. Then at point t the value of v increases by 1. Also, k^+ decreases by 1, since immediately before t process p was poised to execute the FAA operation in line 77. Hence, the upper bound of the invariant $(\varphi \times n - k^+)$ increases by 1 as well. Thus, immediately after t the invariant still holds, because both v and the upper bound of the invariant increase by 1, and its lower bound does not change.

Case 2: Suppose at point t a process, p performs the FAA operation in line 76. Then at point t the value of v decreases by $\varphi - 1$. Also, k^- decreases by 1, since immediately before t process p was poised to execute the FAA operation in line 76. Hence, the lower bound of the invariant $((\varphi - 1)k^-)$

decreases by $\varphi - 1$. Thus, immediately after t the invariant still holds, because both v and the lower bound of the invariant decrease by $\varphi - 1$, and its upper bound does not change.

Case 3a: Suppose at point t a process, p performs the `read` operation in line 75 and becomes poised to execute the FAA operation in line 77. Hence, k^+ increases by 1 and $v < \varphi \times n - n$ at point t. Since k^+ is at most n, at any point during an execution the upper bound of the invariant is at least $\varphi \times n - n$. Therefore, immediately after t the invariant still holds, because the upper bound of the invariant is at least $\varphi \times n - n$, and neither v nor the lower bound of the invariant change.

Case 3b: Suppose at point t a process, p performs the `read` operation in line 75 and becomes poised to execute the FAA operation in line 76. Hence, k^- increases by 1 and $v \geq \varphi \times n - n$ at point t. Since k^- is at most n, at any point during an execution the lower bound of the invariant is at most $\varphi \times n - n$. Therefore, immediately after t the invariant still holds, because the lower bound of the invariant is at most $\varphi \times n - n$, and neither v nor the upper bound of the invariant change. □

The following is a direct result of the Lemma 63.

Corollary 64. *Suppose a process executes* FAA(x) *at a point when* Counter *has value* v*. Then* $v + x \in \{0, \ldots, B - 1\}$