Locking Timestamps Versus Locking Objects

Marcos K. Aguilera[1], Tudor David[2], and Rachid Guerraoui[3]

1 VMware Research
2 EPFL
3 EPFL

Abstract

We present *multiversion timestamp locking (MVTL)*, a new genre of multiversion concurrency control algorithms for serializable transactions. The key idea behind MVTL is simple and novel: lock individual time points instead of locking objects or versions. After showing what a generic MVTL algorithm looks like, we demonstrate MVTL's expressiveness: we present several simple MVTL algorithms that address limitations of current multiversion schemes, by committing transactions that previous schemes would abort, by avoiding the problem of serial aborts and ghost aborts, and by offering a way to prioritize transactions that should not be aborted.

1 Introduction

The serializable transaction abstraction is a powerful paradigm available in many computing systems, such as transactional memory, database systems, and key-value stores. To ensure serializability, transactions require a scheme for concurrency control to handle any negative consequences of transaction interleaving.

The literature on concurrency control is rich [4, 26], and a particularly appealing class of algorithms is called *multiversion concurrency control* [3]. Briefly, these algorithms keep a *history* of each object with timestamped versions. This history gives the system a *choice* of which version to use when an object is accessed. This choice permits more transactions to execute concurrently without blocking or aborting. For example, in some multiversion algorithms [4, 7], read-only transactions can execute without ever blocking or aborting, and update transactions can concurrently update the same object. Enabling more concurrency has become particularly important with the proliferation of multi-core and large-scale systems. Multiversion algorithms have wide application: they are used often in database systems both commercial and academic [9, 19, 26, 27], and more recent work has applied them to key-value stores and transactional memory (e.g., [10, 15–17, 22, 23]). In this paper, we do not restrict ourselves to any particular application, but rather study multiversion algorithms in their broadest and most conceptual scope.

There are three main genres of multiversion algorithms: *lock based*, *timestamp ordering*, and *serialization graph based* [4]. Lock-based algorithms (e.g., MV2PL [4]) acquire locks to avoid the ill-effects of concurrency; these algorithms are very simple. Timestamp ordering algorithms (e.g., MVTO [4]) assign a timestamp to each transaction, and then serialize transactions by timestamp; these algorithms permit read-only transactions to execute without ever aborting. Serialization graph algorithms (e.g., MVSGT [26]) detect cycles in the serialization graph to prevent a violation of serializability; these algorithms permit higher levels of concurrency than the alternatives.

Despite their many benefits, all types of multiversion algorithms have limitations. Lock-based algorithms significantly limit the degree of concurrency. Timestamp ordering algorithms are susceptible to aborts, including *serial aborts*—aborts in serial executions—and *ghost aborts*—aborts caused by a conflict with a transaction that already aborted. Serialization graph algorithms are complex and incur significant computation overheads [1, 16, 20].

In this paper, we introduce a new genre of multiversion algorithms, called *multiversion timestamp locking*, or MVTL. MVTL is based on a simple novel idea: use locks as in lock-based algorithms, but lock individual timestamps of objects, rather than entire objects at a time. A transaction is allowed to commit if it can find at least one timestamp that it managed to lock across all its objects. Intuitively, MVTL excels because it uses locks with very fine granularity: not only individual objects have separate locks, but individual timestamps within objects have their own locks. Locking at fine granularity increases parallelism and decreases blocking and aborting, as the system can explore many serialization points for each transaction.

Conceptually, MVTL keeps a lock state for each object and each timestamp, which amounts to an infinitely large lock state. However, pragmatically we can reduce the lock state significantly using interval compression, so that each object holds just a few lock intervals, and this state can be subsequently discarded when the associated versions are purged.

To precisely define MVTL, we give a generic algorithm (Section 4) that has several nondeterministic choices, such as what timestamps each operation tries to lock, and how locks are acquired (wait or give up on blocked locks). We prove that the generic algorithm is

correct irrespective of those choices: they do not affect safety. They may however be crucial for performance.

We then propose several specific algorithms that specialize the generic MVTL algorithm by fixing these nondeterministic choices to obtain different benefits (Section 5). These algorithms are simple and address some important drawbacks of existing multiversion algorithms, such as serial aborts, ghost aborts, the lack of a priority scheme for transactions, and more. We also show that pessimistic and timestamp ordering algorithms can be seen as special cases of MVTL. Thus, in a precise sense, MVTL unifies these algorithms.

Next, we discuss some pragmatic considerations around MVTL, such as how to compress the lock state (Section 6). We separate out these considerations because they are orthogonal to the concepts underlying the MVTL algorithm. However, they are important to applying MVTL in practice.

Due to space limitations, in the main body of the paper we focus on a shared-memory version of MVTL. However, we believe MVTL is particularly relevant in distributed message-passing systems, where MVTL can achieve a high degree of communication efficiency. We explore this setting in the optional appendix, where we show how to extend MVTL to a distributed setting.

To summarize, the contributions of this paper are as follows:

- We propose a new genre of multiversion algorithms for transactions, called multiversion timestamp locking (MVTL), which is based on the idea of locking timestamps.
- We give several MVTL algorithms, which address various limitations of current multiversion algorithms.
- We show that MVTL generalizes both multiversion timestamp ordering and pessimistic multiversion algorithms.
- We discuss practical considerations for implementing MVTL, including techniques to significantly reduce the space of lock state.
- We explain how MVTL can be applied to a distributed system setting.

Our main contribution is conceptual in nature. We believe that locking individual timestamps introduces a new way to approach multi-version algorithms. The specific MVTL algorithms we present are simple and just scratch the surface; we think the investigation of additional MVTL algorithms is an exciting direction for future work. Also interesting is to carry out an experimental study to measure the performance of MVTL algorithms in a transactional system, such as software transactional memory, transactional key-value storage systems, transaction object systems, or database systems. While the fundamental MVTL algorithms we present are independent of the type of transactional system, the details of how these algorithms are implemented are system specific and deserve further study.

Due to space limitations, some details of our contribution are given in the appendix, including proofs, pseudo-code of some algorithms, and the distributed version of MVTL.

2 Model

We consider a standard model for a multi-threaded concurrent system [14]. The system has processes that communicate via atomic shared memory. The system is asynchronous: there are no bounds on the relative speed of processes. We assume the existence of a discrete global clock with domain $\mathcal{T} = \{0, 1, \ldots\}$, and processes may or may not have access to the global clock. More precisely, processes may have local clocks that match the global clock ("synchronized clocks") or that are within a known bound ϵ of the global clock ("ϵ-synchronized clocks").

We are interested in algorithms that implement a transactional storage system. Such a system maintains a set of objects and allows processes to manipulate the objects using transactions. Each object has a unique key (identifier) and, by abuse of language, we refer to the object and its key interchangeably. The system supports four operations with their usual semantics: BEGIN(tx) starts a transaction tx, COMMIT(tx) tries to commit tx and returns a success indication, READ(tx, k) reads key k within tx, and WRITE(tx, k, v) writes v to k within tx. Transactions are dynamic: their read and write operations can depend on the results of prior operations issued within the transaction. Algorithm 1 shows an example.

Algorithm 1 Example of a transaction

```
1: BEGIN(tx)
2: v ← READ(tx, A)
3: if v ≤ 100 then WRITE(tx, B, 50)
4: result ← COMMIT(tx)
```

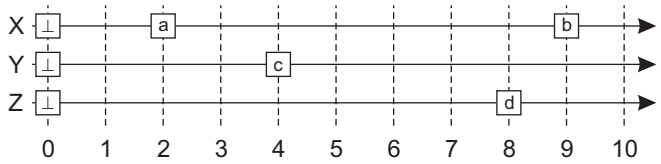
The correctness condition for transactional storage we consider is *multiversion view serializability*, a form of serializability well-suited for multiversion algorithms. Roughly speaking, this condition requires every multiversion schedule of the algorithm to be equivalent to a serial monoversion schedule [4, 26].

Some of our results refer to a *workload*, which is a sequence of operations (indexed by the transaction they belong to) that are input to the system, where each operation is *read*(k), *write*(k, v), or *tryCommit*. The purpose of this definition is to observe how different protocols react under the same operations, to understand which one aborts or delays transactions more.

3 Overview

In this section, we first recall multiversion concurrency control algorithms, after which we introduce our new concept of *timestamp locking* and explain how it enables us to address the weaknesses of existing multiversion algorithms.

Multiversion concurrency control and the MVTO+ algorithm. The basic idea of multiversion timestamp ordering is to assign a timestamp to each transaction and then use the timestamp to determine (a) what version the transaction reads from, (b) what version it writes to, and (c) the serialization order of transactions. This idea can lead to several slightly different algorithms. To focus the discussion, here we present a concrete algorithm denoted MVTO+, which is identical to the MVTO algorithm in [4] but with an improvement: it avoids cascading aborts by not reading uncommitted data. For each object, MVTO+ keeps many versions and a timestamp for each version. It is useful to think of each object as an evolving timeline with values. Each transaction tx has a unique timestamp t, which determines the version of objects that tx reads and writes. Specifically, when tx reads an object, it obtains the version of the object with the largest timestamp before t. When tx writes an object, tx does not immediately produce a new version but instead it stores the written value in a temporary area for the transaction. Upon commit, tx takes each written value in this temporary area and produces a new version with timestamp t.



For example, the figure above depicts three objects X, Y, and Z. Each object has an initial version denoted $\perp$. In addition, X has two other versions with data a and b and timestamps 2 and 9 respectively; Y has data c with timestamp 4; and Z has data d with timestamp 8. Now suppose a transaction tx is assigned a timestamp 6. If tx reads X, it obtains a—the largest version with a timestamp before 6. Similarly, if tx reads Y, it obtains c. If tx writes e to Z and commits, then Z gets a new version with data e and timestamp 6.

Ultimately, transactions are serialized by the order of their timestamps. A key implication is that, after tx reads X and obtains a, another transaction should not produce a version of X with a timestamp between 2 and 6. To prevent this behavior, MVTO+ keeps a *read-timestamp* for each version: this is the largest timestamp with which the version was read by a transaction. In the example, after tx reads X and obtains a, the read-timestamp of a becomes 6 (if it was not already larger than 6).

Timestamp locking. We look at MVTO+ slightly differently, using our new notion of *timestamp locking*. This notion allows us to generalize MVTO+ into our new MVTL algorithm. Rather than read-timestamps, we can think that each object has several locks, one for each timestamp. When tx reads X, rather than updating the read-timestamp of a to 6, we can think that tx obtains a read-lock on each timestamp between 3 and 6. When another transaction wishes to write a version with timestamp, say 5, it must obtain the write-lock on that timestamp. But the read-locks by tx prevent this from happening, as required by MVTO+. We can now see the read-timestamp of a as simply a compact representation of the fact that there are read-locks between 3 and 6.

Thinking about timestamp locks has several advantages over read-timestamps. First, with read-timestamps, it is not clear what should happen if tx aborts: should the read-timestamp of a be updated to its previous value? But what is the previous value if several other transactions read a concurrently? This is a hard question, and MVTO+ avoids it altogether by taking an unnecessarily conservative approach: when tx aborts, it leaves the read-timestamp of a at 6. We show that this choice leads to ghost aborts. In contrast, timestamp locks provide a better alternative: if tx aborts, its read-locks are removed but the read-locks of other transactions remain.

Second, with timestamp locks, there is no reason that a transaction should be restricted to obtaining write-locks on just one timestamp, or obtaining read-locks on a range that ends with the transaction's timestamp. Permitting more choices allows the system to avoid serial aborts, as we explain later.

These advantages are captured by our MVTL algorithm, which we now briefly summarize. With MVTL, when a transaction wishes to read an object, it selects a version of the object to read and obtains read-locks on one or more timestamps adjacent to and immediately following that version. To write an object, the transaction obtains write-locks on one or more timestamps anywhere. To commit, the transaction must find a single common timestamp that is read-locked or write-locked across all objects read or written by the transaction, respectively. If such a timestamp exists, the transaction commits; otherwise, it aborts.

The exact timestamps that are locked by reads and writes depend on a *locking policy*. The algorithm remains correct for any locking policy, but a poorly chosen policy causes many aborts because there is no common locked timestamp. We present some simple but

interesting algorithms using various locking policies, each with its own advantages.

4 Generic MVTL Algorithm

We now present our generic MVTL algorithm in detail. We start with some basic concepts (§4.1), explain a simple lock extension we use (§4.2), and cover the main algorithm (§4.3).

For simplicity of presentation, we focus on the centralized algorithm, and defer discussion of the distributed version to Appendix H. While more complex, the distributed version is even more practically appealing from a communication efficiency perspective. Some practical considerations for implementing MVTL, including how locks and data can be compacted, are discussed in Section 6.

4.1 Preamble

In multiversion algorithms, the system keeps many data versions for the same key, in an array $Values[k, t]$ where k is a key and t is a timestamp. It is often useful for a process to be able to pick distinct timestamps from another process; we do this by adding a process id to each timestamp; thus, each timestamp consists of a pair (v, p) ordered lexicographically, where v is a real number. There is a smallest timestamp, which we call 0, and a special value, which we call $\perp$, such that initially $Values[k, 0] = \perp$ for every key k.

4.2 Freezable locks

The MVTL algorithm deals with *write-once objects*—objects initially set to $\perp$ that may change their state at most once. We define a simple variation of readers-writer locks, which we call freezable locks, which are appropriate for such objects and we use them in MVTL. A freezable lock is similar to a readers-writer lock, except that a lock holder can freeze the lock to indicate that it will never release it. Freezing is useful because it tells other processes that they should not wait to acquire the lock; we use this feature in several specialized MVTL algorithms. If a lock holder does not freeze a lock, it is expected to release it eventually.

We apply freezable locks to write-once objects as follows. A process acquires the lock in write mode if it intends to write the object. The process may ultimately fail to write if the transaction aborts, in which case it releases the lock; but if the transaction commits, the process freezes its lock to ensure other processes will not try to write the object again. Similarly, a process acquires the lock in read mode to read the object and it freezes the lock in case of a commit; if the object was not written (its state is $\perp$), this prevents other processes from writing to it, sealing its fate.

4.3 Algorithm

Algorithm 2 shows the main code of the generic MVTL algorithm. For clarity, we assume that the code in lines 17–19 is executed atomically, but we later remove this assumption (Section 6). To write a value into key k, a transaction obtains zero or more write-locks on timestamps for that key (function WRITE-LOCKS in line 4). Intuitively, a write-lock on a timestamp t for key k allows the transaction to commit with timestamp t as far as accesses to k are concerned. After getting the locks, the transaction remembers the key and value; the write is not visible to other transactions until the transaction commits.

To read a key, a transaction gets zero or more read-locks on timestamps for that key (function READ-LOCKS in line 7), with the requirement that these timestamps form a contiguous interval that starts immediately after the version that the read returns. For instance, if

Algorithm 2 The generic MVTL algorithm (part 1/2): main code

```
1:  function BEGIN(tx)
2:      tx.readset ← ∅; tx.writeset ← ∅; tx.committs ← ⊥
3:  function WRITE(tx, k, v)                          ▷ write v to k in transaction tx
4:      WRITE-LOCKS(tx, k)                            ▷ write lock some subset of timestamps
5:      add (k, v) to tx.writeset                     ▷ remember key and value we wrote
6:  function READ(tx, k)                              ▷ read k in transaction tx
7:      tr ← READ-LOCKS(tx, k)        ▷ read lock some interval [tr+1, ...] with Values[k, tr] ≠ ⊥
8:      if tr = ⊥ then return ⊥                       ▷ read failed
9:      add (k, tr) to tx.readset                     ▷ remember key and version we read
10:     return Values[k, tr]                          ▷ return committed value
11: function COMMIT(tx)                               ▷ try to commit transaction tx
12:     COMMIT-LOCKS(tx)                              ▷ locks to acquire at commit time
13:     T ← {t : ∀k ∈ tx.readset.keys, tx has a lock on (k, t) and ▷ try to find a locked timestamp for
    tx
              ∀k ∈ tx.writeset.keys, tx has a write-lock on (k, t)}
14:     if T = ∅ then mark tx as aborted
15:     else
16:         tx.committs ← COMMIT-TS(T)                ▷ pick some timestamp in T
17:         for (k, v) ∈ tx.writeset do
18:             freeze write-lock for tx on (k, tx.committs)   ▷ freeze locks
19:             Values[k, tx.committs] ← v            ▷ expose committed value
20:         mark tx as committed
21:     if COMMIT-GC(tx) then GC(tx)                  ▷ invoke gc or not
22: function GC(tx)                                   ▷ garbage collect locks of tx after it ended
23:     if tx committed then
24:         for (k, tr) ∈ tx.readset do
25:             freeze read-locks for tx on [tr+1, tx.committs]
26:     release all unfrozen read- and write-locks for tx
```

$[tr{+}1, te]$ denotes the read-locked timestamps, then the read must return the value committed with timestamp tr. This requirement is necessary for serializability: intuitively, the read locks permit the transaction to commit with any timestamp $t \in [tr{+}1, te]$ after having read v, by preventing other transactions from writing a different value with a timestamp between tr and te. After locking, the transaction remembers k and tr; knowledge of k is necessary to commit, and knowledge of both k and tr is needed to garbage collect the locks of the transaction.

To commit, a transaction gets zero or more additional locks (function COMMIT-LOCKS in line 12) and tries to find a commit timestamp t that is write-locked for every k in the write-set, and that is read- or write-locked for every k in the read-set. (A key in the read-set may be write-locked because the transaction read the key and then wrote it.) If there are many such timestamps, the transaction picks one (function COMMIT-TS in line 16). The transaction then freezes write-locks on that timestamp and records the written values so that they can be seen by other transactions. As an optional step (as determined by calling COMMIT-GC in line 21), the transaction may garbage collect the locks it holds. Doing so freezes the read locks between the version read and the commit timestamp, and releases all other locks. If the algorithm skips garbage collection on commit, garbage collection can be invoked any time later in the background; this is not shown in the code.

The algorithm depends on a policy of what locks to acquire, how to pick one of many possible commit timestamps, and whether to garbage collect during commit; these choices

Algorithm 3 The generic MVTL algorithm (part 2/2): policy

```
1: function WRITE-LOCKS(tx, k)
2:     acquire write-locks for tx on (k, T) for some set T
3: function READ-LOCKS(tx, k)                          ▷ returns a timestamp or ⊥
4:     acquire read-locks for tx on (k, T) for some T = [tr+1, ...] where Values[k, tr] ≠ ⊥
5:     either return tr or return ⊥
6: function COMMIT-LOCKS(tx)
7:     acquire read- or write-locks for tx on some keys and timestamps
8: function COMMIT-TS(T) return some t ∈ T
9: function COMMIT-GC(tx) either return true or return false
```

can depend on the transaction and other considerations. The choices are determined by the functions that we mentioned above: WRITE-LOCKS, READ-LOCKS, COMMIT-LOCKS, COMMIT-TS, and COMMIT-GC. The generic MVTL algorithm uses a generic policy that makes these choices nondeterministically (Algorithm 3). For example, to obtain write locks, the generic policy nondeterministically picks a set T of timestamps to lock. We note that in the READ-LOCKS function, the policy always locks an interval of timestamps starting immediately after a committed version, whose value is returned. Given that read locks are applied for this entire interval, no other write lock or version can exist within this interval. We also note that, as explained in Section 4.1, in the case of write locks, we can choose timestamps that are unique to the process performing the transaction.

We prove that the generic MVTL algorithm is correct with its nondeterministic choices. Naturally, this correctness carries over to any specialization that fixes the nondeterministic choices in any way. These specializations lead to different algorithms (Section 5) that achieve different benefits.

Some policies of the generic algorithm may cause deadlocks, where a process waits forever to acquire a lock. In such cases, standard techniques for deadlock detection can be used to abort the required transactions (e.g., cycle detection in the wait-for graph, timeout, etc).

▶ **Theorem 1.** *The generic MVTL algorithm (Algorithms 2 and 3) ensures serializability.*

We provide a detailed proof in Appendix A. We show that any schedule of our protocol is view equivalent to a serial one-version schedule where transactions are serialized in the order of their commit timestamps. We prove this by showing that the multiversion serialization graph [4] resulting from our protocol is acyclic.

5 Simple MVTL Algorithms

We now give several simple algorithms that are special cases of the generic MVTL algorithm, each with a different benefit. To specify these algorithms, we specialize the generic policy of MVTL (Algorithm 3). Details, including the pseudo-code of the algorithms and the proofs of their benefits, are in the appendices.

5.1 The preferential algorithm

Roughly speaking, the preferential algorithm, denoted MVTL-Pref, works with multiple timestamps for each transaction, where one of the timestamps is preferential. The algorithm tries to commit a transaction using its preferential timestamp, but if doing so would abort, it

tries one of the other timestamps. To ensure viability of the other timestamps, the algorithm locks them as necessary during the execution.

More precisely, MVTL-Pref is parameterized by a function $A(t)$ that takes the transaction's preferential timestamp and returns a non-empty set of alternative timestamps different from t. $A(t)$ is a choice of the user of the algorithm. For example, $A(t) = \{t-10, t+10\}$ indicates that $t-10$ and $t+10$ are the alternative timestamps for a transaction with preferential timestamp t. The preferential timestamp itself comes from a clock, as in other timestamp-based protocols.

We assume that clock timestamps are unique (e.g., by appending the process id to each timestamp t) and that $A(t)$ also produces unique timestamps (e.g., by using the process id in t for each timestamp in $A(t)$).

When executing a read on a key k, the algorithm determines a version to return based on the preferential timestamp, and then read-locks contiguous timestamps of k to cover as many alternative timestamps as possible. When executing a write to key k, the algorithm obtains no locks; rather, locks are acquired at commit time, as follows. If the algorithm cannot obtain a write-lock for the preferential timestamp for each written key, it tries one of the alternative timestamps. If it manages to obtain read- and write-locks for all read and written objects at one of the timestamps, the transaction commits; otherwise it aborts. Appendix B has the full pseudo-code.

We can show that if we choose the alternative timestamps $A(t)$ to be smaller than the preferential timestamps t, then MVTL-Pref aborts strictly fewer workloads compared to MVTO+. More precisely:

▶ **Theorem 2.** *Suppose that $\forall t' \in A(t), t' < t$. (a) If a workload W produces no abort under MVTO+, then W produces no abort under MVTL-Pref. (b) There are infinitely many workloads that produce no aborts under MVTL-Pref but produce aborts under MVTO+.*

5.2 The prioritizer algorithm

Multiversion timestamp ordering provides no way for critical transactions to be prioritized over normal transactions. We explain how MVTL can do that, by using a policy that gives more locks to critical transactions. There are many ways to do that, but the simplest one is as follows. Normal transactions obtain their locks as in multiversion timestamp ordering using synchronized clocks, while critical transactions try to acquire all locks as in pessimistic concurrency control except that critical transactions do not block waiting for any of its locks. Both types of transactions garbage collect on commit. The detailed pseudo-code of the algorithm is given in Appendix C.

▶ **Theorem 3.** *In the MVTL-Prio algorithm, transactions labeled critical are never aborted by transactions labeled normal.*

Given that high-priority transactions behave similarly to pessimistic concurrency control, they can cause deadlocks. However, as we show in Appendix C, transactions with normal priority behave identically to those in MVTO+, and thus never cause deadlocks.

5.3 The ϵ-clock algorithm

Multiversion timestamp ordering uses clocks to obtain its timestamps, but if clocks are not synchronized or monotonic[1], it is susceptible to *serial aborts*—aborts that occur in an

[1] A monotonic clock is one that ensures that it returns a higher timestamp if it is queried later in time. Monotonic clocks and time-synchronized clocks are equivalent insofar this discussion is concerned.

execution that is completely serial. This is a concern in modern multicore machines that do not guarantee that clocks across cores are perfectly synchronized. For example, T_2 gets timestamp 2, reads an object X, and commits. Afterwards, T_1 gets a smaller timestamp 1, writes X, and tries to commit. This will cause T_1 to abort since the read-timestamp of X at version 0 is 2. This is the schedule:

$$\begin{array}{lllll} T_2: & R(X) & C & & \\ T_1: & & & W(X) & A \end{array}$$

Here, time flows to the right and each line shows the operations of a transaction. R, W, C, and A indicate a read, write, commit, and abort; and X is the key. Thus, this schedule has two transactions T_1 and T_2, where T_2 reads X and commits, and then T_1 writes X and aborts.

The MVTL-ϵ-clock algorithm, which we now introduce, avoids serial aborts when used with ϵ-synchronized clocks. Briefly, when it starts, a transaction reads the clock, obtains a time t, and for each read and write tries to lock the interval $[t-\epsilon, t+\epsilon]$. At the end, it commits at the smallest common timestamp it locked for every accessed object. Before completing the commit, the transaction runs garbage collection. In a sequential execution, it is possible to show that *tx* picks a commit timestamp that is at most t, and thus it releases the lock on higher timestamps. As a result, the next transaction in the sequence will always have its own real time in the intersection of locked time points, and therefore does not abort. The detailed pseudo-code of the algorithm is given in Appendix D, as well as the proof of the following:

▶ **Theorem 4.** *The MVTL-ϵ-clock algorithm is not susceptible to serial aborts when clocks are ϵ-synchronized.*

5.4 Existing algorithms as special cases

We now show that MVTL generalizes two popular transactional algorithms, MVTO+ and pessimistic concurrency control. More precisely, we give two algorithms MVTL-TO and MVTL-Pessimistic, which specialize MVTL and behave exactly like MVTO+ and pessimistic concurrency control, respectively

In MVTL-TO, each transaction obtains a timestamp t from a clock when the transaction starts. Writes do not lock anything, reads try to lock $[tr{+}1, t]$ (waiting for unfrozen locks) where tr is the largest timestamp before t for which $\mathit{Values}[k, tr] \neq \perp$, and commits lock t for each object in the transaction's write-set. Garbage collection is not invoked on commit. In Appendix E we give the detailed pseudo-code of the algorithm and the proof of the following:

▶ **Theorem 5.** *The MVTL-TO algorithm behaves as the MVTO+ algorithm.*

Pessimistic concurrency control locks objects before accessing them, thus preventing potentially conflicting operations from executing at the same time. To emulate pessimistic concurrency control using MVTL, writes acquire write locks on all timestamps (blocking), while reads acquire read-locks on all timestamps in $[tr{+}1, \infty]$ (blocking). Garbage collection is invoked on commit. In Appendix F we give the detailed pseudo-code of the algorithm and the proof of the following:

▶ **Theorem 6.** *The MVTL-Pessimistic algorithm behaves as the pessimistic concurrency control algorithm.*

5.5 The ghostbuster algorithm

Under multiversion timestamp ordering, a transaction may abort and later create a conflict with another transaction, causing it to abort. For example, suppose that T_1 starts with timestamp 1, T_2 starts with timestamp 2, and T_3 starts with timestamp 3. Then T_3 reads X and commits, T_2 reads Y, writes X, and tries to commit with its timestamp 2, but T_2 aborts because T_3 read X with timestamp 3. Next T_1 writes Y and tries to commit but aborts due to the read by T_2. This is a ghost abort, because the write of T_1 has a conflict with a transaction T_2 that had aborted before the write of T_1 started. This is the schedule:[2]

$$\begin{array}{lccccccc} T_3: & R(X) & C & & & & & \\ T_2: & & & R(Y) & W(X) & A & & \\ T_1: & & & & & & W(Y) & A \end{array}$$

We define ghost aborts precisely in Appendix G.

While multiversion timestamp ordering has ghost aborts, MVTL-Ghostbuster can avoid that. MVTL-Ghostbuster is a simple modification to the MVTL-TO algorithm (Section 5.4): when a transaction commits, it performs garbage collection. This ensures that transactions that abort do not leave behind locks that cause ghost aborts. In Appendix G we give the detailed pseudo-code of the algorithm and the proof of the following:

▶ **Theorem 7.** *The MVTL-Ghostbuster algorithm is not susceptible to ghost aborts.*

6 Practical considerations: lock state space and atomic blocks

Reducing lock state space. When we presented the generic MVTL algorithm, we defined a lock for each timestamp and object, which amounts to an infinite lock state space. We did not include mechanisms to compress this information; such mechanisms are orthogonal to the essence of the algorithm. However, a practical implementation should compress the lock state space. To do so, we observe that MVTL algorithms usually acquire and release locks on a small number of points or contiguous intervals (this is true for all algorithms we presented). Rather than keeping a lock state for each timestamp, an implementation should keep a single lock state for the entire interval. In the algorithms we presented, each object holds at most one lock interval per committed transaction. Furthermore, this state can be discarded when the associated version of the object is purged, as we discuss next.

Purging versions. By its nature, a multiversion algorithm keeps multiple versions of each object; this is true not just for MVTL but also for every other multiversion algorithm. Doing so is sensible as storage prices fall. In fact, disk systems such as databases already employ multiversion algorithms, but even memory systems are targets now. Nevertheless, multiversion algorithms need a way to purge old versions, so that each object has a small number of versions—possibly just one version after write activity on the object quiesces. We now explain how this can be done in MVTL. This is easy: at any time, the system can purge any version older than the latest committed one, without affecting the correctness of the algorithm. Transactions that need that version will abort, so in practice we purge versions older than a time limit chosen based on the duration of its longest transactions. In some MVTL algorithms, there is a lower bound on the timestamps that a transaction locks (e.g.,

[2] Here, transactions get a timestamp before their first operation, but one can construct a more complex schedule with the same problem even if transactions get a timestamp at the first operation.

ϵ-clock algorithm); we can purge versions with timestamps below the bound except the last one before the bound without causing any side-effects.

Removing the atomic block. Algorithm 2 has an atomic block in lines 17–19, to avoid partially exposing the writes of a committing transaction when we assign to the array $Values[k, t]$. We can remove this atomic block by (1) first storing a special value in $Values[k, t]$ for all timestamps in the for loop, (2) then storing the actual value v for all timestamps in the loop, and (3) having other processes wait if they read *Values* and see the special value.

7 Related Work

The main novelty of this work is the idea of locking individual timestamps, leading to a genre of multiversion algorithms called MVTL. No other work proposes this idea, but because MVTL is a broad class, several existing algorithms become special cases of MVTL, leading to similarities in mechanism.

Multiversion concurrency control is an old idea [4] that has seen a resurgence in software transactional memory (STM) systems, several of which provide serializability [1, 6, 10, 15, 16, 20, 22–24]. Prior work in this space falls into three categories: (1) multiversion for read-only transactions, (2) conflict graph schemes, and (3) multiversion timestamp ordering algorithms. The first category [10, 22–24] are systems that use multiversion to benefit solely read-only transactions; update transactions rely on optimistic methods that, upon commit, validate the read-set and abort if any object has changed. While read-only transactions are important, these methods abort under simple concurrent update schedules, such as the following (where full multiversion schemes do not abort):

$$\begin{array}{lccc} T_1: & R(X) & & W(Y) \\ T_2: & & W(X) & \end{array}$$

The second category [1, 16, 20] are multiversion STM systems that ensure serializability by detecting cycles in the *conflict graph*—a data structure that represents the conflicts across transactions—similarly to the MVSGT algorithm [26]. These algorithms have two drawbacks: they are complex and they incur significant computation overhead, as reported in some of these papers.

The third category [17] are systems that extend multiversion timestamp ordering. Specifically, Kumar et al. [17] explain how to provide opacity, which is stronger than serializability. However, the algorithm suffers from the same drawbacks of multiversion timestamp ordering that we address in Section 5. It should be possible to extend MVTL to provide opacity using the ideas of Kumar et al. [17], but this is future work.

Lomet et al. [19] introduce the multiversion timestamp range algorithm (MVTR). With MVTR, each transaction is assigned a range of timestamps, and this range shrinks as the transaction executes; at the end, MVTR commits if the range is non-empty. MVTR differs from MVTL because MVTR locks entire objects instead of timestamps. As a result, MVTR does not enjoy the full benefits of multiversion concurrency control, such as allowing two concurrent transactions to write the same object. Also, with MVTR one transaction manipulates the inner state of another transaction (e.g., by changing the range that another transaction uses), which synchronizes through a transaction scheduler or locks. Another context where ranges of timestamps have been used are elastic transactions [11], an STM technique aimed at search data structures. The system maintains a single version of objects, timestamped with the time they were last updated. Transactions keep track of a range of timestamps that determines if they can commit, based on their start time and the time when

the accessed objects were last written to. A pessimistic two-phase locking protocol is used in order to commit.

Snapshot isolation [2] is both an isolation property and a protocol. The protocol uses multiversioning and timestamps, similarly to multiversion timestamp ordering, but it does not provide serializability. Other protocols that use multiversioning and timestamps provide even weaker notions than snapshot isolation [25].

Optimistic concurrency control [18] is another technique that can use multiversioning. Essentially, transactions do not acquire any locks when running. At commit time, the transaction verifies that the versions it has read still represent the latest data, and, if this is the case, applies updates. More recent work, such as the TicToc [28] has optimized the OCC protocol in order to adaptively serialize transactions based on the data they access. TicToc computes potential serialization points for the transaction before the validation and commit phases. Thus, a transaction for which the read and write sets have been inspected might later abort. In contrast, MVTL ensures that once a serialization point for the transaction has been found, the transaction is guaranteed to commit. Similarly, Faleiro et al. [9] propose Bohm, a multi-version protocol that pre-orders transactions before execution, and is in this sense more pessimistic than MVTL, which determines transaction ordering dynamically during execution. In addition, Bohm requires that the transaction be known ahead of time, and that the write-set be static. MVTL in contrast is able to operate with entirely dynamic transactions.

Our concept of policy is somewhat similar to that of a contention manager [12] in transactional memory, in the sense that it can be implemented in a large variety of ways, which may exhibit very different performance characteristics, but do not jeopardize safety. Nevertheless, the two concepts refer to different components of a transactional system.

Many practical systems providing distributed transactions only ensure snapshot isolation [8, 21], and do not allow concurrent writes for the same object. Other systems, such as Spanner [7], use two-phase locking in the case of read-write transactions, and thus provide very limited parallelism for such transactions. Spanner provides strict serializability (i.e., the serialization order of transactions conforms to their real-time execution order), and read-only transactions that acquire no locks and are guaranteed to succeed. MVTL also guarantees that read-only transactions never abort, and, given access to synchronized clocks, can also avoid any locking for such transactions and provide strict serializability. In summary, MVTL improves on the level of parallelism offered by such systems, while offering similar or stronger guarantees.

8 Conclusion

This paper introduced a new genre of multiversion concurrency control algorithms called multiversion timestamp locking (MVTL). MVTL offers a new way to look at multiversion algorithms, based on locking individual time points. With this perspective, we can find simple algorithms that improve the state of the art in various ways: by committing successfully more workloads than existing multiversion protocols, by avoiding the problems of serial aborts and ghost aborts, and by offering prioritized transactions. We can also view existing algorithms, such as MVTO and pessimistic concurrency control, as special cases of MVTL. Finally, we showed how to realize MVTL in both centralized and distributed systems.

We believe that the algorithms proposed here are only a starting point for many other possibilities opened up by MVTL. The design of other MVTL algorithms is a promising direction for future research.

References

1 Utku Aydonat and Tarek S. Abdelrahman. Serializability of transactions in software transactional memory. In *ACM SIGPLAN Workshop on Transactional Computing*, February 2008.

2 Hal Berenson et al. A critique of ANSI SQL isolation levels. In *International Conference on Management of Data*, 1995.

3 Philip A. Bernstein and Nathan Goodman. Concurrency control in distributed database systems. *ACM Computing Surveys*, 13(2):185–221, June 1981.

4 Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency control and recovery in database systems*. 1987.

5 Christian Cachin, Rachid Guerraoui, and Luis Rodrigues. *Introduction to reliable and secure distributed programming*. Springer, 2011.

6 João Cachopo and António Rito-Silva. Versioned boxes as the basis for memory transactions. *Sci. Comput. Program.*, 2006.

7 J. Corbett et al. Spanner: Google's globally-distributed database. In *Symposium on Operating Systems Design and Implementation*, 2012.

8 Jiaqing Du, Sameh Elnikety, and Willy Zwaenepoel. Clock-SI: Snapshot isolation for partitioned data stores using loosely synchronized clocks. In *IEEE Symposium on Reliable Distributed Systems*, pages 173–184, October 2013.

9 Jose M Faleiro and Daniel J Abadi. Rethinking serializable multiversion concurrency control. *Proceedings of the VLDB Endowment*, 8(11):1190–1201, July 2015.

10 Pascal Felber, Christof Fetzer, Patrick Marlier, and Torvald Riegel. Time-based software transactional memory. *IEEE Transactions on Parallel and Distributed Systems*, 2010.

11 Pascal Felber, Vincent Gramoli, and Rachid Guerraoui. Elastic transactions. In *International Symposium on Distributed Computing*, 2009.

12 Rachid Guerraoui, Maurice Herlihy, and Bastian Pochon. Toward a theory of transactional contention managers. In *ACM Symposium on Principles of Distributed Computing*, 2005.

13 Vassos Hadzilacos and Sam Toueg. A modular approach to fault-tolerant broadcasts and related problems. Technical Report TR 94-1425, Cornell University, Dept. of Computer Science, Cornell University, Ithaca, NY 14853, May 1994.

14 Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming, Revised First Edition*. 2012.

15 Idit Keidar and Dmitri Perelman. Multi-versioning in transactional memory. In *Transactional Memory. Foundations, Algorithms, Tools, and Applications*. 2015.

16 Idit Keidar and Dmitri Perelman. On avoiding spare aborts in transactional memory. *ACM Transactions on Computer Systems*, 57(1):261–285, July 2015.

17 Priyanka Kumar, Sathya Peri, and K. Vidyasankar. A timestamp based multi-version STM algorithm. In *International Conference on Distributed Computing and Networking*, 2014.

18 Hsiang-Tsung Kung and John T Robinson. On optimistic methods for concurrency control. *ACM Transactions on Database Systems (TODS)*, 1981.

19 David Lomet, Alan Fekete, Rui Wang, and Peter Ward. Multi-version concurrency via timestamp range conflict management. In *International Conference on Data Engineering*, 2012.

20 Jeff Napper and Lorenzo Alvisi. Lock-free serializable transactions. Technical report, University of Texas at Austin, 2005.

21 Daniel Peng and Frank Dabek. Large-scale incremental processing using distributed transactions and notifications. In *Symposium on Operating Systems Design and Implementation*, 2010.

22 Dmitri Perelman, Anton Byshevsky, Oleg Litmanovich, and Idit Keidar. SMV: Selective multi-versioning STM. In *International Symposium on Distributed Computing*, pages 125–140, September 2011.

23 Dmitri Perelman, Rui Fan, and Idit Keidar. On maintaining multiple versions in stm. In *ACM Symposium on Principles of Distributed Computing*, pages 16–25, July 2010.

24 Torvald Riegel, Pascal Felber, and Christof Fetzer. A lazy snapshot algorithm with eager validation. In *International Symposium on Distributed Computing*, pages 284–298, September 2006.

25 Yair Sovran, Russell Power, Marcos K. Aguilera, and Jinyang Li. Transactional storage for geo-replicated systems. In *ACM Symposium on Operating Systems Principles*, October 2011.

26 Gerhard Weikum and Gottfried Vossen. *Transactional information systems: theory, algorithms, and the practice of concurrency control and recovery*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2001.

27 Yingjun Wu, Joy Arulraj, Jiexi Lin, Ran Xian, and Andrew Pavlo. An empirical evaluation of in-memory multi-version concurrency control. *Proceedings of the VLDB Endowment*, 10(7):781–792, March 2017.

28 Xiangyao Yu, Andrew Pavlo, Daniel Sanchez, and Srinivas Devadas. TicToc: Time Traveling Optimistic Concurrency Control. In *International Conference on Management of Data*, 2016.

A Correctness of generic MVTL algorithm

Theorem 1. *The generic MVTL algorithm (Algorithms 2 and 3) ensures serializability.*

Proof. We denote by $T.committs$ the timestamp at which transaction T is serialized and commits (aborted transactions do not have a serialization timestamp). Each transaction has a unique serialization timestamp, as explained in Section 4.1. If a transaction T commits at a timestamp $T.committs$, then it holds write locks at $T.committs$ for all the data in its write set, and read locks from the largest timestamp smaller than $T.committs$ containing a committed value to $T.committs$ for all the data in its read set (Algorithm 2, line 13). We denote by $r_i[x_j]$ the fact that transaction T_i has read a version of object x written by transaction T_j (i.e., the read operation has returned $Values[x, T_j.committs]$). In addition, we denote by $w_k[x_k]$ the fact that transaction T_k has written a new version of object x (i.e., it has written a value to $Values[x, T_k.committs]$).

We assume the serialization order is given by the commit timestamp of the transaction. That is, if transaction T_1 creates version v_1 of object o, and transaction T_2 creates version v_2 of object o, we say $v_1 \ll v_2$ iff $T_1.committs < T_2.committs$.

Let H be a multiversion history over a set of transactions $\{T_0, \ldots, T_n\}$, and $C(H)$ the committed projection of this history. The committed projection of an operation history retains only the operations that belong to committed transactions. A multiversion serialization graph (MVSG) has the transactions $\{T_0, \ldots, T_n\} \in C(H)$ as vertices and edges (1) from T_i to T_j if T_j reads from T_i, and (2) for $r_k[x_j]$ and $w_i[x_i] \in C(H)$, if $x_i \ll x_j$, then the graph has an edge from T_i to T_j, otherwise it has an edge from T_k to T_i.

It has been shown [4] that if the multiversion serialization graph is acyclic, then a multiversion history is *one copy serializable*, that is, equivalent to a serial one version history.

Similarly to the proof of the original multiversion timestamp order algorithm, we show the MVSG resulting from MVTL is acyclic by showing that if an edge between T_i and T_j exists in the graph, $T_i.committs < T_j.committs$. We consider the types of edges that can appear in a multiversion serialization graph. The first type of edges are *reads-from edges*. In this case, transaction T_j reads a version written by transaction T_i. Function READ-LOCKS acquires locks for timestamps starting immediately after the timestamp containing the version whose value is returned (and, since it read-locks an interval of timestamps, does not lock timestamps equal or larger to later versions). Hence, the read can only be serialized at a timestamp higher than that at which the read version was created. Thus, $T_i.committs \leq T_j.committs$. The second type of edge appears if $r_k[x_j]$ and $w_i[x_i]$ are in H and $x_i \ll x_j$. In this case, an edge from T_i to T_j exists in the graph. By definition of $\ll$, $x_i \ll x_j$ iff $T_i.committs < T_j.committs$. Finally, the third type of edge appears if $r_k[x_j]$ and $w_i[x_i]$ are in H and $x_j \ll x_i$. In this case, an edge from T_k to T_i is created (this assumes $k \neq i$). Since $x_j \ll x_i$, we know that $T_j.committs < T_i.committs$. Given that T_k has performed a read of version x_j, T_k has necessarily applied read locks for each timestamp from $T_j.committs + 1$ to $T_k.committs$. A read lock can only be acquired if no write lock from another transaction is present. Similarly, a write lock on a timestamp cannot be acquired if a read lock from another transaction is present. Thus, $w_i[x_i]$ could not have occurred in the interval $[T_j.committs + 1, T_k.committs]$. And since we know $T_j.committs < T_i.committs$, $w_i[x_i]$ must have necessarily occurred after the interval. Thus, $T_k.committs < T_i.committs$. Given that all the edges in the graph are from transactions with lower serialization timestamps to transactions with higher serialization timestamps, a cycle cannot exist. Thus, H is one-copy serializable. ◀

B Details of the preferential algorithm

The MVTL-Pref algorithm is given in Algorithm 4. Each transaction is assigned a *preferential timestamp* and one or more alternative timestamps. The system tries to commit the transaction using first the preferential timestamp, but if that would abort the transaction, it tries the alternative timestamps. Transactions are serialized in the order of their commit timestamps.

More precisely, the algorithm is parameterized by a function $A(t)$ that takes the transaction's preferential timestamp and returns a non-empty *set* of alternative timestamps different from t. For example, $A(t) = \{t-10, t+10\}$ indicates that $t-10$ and $t+10$ are the alternative timestamps for a transaction with preferential timestamp t. The preferential timestamp itself comes from a clock, as in other timestamp-based protocols. Similarly, we assume that processes obtain unique timestamps (e.g., by appending the process id to each timestamp t) and that $A(t)$ also produces unique timestamps (e.g., by using the process id in t in each timestamp in $A(t)$).

When reading, the system acquires read-locks for a set that includes the preferential timestamp and as many other timestamps as possible. When committing, the system tries to write-lock on all objects in the write set and the preferential timestamp; if that is not possible, it tries each of the alternative timestamps.

We provide a more precise definition of the concept of a workload:

▶ Definition 1. A *workload* is a set of n transaction inputs, where each transaction input is a finite sequence of operation-timestamp pairs with increasing timestamps and an operation is either *read*(k), *write*(k, v) or *tryCommit*.

We now show that under certain conditions on $A(t)$, MVTL-Pref is strictly better than MVTO+, in the sense that (a) if MVTO+ does not abort under a workload, then MVTL-Pref does not abort either, and (b) there are infinitely many workloads where MVTO+ aborts but MVTL does not. These results hold assuming that $A(t)$ contain only timestamps smaller than t, that is, the alternative timestamps are smaller than the preferential one.

Theorem 2. *Suppose that* $\forall t' \in A(t), t' < t$. *(a) If a workload W produces no abort under MVTO+, then W produces no abort under MVTL-Pref. (b) There are infinitely many workloads that produce no aborts under MVTL-Pref but produce aborts under MVTO+.*

Proof sketch. (a) Consider a workload W that does not abort under MVTO+. We prove that, for each transaction T in W, the execution of T under MVTO+ and MVTL-Pref will read- and write-lock exactly the same timestamps. The intuition here is that MVTL-Pref will choose the same timestamps as MVTO+ under workload W, because W does not cause any aborts. More precisely, we can show that (i) whenever a read occurs, both MVTO+ and MVTL-Pref pick the same value to return for the read (the first non-$\perp$ value with a timestamp smaller than the preferential timestamp); because the preferential timestamp is higher than any of the timestamps in $A(t)$, the MVTL-Pref picks the preferential timestamp as *tmax* and therefore locks the same range as MVTO+. Moreover (ii), whenever a commit occurs, both MVTO+ and MVTL-Pref pick the same timestamp to lock. This is because MVTL-Pref picks the preferential timestamp, given that MVTO+ does not abort. From (i) and (ii), it is possible to show that MVTL-Pref executes in exactly the same way as MVTO+ under W. Therefore, MVTL-Pref does not abort any transactions under W.

(b) Pick three timestamps $t_1 < t_2 < t_3$ such that $\max A(t_2) < t_1$. These will be the timestamps for transactions T_1, T_2, T_3. Consider the following workload: $W_1(Y)\, C_1\, R_2(X)\, R_3(Y)\, C_3\, W_2(Y)\, C_2$. Under MVTO+, this workload aborts T_2 since the timestamp at which T_2 wants to write Y

Algorithm 4 The MVTL-Pref algorithm

```
1: function Initialization(tx)
2:     tx.PrefTS ← clock()
3:     tx.PossTS ← {tx.PrefTS} ∪ A(tx.PrefTS)                    ▷ possible timestamps for tx
4: function write-locks(tx, k) return                            ▷ lock write-set only on commit
5: function read-locks(tx, k)
6:     repeat
7:         tr ← max{t : t < tx.PrefTS and Values[k, t] ≠ ⊥}      ▷ Candidate value to read
8:         tmax ← max{t ∈ tx.PossTS : no timestamps in[tr+1, tmax] are write frozen}
9:         for t ← tr+1 to tmax do                               ▷ read-lock interval [tr+1, tx.TS] if possible
10:            try to acquire read-lock for tx on (k, t), waiting
                   if timestamp is write-locked but not frozen
11:            if found frozen write-lock then release read-locks acquired above; break  ▷ exit the "for" loop
12:    until found no frozen locks in the for loop
13:    tx.PossTS ← tx.PossTS ∩ [tr, tmax]                        ▷ update possible timestamps
14:    return tr
15: function commit-locks(tx)
16:    for t ∈ tx.PossTS do  ▷ Find a good timestamp. Loop order: first tx.TS then arbitrary for PossTS
17:        gotlocks ← true
18:        for (k, tr) ∈ tx.writeset do
19:            try to write-lock for tx on (k, t), without waiting if a timestamp is read-locked
20:            if write-lock not acquired then
21:                gotlocks ← false                              ▷ this timestamp will not work
22:                release all write locks for tx
23:                break                                         ▷ exit inner "for" loop
24:        if gotlocks then break ▷ found a timestamp for which we can get write locks; exit outer "for" loop
25:    if gotlocks then tx.TS ← t                                ▷ found good timestamp
26:    else tx.TS ← ⊥                                            ▷ no good timestamps
27: function commit-ts(T) return tx.TS
28: function commit-gc(tx) return false
```

is between t_1 and t_3. However, under MVTL-Pref, T_2 commits because MVTL-Pref can pick the alternative timestamp $\max A(t_2)$ with which to commit T_2. It is easy to generalize this example to several transactions, and thus obtain infinitely many workloads where MVTO+ causes an abort but MVTL-Pref does not. ◀

C Details of the prioritizer algorithm

The MVTL-Prio algorithm is given in Algorithm 5. Operations from transactions with priority try to lock timestamps up to $+\infty$: writes attempt to lock all timestamps, while reads lock from the latest observed write onwards; the transaction commits at the lowest timestamp that was locked for all its data items. In contrast, transactions with no priority behave identical to the MVTO+ algorithm: they read the clock at the beginning and try to serialize all operations at that point (thus only acquiring locks for timestamps lower than or equal to the clock value at the beginning of the transaction).

Algorithm 5 The MVTL-Prio algorithm

```
1: function INITIALIZATION(tx)
2:     if tx.priority = false then tx.TS ← clock()
3: function WRITE-LOCKS(tx, k)
4:     if tx.priority = true then
5:         for t = +∞ downto 0 do                    ▷ write-lock all the possible timestamps
6:             try to acquire write-lock for tx on (k, t), waiting
               if a timestamp is read- or write-locked but not frozen
7: function READ-LOCKS(tx, k)
8:     if tx.priority = true then
9:         repeat
10:            tr ← max{t : t < tx.TS and Values[k, t] ≠ ⊥}
11:            for t = +∞ downto tr+1 do          ▷ read-lock interval [tr+1, +∞] if possible
12:                try to acquire read-lock for tx on (k, t), waiting
                    if timestamp is write-locked but not frozen
13:                if found frozen write-lock then release read-locks acquired above; break  ▷ exit
   the "for" loop
14:        until found no frozen locks in the for loop
15:    else
16:        repeat
17:            tr ← max{t : t < tx.TS and Values[k, t] ≠ ⊥}
18:            for t = tr+1 to tx.TS do           ▷ read-lock interval [tr+1, tx.TS] if possible
19:                try to acquire read-lock for tx on (k, t), waiting
                    if timestamp is write-locked but not frozen
20:                if found frozen write-lock then release read-locks acquired above; break  ▷ exit
   the "for" loop
21:        until found no frozen locks in the for loop
22:    return tr
23: function COMMIT-LOCKS(tx)
24:    if tx.priority = false then
25:        for (k, tr) ∈ tx.writeset do
26:            try to write-lock for tx on (k, tx.TS), without waiting if a timestamp is read-locked
27:            if write-lock not acquired then
28:                tx.TS = ∅ and release all write locks for tx;
29:                return ;
30: function COMMIT-TS(T)
31:    if tx.priority = true then
32:        return min T
33:    else
34:        return tx.TS
35: function COMMIT-GC(tx)
36:    if tx.priority = true then
37:        return true
38:    else
39:        return false
```

Theorem 3. *In the MVTL-Prio algorithm, transactions labeled critical are never aborted by transactions labeled normal.*

Proof sketch. Assume $maxts$ is the maximum serialization timestamp of all completed or executing transactions with no priority. For any objects, transactions without priority will not prevent a transaction with priority from locking the interval $[maxts, +\infty]$, and thus committing at a timestamp at most $maxts$. Thus, transactions without priority cannot cause a transaction with priority to abort. ◀

D Details of the ϵ-clock algorithm

The MVTL-ϵ-clock algorithm is shown in Algorithm 6. It assumes that clocks are ϵ-synchronized and ensures that transactions never abort in serial executions.

Algorithm 6 The MVTL-ϵ-clock algorithm

```
1: function INITIALIZATION(tx)
2:     now ← clock()
3:     tx.TS ← [now − ε, now + ε]
4: function WRITE-LOCKS(tx, k)
5:     try to write-locks for tx on (k, tx.TS), waiting
           if a timestamp is read- or write-locked but not frozen
6:     tx.TS ← write-locks that tx could acquire
7: function READ-LOCKS(tx, k)
8:     if tx.TS = ∅ then return ⊥
9:     m ← max tx.TS
10:    repeat
11:        tr ← max{t : t < m and Values[k, t] ≠ ⊥}
12:        for t = tr+1 to m do                ▷ read-lock interval [tr+1, m] if possible
13:            try to acquire read-lock for tx on (k, t), waiting
                   if timestamp is write-locked but not frozen
14:            if found frozen write-lock then release read-locks acquired above; break ▷ exit the
    "for" loop
15:    until found no frozen locks in the for loop
16:    tx.TS ← tx.TS ∩ [tr+1, m]
17:    return tr
18: function COMMIT-LOCKS(tx) return
19: function COMMIT-TS(T) return min T
20: function COMMIT-GC(tx) return true
```

Upon start, a transaction tx reads the clock, obtains a time t, and sets a local variable $tx.TS$ to the interval $[t-\epsilon, t+\epsilon]$. This set has the timestamps that tx tries to lock as it executes. To write k, tx obtains a write-lock on as many timestamps in $tx.TS$ as possible, waiting if any of the timestamps is read- or write-locked (but not frozen) by another transaction; if tx already holds a read-lock on a timestamp, it waits until it can upgrade it to a write-lock. Next, if Tw denotes the locks that tx actually manages to acquire, tx sets $tx.TS$ to Tw.

To read k, tx selects the largest timestamp m in $tx.TS$, finds the largest timestamp $tr < m$ under which k has been written, and then tries to acquire a read-lock on $[tr+1, m]$ (if tx already has a write-lock then it does not need to acquire a read-lock), waiting if a timestamp is write-locked (but not frozen) by another transaction. tx may find a frozen write-lock if

some other transaction commits after *tx* picked *tr*; In that case, *tx* picks *tr* again and retries. Then *tx* updates *tx.TS* to contain the locked timestamps.

To commit, *tx* picks the smallest locked timestamp and runs garbage collection before completing the commit.

Note that initially *tx.TS* contains the correct real-time *treal* when *tx* started. In a sequential execution, we show that *tx* picks a commit timestamp that is at most *treal*, and thus it releases the lock on higher timestamps. As a result, the next transaction in the sequence will always have its own real time in its *tx.TS*, so that does not abort.

We now show that MVTL-ϵ-clock is not susceptible to serial aborts, which we define precisely as follows:

- *(Serial abort)* An algorithm is susceptible to serial aborts if it has a serial schedule that aborts some transaction.

Theorem 4. *The MVTL-ϵ-clock algorithm is not susceptible to serial aborts when clocks are ϵ-synchronized.*

Proof sketch. According to the ϵ-clock assumption, the local clock the transaction sees can diverge from the real time by at most ϵ. The first step a transaction takes when it starts is to read its local clock t. Assume t_{real_start} is the real time when local clock value t is read. Given that T starts with the interval $[t-\epsilon, t+\epsilon]$, it is guaranteed that $t_{real\_start} \in [t-\epsilon, t+\epsilon]$. At commit time, according to the ϵ-clock algorithm, a transaction commits with the smallest timestamp in its interval it was able to lock for all data items.

We show that if all transactions execute serially, each transaction will be able to commit, and that its commit point will not be larger than the real time at the beginning of the transaction. We prove this by induction:

Base case. Assume T_1 is the first transaction that executes serially in the system. The first point in its assigned interval $(t-\epsilon)$ will be at most equal to the real time at the start of the transaction. Given that no conflicting data exists in the system, this first transaction will be able to commit at this smallest timestamp in the interval.

Inductive step. Assume $n-1$ transactions have executed serially, and have each committed at a timestamp that was at most equal to the real time at the respective start of the transaction. We now show the n-th serial transaction will also commit with a timestamp at most equal to the real time at which it started.

Given that transactions execute serially, we know that the n-th transaction begins only after the previous one has completed. According to the algorithm, a transaction completes only after it performs garbage collection. Therefore, assuming the transactions committed with timestamps at most equal to the real time when they started, after the first $n-1$ transactions commit, no lock is held for timestamps higher than the real time the $n-1$-th transaction started. As the transactions execute serially, the real time the n-th transaction starts is larger than the real time any of the previous transactions started, and thus higher than any lock held in the system (therefore, no conflict can arise for a serial transaction that tries to commit at this timestamp). As the interval assigned to transaction n is guaranteed to contain the real time at the transaction's start, the n-th transaction will be able to commit with a timestamp at most equal to the real time when it started.

◀

If concurrent transactions start less than $2 * \epsilon$ time apart in real time, since operations always wait if timestamps are locked but not frozen, they may have to wait for each other's operations to complete. Therefore our algorithm intuitively behaves similarly to pessimistic

concurrency control for these transactions. Thus, the trade-off with this algorithm is that deadlocks are possible, and the system requires a deadlock detection mechanism.

E Details of the MVTL-TO algorithm

The MVTL-TO algorithm is given in Algorithm 7. Each transaction chooses a serialization timestamp at the beginning, and attempts to serialize every operation at this timestamp. For reads, it finds the largest timestamp with a committed value smaller than its chosen serialization timestamp, applies read locks to every timestamp between these two, and returns the version's value. This is equivalent to reading the version with the largest timestamp smaller than the transaction timestamp and setting its *read-timestamp* in MVTO+. If a read encounters a timestamp that is write-locked, but not frozen, it waits. This wait is short: it stops when write locks that are not frozen are finally frozen.

For writes, the algorithm simply retains the values it wishes to write in its write set, without acquiring any locks. Only at commit time does the protocol try to lock the write set at the chosen serialization timestamp. If any read lock is encountered (frozen or not), the write lock is unsuccessful (since no garbage collection is performed). When a transaction fails to acquire a write lock, it releases all previously acquired write locks, and aborts. In case all the write locks are successfully acquired, they are then frozen and values are associated with the transaction's timestamp.

Algorithm 7 The MVTL-TO algorithm

```
1:  function INITIALIZATION(tx)
2:      tx.TS ← clock()
3:  function WRITE-LOCKS(tx, k) return
4:  function READ-LOCKS(tx, k)
5:      repeat
6:          tr ← max{t : t < tx.TS and Values[k, t] ≠ ⊥}
7:          for t ← tr+1 to tx.TS do                ▷ read-lock interval [tr+1, tx.TS] if possible
8:              try to acquire read-lock for tx on (k, t), waiting
                    if timestamp is write-locked but not frozen
9:              if found frozen write-lock then release read-locks acquired above; break ▷ exit the
    "for" loop
10:     until found no frozen locks in the for loop
11:     return tr
12: function COMMIT-LOCKS(tx)
13:     for (k, tr) ∈ tx.writeset do
14:         try to write-lock for tx on (k, tx.TS), without waiting if a timestamp is read-locked
15:         if write-lock not acquired then
16:             tx.TS = ∅ and release all write locks for tx
17:             return ;
18: function COMMIT-TS(T) return tx.TS
19: function COMMIT-GC(tx) return false
```

Theorem 5. *The MVTL-TO algorithm behaves as the MVTO+ algorithm.*

Proof sketch. Like MVTO+, MVTL-TO processes transactions such that they appear to execute in the order of their timestamp. The protocol provides all the properties of MVTO+, such as reads never aborting and only having read-write conflicts (given each process can choose unique timestamps, writes never conflict with other writes). ◀

F Details of the MVTL-Pessimistic algorithm

Briefly, the pessimistic concurrency control algorithm works as follows: as reads and writes are executed, they apply locks on the objects they access. At most one write can access any object at a point in time. If an object is locked for a write, no reads from other transactions can proceed concurrently. If a transaction cannot acquire a lock for an object, it waits until the lock is released. When all the locks are successfully acquired, the transaction performs its updates to the objects, and then unlocks.

This algorithm can be seen as a special case of MVTL with a specific policy, as shown in Algorithm 8. Basically, writes try to lock all possible timestamps, starting from $+\infty$ downwards, while reads also start from $+\infty$, and apply read locks to all timestamps down to the first timestamp where a write committed (whose value is also returned). If a transaction has successfully acquired locks for all its data, it will commit at the minimum timestamp that is locked for every data item (since such a timestamp always exists, the transaction will not abort—aborts can only potentially occur in case of deadlock). This timestamp will be equal to one greater than the largest timestamp of any read data, and is guaranteed to be less than $+\infty$. At the end of the transaction, the unneeded locks are released (including, in particular $+\infty$) and the next transaction can acquire locks for the concerned data items.

Proof sketch. Since both reads and writes first try to lock $+\infty$, it is guaranteed that at most one writer or multiple readers can have access to an object. Moreover, a transaction that has completed will never prevent other transactions from accessing any data object. ◀

Algorithm 8 The MVTL-Pessimistic algorithm

```
1: function WRITE-LOCKS(tx, k)
2:     for t = +∞ downto 0 do                          ▷ write-lock all the possible timestamps
3:         try to acquire write-lock for tx on (k, t), waiting
               if a timestamp is read- or write-locked but not frozen
4: function READ-LOCKS(tx, k)
5:     repeat
6:         tr ← max{t : t < m and Values[k, t] ≠ ⊥}
7:         for t = +∞ downto tr+1 do                   ▷ read-lock interval [tr+1, +∞] if possible
8:             try to acquire read-lock for tx on (k, t), waiting
                   if timestamp is write-locked but not frozen
9:             if found frozen write-lock then release read-locks acquired above; break ▷ exit the
   "for" loop
10:    until found no frozen locks in the for loop
11:    return tr
12: function COMMIT-LOCKS(tx) return
13: function COMMIT-TS(T) return min T
14: function COMMIT-GC(tx) return true
```

Theorem 6. *The MVTL-Pessimistic algorithm behaves as the pessimistic concurrency control algorithm.*

G Details of the ghostbuster algorithm

We now give the MVTL-Ghostbuster algorithm, which avoids ghost aborts. We start with a precise definition of ghost aborts. To do so, we first define the notion of an active conflict,

which intuitively means a conflict with a transaction that is concurrently running. More precisely, given an execution of algorithm:

- *(Active conflict)* A transaction T_i has an active conflict if it has an operation o_i that conflicts with some operation o_j of another transaction T_j, where o_i is concurrent with T_j.
- *(Ghost abort)* An algorithm is susceptible to ghost aborts if it has a schedule where a transaction aborts but it has no active conflicts.[3]

To avoid ghost aborts, an algorithm must ensure that each transaction that aborts has at least one operation with an active conflict.

The MVTL-Ghostbuster algorithm is shown in Algorithm 9. This algorithm is similar to MVTL-TO, which emulates MVTO, with the addition of garbage collection before a transaction commits or aborts.

Algorithm 9 The MVTL-Ghostbuster algorithm

```
1: function INITIALIZATION(tx)
2:     tx.TS ← clock()
3: function WRITE-LOCKS(tx, k) return
4: function READ-LOCKS(tx, k)
5:     repeat
6:         tr ← max{t : t < tx.TS and Values[k, t] ≠ ⊥}
7:         for t = tr+1 to tx.TS do                ▷ read-lock interval [tr+1, tx.TS] if possible
8:             try to acquire read-lock for tx on (k, t), waiting
                   if timestamp is write-locked but not frozen
9:             if found frozen write-lock then release read-locks acquired above; break ▷ exit the
   "for" loop
10:    until found no frozen locks in the for loop
11:    return tr
12: function COMMIT-LOCKS(tx)
13:    if tx.TS = ∅ then return
14:    for (k, tr) ∈ tx.writeset do
15:        try to write-lock for tx on (k, tx.TS), waiting
             if a timestamp is read- or write-locked but not frozen
16:        if write-lock not acquired then tx.TS = ∅ and release all write locks for tx;
17: function COMMIT-TS(T) return tx.TS
18: function COMMIT-GC(tx) return true
```

Theorem 7. *The MVTL-Ghostbuster algorithm is not susceptible to ghost aborts.*

Proof sketch. MVTL-Ghostbuster chooses a timestamp at the beginning of the transaction, and it serializes transactions according to this timestamp. As in the MVTO algorithm, the only conflicts triggering aborts are read-write conflicts. If a transaction T_i aborts, it must have been because a write lock could not be acquired. This can only happen because a read lock already exists for $T_i.TS$ at the time of the write. If this is a ghost conflict, the lock must have been held by a transaction T_j that has finished its execution and aborted at the time of the conflict. But in real time, a transaction's commit method only finishes (with either an abort or commit result) after the GC function is called (in which function, if

[3] Ghost aborts are different from cascading aborts [26], which occur when a transaction reads uncommitted data.

the transaction aborts, all its locks are removed). It is worth noting that in this algorithm, garbage collection is always performed. Hence, a transaction that aborts only holds any locks while it is executing (i.e., while it is an active transaction). Therefore, a write cannot encounter a conflict due to a transaction that already aborted, and thus no ghost conflicts can appear using this algorithm. ◀

H Extending MVTL to distributed systems

For the distributed version of MVTL, we consider a standard distributed system model [5], with processes that communicate via message passing. The system is asynchronous: there are no bounds on the relative speed of processes or on communication. Processes have local clocks, with domain $\mathcal{T} = \{0, 1, \ldots\}$, which need not be synchronized. Unless explicitly stated otherwise, processes may exhibit crash-failures: they may stop executing unexpectedly. Where appropriate, we discuss other failure models as well. We assume the data is partitioned among multiple servers, and may or may not be replicated (we discuss both cases). Transactions are coordinated by the processes that want to execute them; we refer to such processes as clients or coordinators.

Algorithms 10 and 12 show the basic algorithm for the client and server respectively, while Algorithm 11 shows a generic policy. The policy is specified by the transaction coordinator, and it is applied by the server.

This generic algorithm leads to specific algorithms with high communication efficiency: only one round-trip to each object in the read set and two round-trips to each object in the write set. This efficiency is possible when the policy does not require garbage collection and its fault tolerance mechanism does not send messages when the coordinator is unsuspected (we discuss when this is viable in Section H.1).

Relative to the centralized MVTL algorithm, the main technical challenge addressed by the distributed MVTL algorithm is handling failures. A transaction coordinator failure may leave write locks in an unfrozen state indefinitely, causing other transactions to block forever. A server failure similarly causes either indefinite waiting from transaction coordinators or failure of all transactions accessing the failed server.

The solution to both types of failure is simple: we associate a *commitment* object with each transaction, to ensure that everyone agrees on whether the transaction committed or aborted. Technically, the commitment object solves consensus: it ensures that (1) no two processes obtain different decisions, (2) the only possible decisions are *abort* or *commit*(t) where t is a timestamp, (3) if the decision is d, some participant proposed d, (4) each correct process eventually decides, and decides only once.

After a coordinator has acquired all the necessary locks and has found a commit timestamp, it proposes the *commit* outcome with the associated timestamp to the commitment object of the transaction. If the decision is to commit, the coordinator then proceeds to inform the servers in the write set of the commit timestamp, without waiting for replies, allowing them to freeze the write locks associated with this transaction (we note that the protocol would be correct even without this step; we include it for performance). When the servers receive a request to freeze the locks of a transaction, they also propose *commit* with the received timestamp from the coordinator. This is because from the point of view of a server, when it has received the serialization timestamp of a transaction, the transaction is committed. However, if a server has held unfrozen write locks for a certain amount of time without receiving the freeze message from the coordinator, it will assume the coordinator has failed and it will propose an *abort* outcome to the commitment object. If the decision is to commit,

Algorithm 10 The generic distributed MVTL algorithm

```
1: function BEGIN(tx)
2:     tx.readset ← ∅; tx.writeset ← ∅; tx.committs ← ⊥
3: function WRITE(tx, k, v)                          ▷ write v to k in transaction tx
4:     status ← WRITE-LOCKS(tx, k, v)                ▷ write lock some subset of timestamps
5:     if status = abort then
6:         decision ← tx.commitment.tryAbort();      ▷ decision must be abort in this case
7:         mark tx as aborted
8:         return
9:     add (k, v) to tx.writeset                     ▷ remember key and value we wrote
10: function READ(tx, k)                             ▷ read k in transaction tx
11:    (tr, V) ← READ-LOCKS(tx, k)   ▷ read lock some interval [tr+1, ...] with Values[k, tr] ≠ ⊥
12:    if tr = ⊥ then return ⊥                       ▷ read failed
13:    add (k, tr) to tx.readset                     ▷ remember key and version we read
14:    return V                                      ▷ return committed value
15: function COMMIT(tx)                              ▷ try to commit transaction tx
16:    COMMIT-LOCKS(tx)                              ▷ locks to acquire at commit time
17:    T ← {t : ∀k ∈ tx.readset.keys, tx has a lock on (k, t) and ▷ try to find a locked timestamp for
       tx
                ∀k ∈ tx.writeset.keys, tx has a write-lock on (k, t)}
18:    if T = ∅ then
19:        decision ← tx.commitment.tryAbort();      ▷ decision must be abort in this case
20:        mark tx as aborted
21:    else
22:        tx.committs ← COMMIT-TS(T)                ▷ pick some timestamp in T
23:        decision ← tx.commitment.tryCommit(tx.committs);
24:        if decision = abort then
25:            mark tx as aborted
26:        else
27:            for (k, v) ∈ tx.writeset do
28:                send(server(k), freeze-write-lock, k, tx.committs)   ▷ freeze locks
29:    if COMMIT-GC(tx) then GC(tx)                  ▷ invoke gc or not
30: function GC(tx)                                  ▷ garbage collect locks of tx after it ended
31:    if tx committed then
32:        for (k, tr) ∈ tx.readset do
33:            send(server(k), freeze-read-locks, k, [tr+1, tx.committs])
34:    send messages to release all unfrozen read- and write-locks for tx
```

Algorithm 11 Client policy for the generic distributed MVTL algorithm

```
1: function WRITE-LOCKS(tx, k, v)
2:     send(server(k), (tx, write-locks, k, v, T)), for some set T
3:     wait_message(server(k), status, T′)   ▷ T′ subset of T for which write locks acquired
4:     return status
5: function READ-LOCKS(tx, k)                        ▷ returns a timestamp or ⊥
6:     send(server(k), (tx, read-lock, k, T, criteria)), for some set T
7:     wait_message(server(k), tr, te, V)   ▷ [tr + 1, te] read locked if tr ≠ ⊥, V read value
8:     either return (tr,V) or return (⊥, bot)
9: function COMMIT-LOCKS(tx)
10:    acquire read- or write-locks for tx on some keys and timestamps as above
11: function COMMIT-TS(T) return some t ∈ T
12: function COMMIT-GC(tx) either return true or return false
```

Algorithm 12 The server

```
1: function RECEIVE-WRITE-LOCK-MESSAGE(tx, k, v, T)
2:     acquire write-locks for tx on (k, T') for T' ∈ T in which acquiring locks is possible
3:     tx.pending_value(k) ← v;                          ▷ remember v as new value
4:     send(client(tx), write-locks-acquired, T')
5: function RECEIVE-READ-LOCK-MESSAGE(tx, k, T, criteria)
6:     acquire read-locks for tx on (k, I) for I = [tr+1, te] where te ∈ T chosen according to criteria
   and Values[k, tr] ≠ ⊥
7:     send(client(tx), read-locks-acquired, tr, te or ⊥, Values[k, tr] or ⊥)
8: function RECEIVE-FREEZE-WRITE-LOCK-MESSAGE(tx, k, t)
9:     decision ← tx.commitment.tryCommit(t)
10:    if decision = abort then
11:        release tx's write locks
12:        return
13:    freeze write-lock for tx on (k, t)                ▷ freeze locks
14:    Values[k, t] ← tx.pending_value(k)                ▷ expose committed value
15:    send(client(tx), write-locks-frozen, k)
16: function RECEIVE-FREEZE-READ-LOCK-MESSAGE(tx, k, [start, commit])
17:    freeze read-locks for tx on k for [start, commit]
18:    send(client(tx), read-locks-frozen, k)
19: function WRITE-LOCK-TIMEOUT(tx)
20:    decision ← tx.commitment.tryAbort()
21:    if decision = "commit @ t" then
22:        freeze write-lock for tx on (k, t)            ▷ freeze locks
23:        Values[k, t] ← tx.pending_value(k)            ▷ expose committed value
24:        send(client(tx), write-locks-frozen, k)
25:    else                                              ▷ decision = abort
26:        release tx's write locks
```

the server will receive a timestamp along with the decision and will be able to simply freeze its write locks at that timestamp and consider the transaction committed. The server can make this assumption because a commit decision is only possible if someone proposed *commit*, and a commit proposal only happens after the coordinator has performed all its updates and has found a commit timestamp, or after the coordinator has already informed a server of the commit timestamp. In both these instances the transaction can be committed. In the eventuality of an *abort* decision from the commitment object, a server releases all the write locks associated with that transaction and considers it aborted.

H.1 Commitment object implementations

This general mechanism used in our protocol allows various commitment object implementations, depending on the failure model we assume. If the coordinator or any minority of servers may fail, a Paxos-like consensus protocol could be used, with all the servers in the system as participants. This is because no server knows the write set of transactions, which can change dynamically with the execution.

However, in practice, storage servers are often replicated and their failures are masked, to provide both availability and durability of data. In this case, we can consider the storage server as a logical entity that does not fail, and consider only failures of the coordinator. By doing so, we can obtain an efficient implementation of commitment, one that requires little communication in the common failure-free case. We do so by implementing the commitment

object using Terminating Reliable Broadcast (TRB) [13], as we now explain.

Essentially, the coordinator designates a single server per transaction as the *decision point*. This server can be, for example, the first server accessed by a write operation. Consensus on the outcome of the reliable broadcast (i.e., whether the source has delivered or has crashed) is achieved on this decision server. Servers accessed on subsequent writes will then be informed of the decision point of the transaction. When the coordinator proposes a value to the commitment object, it needs to inform the decision point of this proposal and wait for its decision. When proposing a commit, the message can also act as a *freeze-write-locks* message to the decision server, which is only applied if the decision is to commit. If the coordinator has proposed *abort*, no other outcome is possible (since without receiving a *commit* message from the coordinator, servers themselves can only propose *abort*). However, in case of a commit proposal, the outcome may be that of *abort*: when write locks have been acquired for a certain amount of time, but have not been frozen, the servers suspect the coordinator of having failed, and thus propose *abort*. The *abort* proposal from a server is similar to that of the coordinator: the decision point is contacted, and its decision is followed. If the coordinator's commit proposal has been executed at the decision point earlier than any *abort* proposal, the decision will be to commit, and the decision server sends the commit timestamp along with the decision. A server proposes commit only once it has received the *freeze-write-locks* message. But this message is only sent by the coordinator if the decision has been to commit. Hence, the commit proposal that a server does when freezing write locks can be executed entirely locally. If the server has not been informed of a transaction abort, it simply stores the *commit* decision locally. Thus, in the common, failure-free case, the coordinator does not need to exchange extra messages to be able to ensure fault tolerance.

H.2 Correctness

We start by proving the following lemma concerning the outcome of a transaction:

▶ **Lemma 8.** *In Algorithms 10 and 12, with the generic policy in Algorithm 11, if a participant considers a transaction as committed, no other participant considers it as aborted.*

Proof. The *commit* object associated with each transaction provides the standard properties of uniform consensus:

- *(Termination.)* Every correct process eventually decides some value.
- *(Validity.)* If a process decides v, then v was proposed by some process (and, as previously mentioned, v can only be *abort* or *commit*).
- *(Integrity.)* No process decides twice.
- *(Agreement.)* No two processes decide differently.

Each process, be it the coordinator or a server, uses the commit object in order to obtain the decision as to whether the transaction should be committed or aborted. Before this, it makes no assumptions about the state of a transaction. At commit time, the coordinator proposes *abort* if no serialization point was found, and commit otherwise. A server that times out proposes *abort*, and a server that receives a freeze write lock message (essentially a commit message) proposes *commit*. By the agreement property, all the processes involved with a particular transaction obtain the same outcome of the transaction.

◀

Using Lemma 1, we now prove the following theorem:

▶ **Theorem 9.** *Algorithms 10 and 12 with the generic policy in Algorithm 11 ensures serializability.*

The proof is largely similar to the centralized version, but we recall it here for completeness.

Proof. We denote by $T.committs$ the timestamp at which transaction T is serialized and commits (aborted transactions do not have a serialization timestamp). Each transaction has a unique serialization timestamp, as explained in Section 4.1. If a transaction T commits at a timestamp $T.committs$, then it holds write locks at $T.committs$ for all the data in its write set, and read locks from the largest timestamp smaller than $T.committs$ containing a committed value to $T.committs$ for all the data in its read set (Algorithm 10, line 17). By Lemma 1, if the coordinator considers a transaction to be committed, no server can consider it to be aborted, and thus the locks of the transaction must still be held. We denote by $r_i[x_j]$ the fact that transaction T_i has read a version of object x written by transaction T_j (i.e., the read operation has returned $Values[x, T_j.committs]$). In addition, we denote by $w_k[x_k]$ the fact that transaction T_k has written a new version of object x (i.e., it has written a value to $Values[x, T_k.committs]$).

We assume the serialization order is given by the commit timestamp of the transaction. That is, if transaction T_1 creates version v_1 of object o, and transaction T_2 creates version v_2 of object o, we say $v_1 \ll v_2$ iff $T_1.committs < T_2.committs$.

Let H be a multiversion history over a set of transactions $\{T_0, \ldots, T_n\}$, and $C(H)$ the committed projection of this history. The committed projection of an operation history retains only the operations that belong to committed transactions. A multiversion serialization graph (MVSG) has the transactions $\{T_0, \ldots, T_n\} \in C(H)$ as vertices and edges (1) from T_i to T_j if T_j reads from T_i, and (2) for $r_k[x_j]$ and $w_i[x_i] \in C(H)$, if $x_i \ll x_j$, then the graph has an edge from T_i to T_j, otherwise it has an edge from T_k to T_i.

It has been shown [4] that if the multiversion serialization graph is acyclic, then a multiversion history is *one copy serializable*, that is, equivalent to a serial one version history.

Similarly to the proof of the original multiversion timestamp order Algorithm, we show the MVSG resulting from MVTL is acyclic by showing that if an edge between T_i and T_j exists in the graph, $T_i.committs < T_j.committs$. We consider the types of edges that can appear in a multiversion serialization graph. The first type of edges are *reads-from edges*. In this case, transaction T_j reads a version written by transaction T_i. Function READ-LOCKS acquires locks for timestamps starting immediately after the timestamp containing the version whose value is returned (and, since it read-locks an interval of timestamps, does not lock timestamps equal or larger to later versions). Hence, the read can only be serialized at a timestamp higher than that at which the read version was created. Thus, $T_i.committs \leq T_j.committs$. The second type of edge appears if $r_k[x_j]$ and $w_i[x_i]$ are in H and $x_i \ll x_j$. In this case, an edge from T_i to T_j exists in the graph. By definition of $\ll$, $x_i \ll x_j$ iff $T_i.committs < T_j.committs$. Finally, the third type of edge appears if $r_k[x_j]$ and $w_i[x_i]$ are in H and $x_j \ll x_i$. In this case, an edge from T_k to T_i is created (this assumes $k \neq i$). Since $x_j \ll x_i$, we know that $T_j.committs < T_i.committs$. Given that T_k has performed a read of version x_j, T_k has necessarily applied read locks for each timestamp from $T_j.committs + 1$ to $T_k.committs$. A read lock can only be acquired if no write lock from another transaction is present. Similarly, a write lock on a timestamp cannot be acquired if a read lock from another transaction is present. Thus, $w_i[x_i]$ could not have occurred in the interval $[T_j.committs + 1, T_k.committs]$. And since we know $T_j.committs < T_i.committs$, $w_i[x_i]$ must have necessarily occurred after the interval. Thus, $T_k.committs < T_i.committs$. Given that all the edges in the graph are from transactions with lower serialization timestamps to transactions with higher serialization timestamps, a cycle cannot exist. Thus, H is one-copy serializable. ◀

We now focus on the liveness guarantees of the protocol.

▶ **Lemma 10.** *If the coordinator does not propose commit for a transaction, no server does either.*

Proof sketch. Servers only propose commit when receiving a freeze write locks request from the coordinator. However, the coordinator only sends these messages once it has proposed to commit the transaction and has received a positive decision. Hence, the servers cannot propose a commit before the coordinator does.

◀

▶ **Lemma 11.** *If a coordinator that has obtained write locks but has not committed fails, it is eventually suspected by every correct server that holds unfrozen write locks for the coordinator's ongoing transaction.*

Proof sketch. The proof is straight-forward, as every server holding unfrozen write locks suspects the coordinator after a certain (finite) amount of time has passed since the locks were acquired (and the *Write-Lock-Timeout* function is called).

◀

▶ **Lemma 12.** *If a coordinator fails before committing a transaction, its write locks are eventually released and the transaction aborted on the correct servers.*

Proof sketch. A transaction is effectively committed when the coordinator proposes *commit* to the commitment object corresponding to the transaction, and obtains the same decision. If a coordinator fails before proposing *commit*, according to Lemma 2, no-one proposes *commit* for its transaction. Therefore, a *commit* decision cannot be reached (according to the Validity property of the commitment object). According to Lemma 3, every server currently holding unfrozen write locks for the coordinator's ongoing transaction at the time of failure suspects it to have failed. In Algorithm 12, when a server suspects a coordinator (when a time-out for the unfrozen write-locks occurs), it proposes *abort*. Thus, since *commit* cannot be proposed, and every server holding write locks must propose *abort*, the only decision that can be reached is to abort.

◀

We now prove the following theorem:

▶ **Theorem 13.** *No transaction initiated by a correct coordinator is indefinitely delayed by a failed coordinator.*

Proof sketch. Indefinite delays can happen in one scenario: when unfrozen write locks are held for an object. A read at a higher timestamp (no matter how high) whose result would depend on whether or not a new version of the object is created at the timestamps that are currently write-locked but not frozen has no choice but to wait. Other operations may be affected by the ongoing transaction of a failed coordinator, but this does not result in waiting, but rather in aborting, and potentially retrying at a higher timestamp, where the two transactions would not interfere. According to Lemma 4, either a coordinator is correct, and thus eventually commits or aborts its transaction, or its write locks are eventually released. Therefore, it cannot be the case that a transaction initiated by a correct coordinator is delayed indefinitely by a failed coordinator.

◀

▶ **Theorem 14.** *Unless at least one server suspects the coordinator to have failed, a transaction that has chosen a serialization timestamp eventually commits.*

Proof sketch. Servers only propose *abort* when suspecting the coordinator to have failed (i.e., unfrozen write locks have been held for too long). Thus, if the coordinator is not suspected, no server proposes *abort*. Thus, the only proposal servers can make in this scenario is to commit. Additionally, if the coordinator has found a serialization timestamp for the transaction, then it must propose *commit*. Thus, since no-one in the system proposes *abort*, and the coordinator must propose *commit*, the final decision of the commitment object must be to commit.

◀