Space Bounds for Reliable Multi-Writer Data Store: Inherent Cost of Read/Write Primitives

Gregory Chockler
Royal Holloway, University of London
Gregory.Chockler@rhul.ac.uk

Dan Dobre
NEC Labs Europe
dan.dobre@neclab.eu

Alexander Shraer
Google, Inc.
shralex@google.com

Alexander Spiegelman
Technion
sashas@tx.technion.ac.il

Abstract

Reliable storage emulations from fault-prone components have established themselves as an algorithmic foundation of modern storage services and applications. Most existing reliable storage emulations are built from storage services supporting arbitrary read-modify-write primitives. Since such primitives are not typically exposed by pre-existing or off-the-shelf components (such as cloud storage services or network-attached disks) it is natural to ask if they are indeed essential for efficient storage emulations. In this paper, we answer this question in the affirmative. We show that relaxing the underlying storage to only support read/write operations leads to a linear blow-up in the emulation space requirements. We also show that the space complexity is not adaptive to concurrency, which implies that the storage cannot be reliably reclaimed even in sequential runs. On a positive side, we show that Compare-and-Swap primitives, which are commonly available with many off-the-shelf storage services, can be used to emulate a reliable multi-writer atomic register with constant storage and adaptive time complexity.

Corresponding author:
Gregory Chockler
Royal Holloway, University of London
Egham TW11 0RP United Kingdom
tel: +44 (0)1784 443690
Gregory.Chockler@rhul.ac.uk

Regular submission

Eligible to be considered for the best student paper award: Alexander Spiegelman is a full-time student

1 Introduction

Reliable storage emulations seek to construct fault-tolerant shared primitives, such as read/write registers, from a collection of failure-prone components, such as storage servers, or network-attached disks. These emulations are core enablers of many modern storage services and applications, such as cloud and online data stores [1, 2, 3, 4, 5] and Storage-as-a-Service offerings [6, 7, 8, 9].

Most existing emulation algorithms are constructed from storage services capable of supporting custom-built read-modify-write (RMW) primitives [10, 11, 11, 12, 13, 14, 15]. For example, the ABD algorithm [10], emulating a fault-tolerant atomic read/write register from crash-prone nodes, assumes that each node has an ability to test and update the stored data along with its associated metadata in a single atomic step. In reality though reliable storage services must often be built from pre-existing or off-the-shelf building blocks (such as network-attached disks or cloud storage services), which typically offer a set collection of read/write capabilities sometimes augmented with simple conditional update primitives similar to Compare-and-Swap (CAS).

In this paper, we study the question of what *minimal* functionality must be supported by fault-prone storage nodes to enable space-efficient emulations of reliable storage primitives. We start by considering storage servers equipped with read/write primitives, which we abstract as read/write atomic registers. A notable prior work assuming a similar setting is Disk Paxos [16], which builds a reliable consensus service from crash-prone network attached disks. Interestingly, in Disk Paxos, each client is allocated a dedicated register on each server, which naturally leads to the question if linear space is necessary for constructing reliable multi-writer storage from fault-prone read/write primitives.

In Section 3, we prove that this is indeed inherent: the number of registers required to implement a reliable multi-writer read/write register for k clients from a collection of multi-writer multi-reader (MWMR) atomic read/write registers hosted on crash-prone servers requires at least kf registers where f is the maximum number of tolerated server failures. We further show that no such algorithm can have its storage consumption adaptive to concurrency, which implies that the storage costs cannot be further optimized (e.g., by reclaiming old values) even in sequential runs. Since the registers can be assigned to the servers in a variety of ways, we further restrict possible assignments by showing that if the number of registers per server is bounded by a known constant m, then supporting ℓm clients requires $f + 1$ more servers in addition to the requisite ℓf servers stipulated by our storage bound. Our bounds apply to any fault-tolerant implementations of a MWMR register, which are at least *single-writers safe* (a consistency notion weaker than the standard multi-writer safety [17, 18]), and solo-terminating (a weak liveness condition where only the operations eventually run in isolation are required to terminate).

We prove our results in a fault-prone shared memory model [19, 20, 21], which faithfully captures the settings where constituent storage services are provided as pre-existing building blocks. Our impossibility proofs employ a variation of a covering argument [22] to construct a sequential run where f new registers become covered with each consecutive write invoked by a client thus gradually exhausting the available storage capacity.

Understanding the cost of using read/write primitives, we turn our attention to identifying a simple RMW primitive that can be used to efficiently support a reliable emulation. We focus on Compare-and-Swap (CAS), which closely matches a variety of conditional write primitives available with many of the today's cloud storage service interfaces [1, 7, 8, 9, 5]. In Section 4, we present a constant space emulation of a MWMR atomic read/write register that utilizes a single CAS object per server, and tolerates up to a minority of server crashes. Our emulation is derived in a modular fashion by first constructing the ABD update primitive from a single CAS object, and then plugging the resulting construction into the multi-writer

ABD emulation [10, 12]. We show that the time complexity our implementation matches that of ABD in contention-free runs, and, at the worst case, is adaptive to the number of concurrently executing clients.

2 Preliminaries

2.1 Model

We consider an *asynchronous fault-prone shared memory system* [19] consisting of a set of *base* objects $\mathcal{B} = \{b_1, b_2, \dots\}$. The objects are accessed by *clients* from some set $\mathcal{C} = \{c_1, c_2, \dots\}$. The clients interact with base objects via a set of operations supported by the objects. We will consider base objects supporting either simple *read* and *write* (i.e., read/write registers) or *compare-and-swap (CAS)* operations.

We consider a slight generalization of the model in [19] where the objects are mapped to a set $\mathcal{S} = \{s_1, s_2, \dots\}$ of servers via a function δ from $\mathcal{B}$ to $\mathcal{S}$. For $B \subseteq \mathcal{B}$, we will write $\delta(B)$ to denote the *image* of B, i.e., $\delta(B) = \{\delta(b) : b \in B\}$. Conversely, for $S \subseteq \mathcal{S}$, we will write $\delta^{-1}(S)$ to denote the *pre-image* of S, i.e., $\delta^{-1}(S) = \{b : \delta(b) \in S\}$. Both servers and clients can fail by crashing. A crash of a server causes all objects mapped to that server to instantaneously crash[1].

We study algorithms that emulate shared read/write registers to a set of clients. Clients interact with the emulated register via high-level read and write operations. To distinguish the high-level emulated reads and writes from low-level base object access, we refer to the former as READ and WRITE. We say that high-level operations are *invoked* and *return* whereas low-level operations are *triggered* and *respond*. A high-level operation consists of a series of trigger and respond actions on base objects, starting with the operation's invocation and ending with its return. Since base objects are crash-prone, clients must be able to continue executing without awaiting responses to previously issued operations. Thus, the trigger actions occur locally at clients without involving any actual interaction with their target base objects. Once triggered a low-level operation can then *take effect* (or, be *applied* to) the base object state followed by a response being returned to the client.

An algorithm A defines the behavior of clients as deterministic state machines where state transitions are associated with actions, such as trigger/response of low-level operations. A *configuration* is a mapping to states from system components, i.e., clients and base objects. An *initial configuration* is one where all components are in their initial states.

A *run* of algorithm A is a (finite or infinite) sequence of alternating configurations and actions, beginning with some initial configuration, such that configuration transitions occur according to A. We use the notion of time t during a run r to refer to the configuration reached after the t^{th} action in r. A *run fragment* is a contiguous sub-sequence of a run. A run is *write-only* if it has no invocations of the high-level read operations.

We say that a base object, client, or server is *faulty* in a run r if it fails at some time in r, and correct, otherwise. A run is *fair* if (1) for every low-level operation triggered by a correct client on a correct base object, there is eventually a matching response, and (2) every correct client gets infinitely many opportunities to both trigger a low-level operation and execute the return actions. We say that a low-lever operation on a base object is *pending* in run r if it was triggered but has no matching response in r.

We say that a high-level operation op_i *precedes* a high-level operation op_j in a run r, denoted $op_i \prec_r op_j$, if op_i returns before op_j is invoked in r. Operations op_i and op_j are concurrent in a run r, if neither one precedes the other. A run with no concurrent operations is *sequential*.

[1] Note that the original faulty shared model of [19] can be derived from our model by choosing δ to be an injective function.

2.2 Storage Service Definitions

We study storage services emulating a *multi-writer/multi-reader (MWMR) register*, which stores values from a domain $\mathbb{V}$, and offers an interface for invoking read and write operations. Initially, the register holds some distinguished initial value $v_0 \in \mathbb{V}$. The sequential specification of the register is as follows: A read returns the latest written value, or v_0 if none was written.

Liveness We consider the following liveness conditions that must be satisfied in fair runs of an emulation algorithm. A *wait-free* object is one that guarantees that every high-level operation invoked by a correct client eventually returns, regardless of the actions of other clients. A *solo-terminating* object guarantees that every high-level operation that takes steps in isolation eventually returns.

Safety Two runs are *equivalent* if every client performs the same sequence of high-level operations in both, where operations that are pending in one can be either included (with some response) in or excluded from the other. A *linearization* of a run r is an equivalent sequential run that satisfies r's operation precedence relation and the object's sequential specification.

We consider the following safety requirements for an emulation algorithm. A run of the emulation algorithm satisfies *atomicity* if it has a linearization. An emulated object is *atomic* (or, *linearizable*) if all its runs satisfy atomicity. For our storage lower bound, we will also consider the following weak safety guarantee: A run r of the MWMR emulation algorithm is *single-writers* if no two write operations overlap in r: i.e., for any two distinct writes w_i and w_j in r either $w_i \prec_r w_j$ or $w_j \prec_r w_i$. A run r of the MWMR register emulation algorithm satisfies *safety* [17] if for every read rd that returns in r and does not overlap any writes, there exists a linearization L_{rd} of the subsequence of r consisting of all write operations in r and rd. An emulated MWMR register is *single-writers safe (SW-safe)* if all its single-writers runs satisfy safety.

For our space lower bound, we will restrict our attention to *single-reader (SR)* emulations where only a single designated client is allowed to read the emulated register.

Fault-Tolerance The emulation algorithm is f-tolerant if it remains correct (in the sense of its safety and liveness properties) as long as at most f servers crash for a fixed $f > 0$.

Complexity measures The *resource consumption* of an emulation algorithm A in a (finite) run r is the number of base objects used by A in r. The *resource complexity* [19] of A is the maximum resource consumption of A in all its runs. To measure running time, we assume that each operation triggered on a base object takes at most one unit of time to complete, and the local computation delays are negligibly small. The *(asynchronous) time complexity* of A [23] is then the maximum time required by any client to complete the high-level object invocation.

Adaptivity to Contention Given a run fragment r of an emulation algorithm, the *point contention* [24, 25] of r, PntCont(r), is the maximum number of clients that have an incomplete high-level invocation after some finite prefix of r. Similarly, we use PntCont(*op*) to denote PntCont(r_{op}), where r_{op} is the run fragment including all events between the *op*'s invocation and response.

The resource complexity of A is *adaptive to point contention* if there exists a function M such that after all finite runs r of A, the resource consumption of A in r is bounded by $M(\text{PntCont}(r))$. Likewise, the time complexity of A is *adaptive to point contention* if there exists a function T such that for each client c_i, and operation *op*, the time to complete the invocation of *op* by c_i is bounded by $T(\text{PntCont}(op))$.

3 Resource Complexity of Emulating SW-Safe MWSR Register

In this section, we prove that any f-tolerant emulation of a solo-terminating multi-writer/single-reader (MWSR) SW-safe register for k clients from of a collection of MWMR atomic registers stored on crash-prone servers has resource complexity kf. As there are many possible ways in which these kf registers can be mapped to the given set of servers, we further restrict possible mappings by showing that if the number of registers assigned to each server is at most m, then for any $\ell > 0$, the number of servers required to support ℓm clients is at least $\ell f + f + 1$. In other words, supporting that many clients requires extra $f + 1$ servers in addition to ℓf stipulated by our resource complexity bound. For completeness, we will also show that $2f + 1$ servers are necessary regardless of the individual server capacities though this bound can also be derived from well-known results (e.g., [26, 27]). Our last result shows that the emulation resource complexity cannot be adaptive to point contention.

Our proof exploits the fact that the environment is allowed to prevent a pending low-level write from taking effect on the base object states for arbitrary long. As a result, a client cannot reliably store a value in a base register having a pending write (by a different client) as this write may take effect at a later time thus erasing the stored value. We will reuse the terminology of [22], and refer to a pending write operation W on some base register b as a *covering write*, and to b as being *covered* by W.

For any time t (following the t^{th} action) in a run r of the emulation algorithm we define the following:

- $C(t)$: the set of clients that have completed a high-level write operation on the emulated register at time $\leq t$.
- $Cov(t)$: the set of the base registers that have a covering low-level write at time t.

We first prove the following key lemma:

Lemma 3.1 *For all $F \subseteq \mathcal{S}$ such that $|F| = f$, there exists a write-only sequential run r_i of an f-tolerant algorithm that emulates an SW-safe solo-terminating MWSR register consisting of $i \geq 0$ complete high-level writes of values $v_1, \ldots, v_i$ by i distinct clients $c_1, \ldots, c_i$, and t_i steps such that $|Cov(t_i)| \geq if$, and $\delta(Cov(t_i)) \cap F = \emptyset$.*

We construct r_i inductively as follows. First, it is easy to see that a run r_0 consisting of $t_0 = 0$ steps satisfies the lemma. Next, fix an arbitrary set of servers F such that $|F| = f$, and assume that r_{i-1} exists for all $i > 0$. We show how r_{i-1} can be extended up to time $t_i > t_{i-1}$ so that the lemma holds for the resulting run.

We introduce the following notation for all times $t \geq t_{i-1}$:

- $Tr_i(t)$: the set of base registers which had a low-level write triggered on between t_{i-1} and t.
- $Cov_i(t) = Cov(t) \setminus Cov(t_{i-1})$: the set of base registers that have been newly covered between t_{i-1} and t. Note that $Cov_i(t) \subseteq Tr_i(t)$.
- $Q_i(t) \subseteq \mathcal{S}$: the set of servers such that $Q_i(t) = \delta(Cov_i(t)) \setminus F$ if $|\delta(Cov_i(t)) \setminus F| \leq f$, and $Q_i(t) = Q_i(t-1)$, otherwise.

We will define the following adversarial behaviour of the environment, which whilst being tolerated by the algorithm causes it to consume a gradually growing amount of the storage resources:

Definition 3.2 **(Ad_i)** *: At any time $t \geq t_{i-1}$: prevent the following writes from taking effect on the base register states:*

1. *all covering writes by clients in $C(t_{i-1})$, and*

2. *all covering writes on the base registers in $\delta^{-1}(Q_i(t))$.*

Observation 1 *If the environment behaves like Ad_i, then for all $t \geq t_{i-1}$, $Q_i(t) \subseteq Q_i(t+1)$.*

We first show that r_{i-1} can be extended with a complete high-level write W_i by a new client c_i such that the environment behaves like Ad_i until W_i returns. Intuitively, this means that Ad_i delays applying low-level writes triggered by c_i on at most f servers as well as the past covering writes. As a result c_i cannot distinguish this scenario from the one where all the involved servers and clients have crashed, and therefore, by solo-termination, must return without before receiving the delayed replies.

Lemma 3.3 *Suppose that the environment behaves like Ad_i, and let W_i be a high-level write invocation by client $c_i \notin C(t_{i-1})$. Then, there exists time $t_r > t_{i-1}$ at which W_i returns while the environment continues to behave like Ad_i until t_r.*

Proof: By definition of Ad_i, there exists time $t_f > t_{i-1}$ such that for all times $t \geq t_f$, $Q_i(t) = Q_i(t_f)$. If W_i returns before t_f, then $t_r = t_f$ satisfies the lemma. Otherwise, for each server $s \in Q_i(t_f)$, let t_s be the earliest time such that $s \in Q_i(t_s)$. Since by Observation 1, $Q_i(t) \subseteq Q_i(t_f)$ for all $t \leq t_f$, $s \in Q_i(t)$, for all $t \geq t_s$.

Let r' be a fair run, which includes the same sequence of steps as r_{i-1} up to time t_f, and in addition, each server $s \in Q_i(t_f)$ fails immediately after the step t_s, and each client $c_1, \ldots, c_{i-1}$ fails before any of its covering writes on registers in $Cov(t_{i-1})$ takes effect on the register states. Since r' is fair, by f-tolerance and solo-termination, there exists time t' at which W_i returns in r'. Since r_{i-1} is indistinguishable from r' to c_i for the entire duration of W_i, it must return in r_{i-1} at time $t_r = t'$ as well. □

We next show that in order to guarantee correctness in the face of the environment behaving like Ad_i, W_i must trigger a low-level write on at least one non-covered base register on each server in a set of $2f+1$ servers.

Lemma 3.4 *Let W_i be a high-level write invocation by client $c_i \notin C(t_{i-1})$ that returns at time $t_r > t_{i-1}$, and suppose that the environment behaves like Ad_i until t_r. Then, $|\delta(Tr_i(t_r) \setminus Cov(t_{i-1}))| > 2f$.*

Proof: Denote $M \triangleq \delta(Tr_i(t_r) \setminus Cov(t_{i-1}))$, and assume by contradiction that $|M| \leq 2f$. Let $S_1 = M \cap F$, $S_2 = Q_i(t_r)$, and $S_3 = M \setminus (S_1 \cup S_2)$. Note that S_1, S_2, S_3 are pairwise disjoint, $M = S_1 \cup S_2 \cup S_3$, and by definition of $Q_i(t_r)$, and since $|F| = f$, $|S_1 \cup S_3| = |S_1| + |S_3| \leq f$.

Let r be a run, which is identical to r_{i-1} up to time t_{i-1}, after which all the covering writes in r_{i-1} take effect on register states, and all servers in the set $S_1 \cup S_3$ crash. Extend r with an invocation of a high-level read operation R by client $c_{rd} \neq c_i$. Since r is fair, by solo-termination and f-tolerance, there exists time $t_{rd} > t_{i-1}$ at which R returns. Since r is single-writers, by SW-safety, R must return v_{i-1}.

Let r' be a run, which is identical to r_{i-1} up to time t_r, after which it is extended to time $t' > t_r$ by having all servers in the set $S_1 \cup S_3$ crash, and the covering writes in r_{i-1} to take effect on the base register states. As a result, the values stored in the registers in $Cov(t_{i-1})$ are now identical to those in r. Furthermore, since Ad_i prevents all low-level writes triggered on registers in $\delta^{-1}(S_2)$ from taking effect before t_r, their values are also the same as those in r. Thus, at t', all registers in both r and r' have the same content.

We extend r' by having client $c_{rd} \neq c_i$ to invoke high-level read R while allowing the environment to continue preventing all covering writes by client c_i on the registers in $\delta^{-1}(S_2)$ from taking effect on their states. Since r' is indistinguishable from r to c_{rd}, the sequence of steps executed by c_{rd} in r' is the same as that in r. Hence, R returns v_{i-1} in r'. However, since W_i is the last complete write preceding R in r', by SW-safety, the R's return value must be $v_i \neq v_{i-1}$. A contradiction.

□

The following two corollaries follow immediately from Lemmas 3.3 and 3.4:

Corollary 3.5 *Let W_i be a high-level write invocation by client $c_i \notin C(t_{i-1})$ that returns at time $t_r > t_{i-1}$, and suppose that the environment behaves like Ad_i until t_r. Then, $Q_i(t_r) = f$.*

Corollary 3.6 *For all $i > 0$, $|\mathcal{S} \setminus \delta(Cov(t_{i-1}))| > 2f$.*

We are now ready to prove Lemma 3.1:

Proof: [of Lemma 3.1] By Lemma 3.3, r_{i-1} can be extended with a complete high-level write W_i by client $c_i \neq c_{i-1}$ writing a value $v_i \neq v_{i-1}$ while allowing the environment to behave like Ad_i until time t_r when W_i returns. We further extend r_{i-1} by allowing the environment to behave like Ad_i until time $t' > t_r$ when all writes triggered after t_{i-1} on the registers in $\delta^{-1}(F)$ take effect. Hence, $F \cap \delta(Cov_i(t')) = \emptyset$.

Since by Corollary 3.5, $Q_i(t_r) = f$, and by Observation 1, $Q_i(t_r) \subseteq Q_i(t')$, $Q_i(t') = f$, and therefore, $|Cov_i(t')| \geq f$. Now since $Cov_i(t')$ and $Cov(t_{i-1})$ are disjoint, $Cov(t') = Cov(t_{i-1}) \cup Cov_i(t')$, and by the induction hypothesis $|Cov(t_{i-1})| \geq (i-1)f$, and $\delta(Cov(t_{i-1})) \cap F = \emptyset$, we receive $|Cov(t')| \geq (i-1)f + f = if$, and $\delta(Cov(t')) \cap F = (\delta(Cov(t_{i-1})) \cap F) \cup (\delta(Cov_i(t')) \cap F) = \emptyset$. Thus, $t_i = t'$ satisfies the lemma. □

Resource Complexity The following theorem follows immediately from Lemma 3.1 (please see Section A of the Appendix for a full proof):

Theorem 3.7 *For any $k \geq 0$, $f \geq 0$, there is no f-tolerant algorithm emulating an SW-safe solo-terminating MWSR register for k clients using less than kf base registers.*

Number of Servers We now turn our attention to deriving the number of servers required for supporting the emulation. The following result follows immediately from Corollary 3.6 (please see Section A of the Appendix for a full proof), but can also be derived from well-known results in the literature (e.g., [26, 27])

Theorem 3.8 *For any $k > 0$, and $f \geq 0$, there is no f-tolerant algorithm emulating an SW-safe solo-terminating MWSR register for k clients with less than $2f + 1$ servers.*

Next, we show that if the storage per server is bounded by a known constant, an extra $f + 1$ servers beyond the minimum capacity established by Theorem 3.7 are necessary to accommodate a given number of clients.

Theorem 3.9 *For any $m > 0$, $\ell > 0$, and $f \geq 0$, there is no f-tolerant algorithm emulating an SW-safe solo-terminating MWSR register for $k \geq \ell m$ clients using less than $\ell f + f + 1$ servers if each server can store at most m registers.*

Proof: Assume by contradiction there exists an f-tolerant algorithm A emulating an SW-safe solo-terminating MWSR register for $k = \ell m$ clients using $\ell f + f$ servers. Fix a set $F \subseteq S$, such that $|F| = f$, and let $N \leq mf$ be the number of registers mapped to the servers in F.

By Lemma 3.1, there exists a run r_{k-1} of A consisting of $k-1 = \ell m - 1$ high-level writes by $k-1$ distinct clients such that by the end of r_{k-1}, the number of distinct base registers having a covering write is at least $(k-1)f$, and no registers in $\delta^{-1}(F)$ have a covering write. Thus, the number of registers that remain not covered by the end of r_{k-1} is at most $\ell fm + N - (k-1)f = \ell fm + N - \ell fm + f = N + f \triangleq R$.

Now since no register in $\delta^{-1}(F)$ has a covering write, N out of total R registers must be mapped to the f servers in F. And since the remaining f registers can be mapped to at most f servers, by the end of r_{k-1}, the total number of servers that may have a register without a covering write is at most $2f$. A contradiction to Corollary 3.6. □

Adaptivity We show that no SW-safe solo-terminating MWSR register can have a fault-tolerant emulation adaptive to point contention:

Theorem 3.10 *For any $f > 0$, there is no f-tolerant algorithm that emulates an SW-safe solo-terminating MWSR register with resource complexity adaptive to point contention.*

Proof: Pick an arbitrary $f > 0$, and assume by contradiction that such an algorithm A exists. By Lemma 3.1, there exists a run r of A consisting of k high-level writes by k distinct clients such that the resource complexity grows by f for each consecutive write that completes in r whereas the point contention remains equal 1 for the entire r. We conclude that no function mapping point contention to resource consumption can exist, and therefore, A's resource complexity is not adaptive to point contention. A contradiction. □

4 Atomic Register Implementation

In this section we present a space-efficient f-tolerant algorithm implementing a wait-free MWMR atomic register from a collection of $n > 2f$ servers each storing a single *CAS* object. Unlike previous space-efficient approaches our algorithm does not require support for any specialized read-modify-write functionality besides *CAS*, i.e., conditional write, obviating the need for a custom server code. The algorithm's time complexity is *adaptive* to concurrency guaranteeing that each operation *op* terminates in at most $O(c^2)$ steps where $c = \text{PntCont}(op)$.

Our algorithm, called *CAS-ABD*, is derived from the multi-writer ABD [10] emulation of an atomic read/write register to which we refer as *MW-ABD*. For completeness, the MW-ABD implementation is briefly reviewed in Section 4.1 below (full details can be found in [12]). The CAS-ABD algorithm is described in Section 4.2.

4.1 MW-ABD Algorithm

The MW-ABD shared state consists of a set $\mathcal{B}$ of $n > 2f$ crash-prone objects $\{b_1, \ldots, b_n\}$ mapped to a set $\mathcal{S}$ of n servers $\mathcal{S} = \{s_1, \ldots, s_n\}$ such that $\delta(b_i) = s_i$ for each $1 \leq i \leq n$. Each object b_i stores a pair (ts, val) where ts is a timestamp and $val \in \mathbb{V}$. We will write $b_i.ts$ and $b_i.val$ to refer to the timestamp and value components of b_i respectively. Each timestamp ts is a pair (num, c) where $num \in \mathbb{N}$ is a natural number, and $c \in \mathcal{C}$ is a client. We will write $ts.num$ and $ts.c$ to refer to the ts's first and second component respectively. The timestamps are ordered lexicographically so that $ts < ts'$ if $ts.num < ts'.num$, or $ts.num = ts'.num$ and $ts.c < ts.c'$. The MW-ABD types and shared states are summarized in Algorithm 1.

Algorithm 1 Types and States of MW-ABD and CAS-ABD

```
$TS = \mathbb{N} \times \mathcal{C}$, the set of timestapms with selectors $num$ and $c$
$TSVal = TS \times \mathbb{V}$, with selectors $ts$ and $val$
$\mathcal{B} = \{b_1, \ldots, b_n\}$: the set of shared objects such that $b_i \in TSVal$ for all $1 \leq i \leq n$; initially $b_i = ((0,0), v_0)$
```

The sequential specification supported by each object $b_i \in \mathcal{B}$ is shown in Algorithm 2. It consists of two atomic operations: $read$ and $update$. The read operation returns the current content of b_i (i.e., $(b_i.ts, b_i.val)$); and the $update$ operation is a read-modify-write (RMW) primitive comprised of atomically executed sequence of steps shown in lines 2–5 of Algorithm 2. We henceforth refer to the object type supporting the sequential specification in Algorithm 2 as *ABD Object (ABDO)*.

Algorithm 2 The ABDO sequential specification for each b_i, $1 \leq i \leq n$

```
1: operation $update(b_i, t, v)$
2:     if $b_i.ts < t$
3:         $b_i.ts \leftarrow t$
4:         $b_i.val \leftarrow v$
5:     return ack
6: end
7: operation $read(b_i)$
8:     return $(b_i.ts, b_i.val)$
9: end
```

The implementation of both write and read proceeds by invoking consecutive *rounds* of base object accesses. At each round, the client triggers operations on all base objects in parallel, and awaits responses from at least $n - f$ objects. The write implementation consists of two rounds. In the first round, the writer collects the set R of $(b_i.ts, b_i.val)$ pairs from $n - f$ objects by triggering $b_i.read$ on all objects $b_i \in \mathcal{B}$. The writer then determines a new timestamp ts' to be stored alongside the value v being written so that $ts'.num = \max\{num' : (num', *) \in R\} + 1$ and $ts'.c$ is the writer's identifier. This is followed by another round where the writer triggers $b_i.update(b_i, ts, v)$ on each base object b_i to replace its current content with (ts, v).

The first round of read is identical to that of write except that the set R is used to identify the value $v' \in \mathbb{V}$ having the highest timestamp ts' among the timestamp/value pairs in R. This is followed by another round where the reader invokes $b_i.update(b_i, ts', v')$ on each base object b_i to ensure (ts', v') is available from all sets of $n - f$ base objects. The reader then returns v'.

4.2 CAS-ABD Algorithm

Suppose that the base ABD objects in $\mathcal{B}$ are substituted with *Compare-and-Swap (CAS)* objects: i.e., the sequential specification of each $b_i \in \mathcal{B}$ consists of a single *CAS* primitive whose code is shown in lines 15–19 of Algorithm 3. We obtain an implementation of an f-tolerant MWMR atomic read/write register from a collection of $n > 2f$ *CAS* base objects, to which we refer as *CAS-ABD*, in a modular fashion by first constructing an ABDO from a *single CAS* base object b_i using the emulation algorithm in Algorithm 3, and then, plugging the resulting construction into the MW-ABD algorithm described above.

Algorithm 3 The ABDO emulation from a single *CAS* object b_i, $1 \leq i \leq n$

```
    Local variables:
        $exp \in TSVal$, initially $((0,0), v_0)$
 1: operation $update(b_i, t, v)$
 2:     $done \leftarrow false$
 3:     if $t > exp.ts$
 4:         repeat
 5:             $old \leftarrow CAS(b_i, exp, (t, v))$
 6:             if $old = exp \vee old.ts \geq t$
 7:                 $done \leftarrow true$
 8:             $exp \leftarrow old$
 9:         until $done \leftarrow true$
10:     return ack
11: end

12: operation $read(b_i)$
13:     return $CAS(b_i, exp, exp)$
14: end

15: operation $CAS(b_i, exp, new)$, $exp, new \in TSVal$
16:     $prev \leftarrow b_i$
17:     if $exp = b_i$
18:         $b_i \leftarrow new$
19:     return $prev$
20: end
```

In order to prove that CAS-ABD is a correct implementation of an f-tolerant wait-free MWMR read/write register, it suffices to show that the ABDO emulation in Algorithm 3 is a wait-free linearizable implementation of the ABD object. Below we show that this is indeed the case assuming that the following property, to which we henceforth refer as *timestamp uniqueness*, is satisfied in all runs r of ABDO: for all objects $b_i \in \mathcal{B}$, r includes at *most* one invocation of the form $update(b_i, ts, *)$. Given that linearizability is a composable property [28], and MW-ABD is known to satisfy timestamp-uniqueness in all runs, the correctness of CAS-ABD then follows from the correctness of MW-ABD [12].

To show linearizability [28], we first identify for each invocation of $update$ and $read$ in each possible run of the ABDO emulation, a single step within the operation execution, called a *linearization point* (i.e., a single step where the operation takes effect on the base object state), as follows: For each $read$ invocation, the linearization point is simply the return step in line 13. The linearization points for the $update$ invocations are assigned to either one of the following two steps: (1) if $update$ returns without entering the loop in lines 4–9, the condition test step in line 3 is the linearization point; and (2) if $update$ returns due to the condition in line 6 being true, then the *CAS* call in line 5 is the linearization point. The linearizability then follows from following lemma (proven in Section B of the Appendix), which asserts that the sequence obtained by shrinking each operation to occur atomically at its linearization point is a valid sequential run of ABDO.

Lemma 4.1 *Let r be a run of the ABDO emulation in Algorithm 3, and σ be a sequential run obtained from r by shrinking each $update$ and $read$ operation to occur at its linearization point. Then, σ is a sequential run of the ABD object in Algorithm 2.*

Since the $read$ implementation is obviously wait-free, we only need to argue wait freedom for the $update$ operations. To see this, observe that $t > exp.ts$ every time before *CAS* is called in line 5 (see Lemmas B.1 in Section B of the Appendix). Since $b_i.ts = exp.ts$ is a necessary condition for a successful *CAS* call, the value of b_i can only be changed when $t > b_i.ts$. Hence, the timestamps of the values stored in each b_i are non-decreasing (see Lemma B.2 in Section B of the Appendix). If $b_i.ts$ does not change between the consecutive iterations of the loop in lines 4–9, timestamp uniqueness implies that the next call to *CAS* will be successful and the loop terminates. Otherwise, the fact that the timestamps are non-decreasing implies that $b_i.ts$ is superseded by a higher timestamp. Since there are only finitely many timestamps lower than t, the loop will terminate no later than the value of $b_i.ts$ reaches or exceeds t. Thus, we have the following result (see Section B of the Appendix for the full proof):

Lemma 4.2 *The ABDO emulation in Algorithm 3 is wait-free provided all its runs satisfy timestamp uniqueness.*

Given that timestamp uniqueness holds in all runs of MW-ABD, we receive the following:

Theorem 4.3 *The CAS-ABD algorithm is an f-tolerant implementation of a wait-free MWMR atomic register.*

Time Complexity It is easy to see that in the absence of contention, the $update$ operation terminates in at most 2 rounds of the base object accesses. This can be further optimized if the clients keep a local copy of the most recent value read from each object b_i at the read round of CAS-ABD, and then use this value to initialize the expected value parameter exp of *CAS*. Thus, in the best case scenarios when the object replies are received in a timely fashion, and there is no contention, $update$ will terminate in just 1 round, thus achieving the 2 round complexity of MW-ABD overall.

In the presence of contention, the number of unsuccessful *CAS* calls executed within the update operation loop in lines 4–9 is bounded by the number of unique timestamps returned by the *CAS* calls that are smaller than the timestamp t supplied to the update. Given the way the timestamps are chosen by the algorithm, the number of such timestamps per each of the c concurrently executing clients is constant. However, since the num component of each timestamp can be shared by concurrently executing clients, the overall time complexity of update can be as high as c^2. In Section B of the Appendix, we prove that c is equal to the maximum number of clients that can execute concurrently with the update thus obtaining the following:

Theorem 4.4 *The CAS-ABD time complexity is* adaptive *to concurrency guaranteeing that each operation op terminates in at most $O(c^2)$ base object accesses where $c = \mathit{PntCont}(op)$.*

5 Conclusions and Future Work

We studied the resource complexity of emulating an f-tolerant read/write MWMR register from a collection of atomic MWMR registers stored on crash-prone servers. We established a number of lower bounds that apply to any fault-tolerant emulation of a MWMR register, which satisfies weak correctness guarantees: single-writers safety, and solo-termination. In particular, we proved that no such emulation can use fewer than kf registers to support $k > 0$ clients or have its storage consumption adaptive to concurrency. We also characterized possible allocations of registers to servers by showing that if the number of registers per server is bounded by a known constant m, then supporting ℓm clients requires $f + 1$ more servers in addition to the requisite ℓf servers implied by our storage bound.

In search for a simple RMW primitive that can be leveraged for obtaining a space-efficient implementation, we studied reliable storage emulations from crash-prone CAS objects. To this end, we presented a constant space emulation of an MWMR atomic read/write register that utilizes a single CAS object per server, tolerates up to a minority of server crashes, and has time complexity adaptive to point contention.

Our work leaves some questions open for future work. First, observe that ABD can be applied in a straightforward fashion to implement an MWMR wait-free atomic register from fault-prone registers by assigning each client to a dedicated set of $2f + 1$ registers stored on $2f + 1$ different servers. An interesting open question is then whether our lower bound can be further tightened to match this storage cost, or there are emulations that can achieve a tighter storage cost (e.g., by weakening their correctness guarantees). Second, the worst-case time complexity of our CAS-based ABD implementation is quadratic in point contention. It will be interesting to explore whether it can be further improved (e.g., by modifying the ABD timestamp selection mechanism), or this is an inherent limitation.

References

[1] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni, "Pnuts: Yahoo!'s hosted data serving platform," *Proc. VLDB Endow.*, vol. 1, no. 2, pp. 1277–1288, Aug. 2008. [Online]. Available: http://dl.acm.org/citation.cfm?id=1454159.1454167

[2] Riak, http://basho.com/riak.

[3] J. Rao, E. J. Shekita, and S. Tata, "Using paxos to build a scalable, consistent, and highly available datastore," *PVLDB*, vol. 4, no. 4, pp. 243–254, 2011.

[4] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed, "Zookeeper: wait-free coordination for internet-scale systems," in *USENIX ATC*, Berkeley, CA, USA, 2010.

[5] mongoDB, http://www.mongodb.org/.

[6] A. S. S. S. A. S3), http://aws.amazon.com/s3/.

[7] A. SimpleDB, http://aws.amazon.com/simpledb/.

[8] A. DynamoDB, http://aws.amazon.com/dynamodb/.

[9] M. A. Storage, http://www.windowsazure.com/en-us/manage/services/storage.

[10] H. Attiya, A. Bar-Noy, and D. Dolev, "Sharing memory robustly in message-passing systems," *J. ACM*, vol. 42, no. 1, pp. 124–142, Jan. 1995. [Online]. Available: http://doi.acm.org/10.1145/200836.200869

[11] B. Englert and A. A. Shvartsman, "Graceful quorum reconfiguration in a robust emulation of shared memory," in *International Conference on Distributed Computing Systems (ICDCS)*, 2000, pp. 454–463.

[12] S. Gilbert, N. A. Lynch, and A. A. Shvartsman, "Rambo: a robust, reconfigurable atomic memory service for dynamic networks," *Distributed Computing*, vol. 23, no. 4, pp. 225–272, 2010.

[13] P. Dutta, R. Guerraoui, R. R. Levy, and M. Vukolic, "Fast access to distributed atomic memory," *SIAM Journal on Computing*, vol. 39, no. 8, pp. 3752–3783, 2010.

[14] C. Georgiou, N. C. Nicolaou, and A. A. Shvartsman, "Fault-tolerant semifast implementations of atomic read/write registers," *Journal of Parallel Distributed Computing*, vol. 69, no. 1, pp. 62–79, 2009.

[15] M. K. Aguilera, I. Keidar, D. Malkhi, J.-P. Martin, and A. Shraer, "Reconfiguring replicated atomic storage: A tutorial," *Bulletin of the EATCS*, vol. 102, pp. 84–108, 2010.

[16] E. Gafni and L. Lamport, "Disk Paxos," *Distributed Computing*, vol. 16, no. 1, pp. 1–20, 2003.

[17] L. Lamport, "On interprocess communication. part ii: Algorithms," *Distributed Computing*, vol. 1, no. 2, pp. 86–101, 1986.

[18] C. Shao, E. Pierce, and J. L. Welch, "Multi-writer consistency conditions for shared memory objects," in *DISC 2003*, 2003, pp. 106–120.

[19] P. Jayanti, T. Chandra, , and S. Toueg, "Fault-tolerant wait-free shared objects," *Journal of the ACM*, vol. 45, no. 3, pp. 451–500, 1998.

[20] Y. Afek, M. Merritt, and G. Taubenfeld, "Benign failure models for shared memory," in *Distributed Algorithms*. Springer, 1993, pp. 69–83.

[21] I. Abraham, G. Chockler, I. Keidar, and D. Malkhi, "Byzantine disk Paxos: Optimal resilience with Byzantine shared memory," *Distributed Computing*, vol. 18, no. 5, pp. 387–408, 2006.

[22] J. E. Burns and N. A. Lynch, "Bounds on shared memory for mutual exclusion," *Inf. Comput.*, vol. 107, no. 2, pp. 171–184, 1993.

[23] H. Attiya and J. Welch, *Distributed Computing: Fundamentals, Simulations, and Advanced Topics*, 2nd ed. Wiley, 2004, ch. 2, pp. 13–14.

[24] Y. Afek, H. Attiya, A. Fouren, G. Stupp, and D. Touitou, "Long-lived renaming made adaptive," ser. PODC '99. New York, NY, USA: ACM, 1999, pp. 91–103. [Online]. Available: http://doi.acm.org/10.1145/301308.301335

[25] H. Attiya and A. Fouren, "Algorithms adapting to point contention," *J. ACM*, vol. 50, no. 4, pp. 444–468, Jul. 2003. [Online]. Available: http://doi.acm.org/10.1145/792538.792541

[26] H. Attiya, A. Bar-Noy, D. Dolev, D. Peleg, and R. Reischuk, "Renaming in an asynchronous environment," *J. ACM*, vol. 37, no. 3, pp. 524–548, Jul. 1990. [Online]. Available: http://doi.acm.org/10.1145/79147.79158

[27] G. Bracha and S. Toueg, "Asynchronous consensus and broadcast protocols," *J. ACM*, vol. 32, no. 4, pp. 824–840, Oct. 1985. [Online]. Available: http://doi.acm.org/10.1145/4221.214134

[28] M. Herlihy and J. M. Wing, "Linearizability: A correctness condition for concurrent objects," *ACM Trans. Program. Lang. Syst.*, vol. 12, no. 3, pp. 463–492, 1990.

A Space Lower Bounds

Theorem A.1 *For any $k \geq 0$, $f \geq 0$, there is no f-tolerant algorithm emulating an SW-safe solo-terminating MWSR register for k clients using less than kf base registers.*

Proof: Pick arbitrary $k \geq 0$, $f \geq 0$, and assume by contradiction that there exists an f-tolerant algorithm A that emulates an SW-safe solo terminating MWSR register for k clients with fewer than kf base registers. By Lemma 3.1, there exists a run r of A consisting of k high-level writes by k distinct clients such that by the end of r, the number of distinct base registers having a covering write is at least kf. Hence, A will require at least kf distinct base registers to support k clients. A contradiction. □

Theorem A.2 *For any $k > 0$, and $f \geq 0$, there is no f-tolerant algorithm emulating an SW-safe solo-terminating MWSR register for k clients with less than $2f + 1$ servers.*

Proof: Assume by contradiction that there exists an f-tolerant algorithm emulating an SW-safe solo-terminating MWSR register for $k > 0$ clients using $2f$ servers. By Corollary 3.6, there exists a run r_1 of A consisting of a single high-level write W_1 by a client c_1 such that $|\mathcal{S} \setminus \delta(Cov(t_0))| > 2f$ where $t_0 = 0$. Since no base registers are covered at t_0, $|\mathcal{S} \setminus \delta(Cov(t_0))| = |\mathcal{S}| > 2f$. However, by assumption, $|\mathcal{S}| = 2f$. A contradiction. □

B Correctness of CAS-ABD

We first argue that our emulation is a linearizable implementation of ABDO. The argument relies on the following auxiliary invariants.

Lemma B.1 *If line 5 is reached, then $t > exp.ts$.*

Proof: The proof is by induction on the number of iteration of the loop in lines 4–9. For the base case, note that line 3, $t > exp.ts$ is the necessary condition for entering the loop. Hence, the lemma holds first time line 5 is reached. Next, assume that the result is true for all iterations $k \geq 1$, and consider iteration $k + 1$. Since iteration $k + 1$ is reached, the condition in line 6 must be false at iteration k, that is, $old.ts < t$. By line 8, at the beginning of iteration $k + 1$, $exp = old$, and therefore, $exp.ts = old.ts < t$ as needed. □

We now show that $b_i.ts$ is non-decreasing:

Lemma B.2 *Let $b_i.ts_1$ and $b_i.ts_2$ be the values of $b_i.ts$ at times t_1 and t_2 respectively. If $t_1 < t_2$, then $b_i.ts_1 \leq b_i.ts_2$.*

Proof: Observe that $b_i.ts$ can only change as a result of a successful *CAS* invocation in line 5. The necessary condition for that to happen is $exp = b_i$ in line 5. By Lemma B.1, $t > exp.ts = b_i.ts$. Hence, the value of $b_i.ts$ is either left unchanged, or increases as needed. □

Next, we show linearizability:

Lemma B.3 *Let r be a run of the ABDO emulation in Algorithm 3, and σ be a sequential run obtained from r by shrinking each $update$ and $read$ operation to occur at its linearization point. Then, σ is a sequential run of the ABD object in Algorithm 2.*

Proof: Let $t_1, t_2, \ldots$ such that $t_i < t_{i+1}$, $i \geq 1$, denote the times at which the linearization points occur in r. The proof is by induction on t_i. For the base case, consider the first linearization point t_1. If t_1 is the linearization point of $read$, then its return value $((0, 0), v_0)$; and if t_1 is the linearization point of $update$, then its return value is ack. Since both return values are identical to those produced by the $read$ and $update$ of the ABD object if invoked at the initial state, the result holds.

Next, assume that the result is true for the first $k - 1$ linearization points, and consider the kth linearization point t_k. If t_k is the linearization point of $update$, then its return value is ack, which is consistent with the sequential specification of the ABD object.

Suppose that t_k is the linearization point of a read operation. Suppose that the linearization point t_{k-1} is associated with a read. Since for any value of exp, *CAS*(exp, exp) does not changes the content of b_i, the return value of read will be the same as that of the read linearized at t_{k-1}, which complies with the sequential specification of the ABD object.

Next, suppose that the operation linearized at t_{k-1} is an update operation $u = update(b_i, t, v)$ for some $t \in TS$ and $v \in \mathbb{V}$. Let x_j denote the value of variable x at time t_j. The sequential specification of the ABD object requires the read to return (t, v) if $t > b_i.ts_{k-2}$, and $b_{i,k-2}$, otherwise. We show that this is indeed the case.

First, suppose that $t > b_i.ts_{k-2}$. Since no linearization points occur between t_{k-2} and t_{k-1}, and b_i can only be changed at a linearization point, at line 3, $exp.ts_{k-1} \leq b_i.ts_{k-2} = b_i.ts_{k-1} < t$. Hence, linearization point t_{k-1} must occur at line 5. This means that *CAS* in line 5 is successful as otherwise *old*$.ts_{k-1} \geq t$ implies that *old*$.ts_{k-1} = b_i.ts_{k-1} = b_i.ts_{k-2} \geq t$ contradicting the assumption. Therefore, the linearization point t_{k-1} coincides with a successful *CAS* in line 5 so that $b_{i,k-1} = (t, v)$. Since no linearization points occur between t_{k-1} and t_k, and b_i can only be changed at a linearization point, $b_{i,k-1} = b_{i,k} = (t, v)$. Hence, the read will return (t, v) as needed.

Finally, suppose that $t \leq b_i.ts_{k-2}$. If $t \leq exp.ts$, then linearization point t_{k-1} occurs at line 3, and therefore, u returns without changing b_i. Hence, $b_{i,k-1} = b_{i,k-2}$. Suppose $t > exp.ts$, and consider the *CAS* invocation occurring at the first iteration of the loop in lines 4–9. Observe that this invocation must be unsuccessful as otherwise, $b_i.ts_{k-2} = exp.ts < t$ contradicting the assumption that $t \leq b_i.ts_{k-2}$. At the same time, *old*$.ts = b_i.ts_{k-2} \geq t$. Hence, the condition in line 6 is true, which implies that u leaves the loop without changing the value of $b_{i,k-2}$ at t_{k-1}. We conclude that $b_{i,k-1} = b_{i,k-2}$. Thus, the read will return $b_{i,k-2}$ as required. □

We next show that the ABDO emulation is wait-free if all its runs satisfy timestamp uniqueness.

Lemma B.4 *The ABDO emulation in Algorithm 3 is wait-free provided all its runs satisfy timestamp uniqueness.*

Proof: Since the read operation is obviously wait-free, we only need to show that the update operation is wait-free as well.

Consider an update invocation $u = update(b_i, t, v)$. If the condition in line 3 is false, then u returns, and we are done. Otherwise, let ts_j, $j \geq 1$, be the value of $b_i.ts$ before *CAS* is invoked at the jth iteration of the loop in lines 4–9.

At all iterations $j \geq 1$, if $ts_j \geq t$, then the condition in line 6 is true, and the loop terminates. Otherwise, by Lemma B.2 and timestamp-uniqueness, $ts_{j+1} > ts_j$. Since there are only finitely many timestamps between ts_1 and t, there exists an iteration where the condition in line 6 is satisfied, and the loop terminates. □

Given that timestamp uniqueness holds in all runs of MW-ABD, we receive the following:

Theorem B.5 *The CAS-ABD algorithm is an f-tolerant implementation of a wait-free MWMR atomic register.*

Lemma B.6 *Let op be an operation that invokes $update$ at time t and let op$'$ be another operation that starts at time $t' \geq t$. If k operations are invoked but do not complete before time t then $ts(op').num \geq ts(op).num - k - 1$*

Proof: Let op'' be the operation with the highest timestamp that returns before time t. By MW-ABD timestamp selection mechanism, $ts(op') \geq ts(op'')$. Therefore it is sufficient to prove that $ts(op'').num \geq ts(op).num - k - 1$. Suppose, for the purpose of contradiction, that $ts(op'').num = ts(op).num - k - 2$. Since every operation increments num by at most one and the timestamp of op is $ts(op)$, at least $k+1$ operations must be invoked before time t with timestamps strictly greater than $ts(op).num - k - 2$. At least one of these operations returns before time t by the statement of our Lemma. This is a contradiction since op'' was chosen to be the operation with the highest timestamp that returns before t. □

Lemma B.7 *Let op be an operation that invokes $update$ at time t and op$'$ be another operation that obstructs op on some object b_i but is not one of the first two operations to obstruct op on b_i. Then op$'$ does not complete by time t.*

Proof: Since op is obstructed at least three times, the following sequence of invocations on b_i must occur ($op.b_i.CAS$ denotes the invocation of CAS on register b_i during high-level operation op):
$op.b_i.CAS \ldots op.b_i.CAS \ldots op.b_i.CAS$. Since all three invocations of $op.b_i.CAS$ fail (the third one due to op'), we know that there are at least three invocations of $b_i.CAS$ by other operations that succeed: $op'''.b_i.CAS \ldots op.b_i.CAS \ldots op''.b_i.CAS \ldots op.v_i.CAS \ldots op'.b_i.CAS \ldots op.b_i.CAS$.

Since $op'.b_i.CAS$ succeeds updating b_i, it learns the value written by $op''.b_i.CAS$, which happens after the first invocation of $op.b_i.CAS$, which in turn must occur after update is invoked during op, i.e., after time t. Hence, op' does not complete by time t. □

Lemma B.8 *Let op be an operation. For any constant n the number of operations op$'$ that are concurrent with op and such that $ts(op').num = n$ is at most PntCont(op).*

Proof: Suppose for the sake of contradiction that there exists a constant n such that there are PntCont$(op)+1$ operations concurrent with op with the first component of their timestamp equal to n. Since there are PntCont$(op)+1$ operations and at most PntCont(op) clients executing operations concurrently with op at any single point in time (by definition of point contention), there is a client that executes two operations, both of which have the same first component of the timestamp. However, since each client executes operations sequentially, MW-ABD timestamp selection mechanism guarantees that the num component of the first timestamp will be greater than that of the second one. A contradiction. □

Theorem B.9 (Time Complexity) *The CAS-ABD time complexity is* adaptive *to concurrency guaranteeing that each operation op terminates in at most* $O(c^2)$ *base object accesses where* $c =$ *PntCont*(op).

Proof: Let t be the time when *op* invokes update. There are three types of operations that can obstruct *op*: (1) an operation that completes before time t; (2) an operation that starts but does not complete before time t; and (3) an operation invoked at time t or later. We next quantify the number of operations of each type that can obstruct *op*.

By Lemma B.7 at most two operation completing before time t can obstruct *op* on a given register. Thus, at most two operations fall into the first category. By definition of PntCont(op), the number of operations of the second type is at most PntCont(op). By Lemma B.6, this also implies that any operation op' of the third type, that is, starting at time t or later, satisfies $ts(op).num - ts(op').num \leq$ PntCont$(op) + 1$. Since operations with timestamps higher than $ts(op)$ cannot obstruct *op* (see line 6), we only care about the case $0 \leq ts(op).num - ts(op').num$. There are at most PntCont$(op) + 2$ numbers in this range. Since all operations that start at time t or later and obstruct *op* are concurrent with *op*, by Lemma B.8 there are at most PntCont(op) such operations whose first timestamp component is each of the numbers in the range described above. Overall, there are at most (PntCont$(op) + 2) *$ PntCont(op) operations with timestamps in this range, and in total there are PntCont$(op)^2 +$ 3PntCont$(op) + 2$ operations that may obstruct *op*.

Notice that an operation op' can obstruct *op* on an object b_i only by changing the value of b_i using *CAS* on line 5. By the specification of *CAS*, the old value of b_i was the expected value passed to *CAS* in this invocation during op'. By the conditions on lines 6 and 9, once this *CAS* returns, update terminates, and op' returns. This means that op' can obstruct *op* at most once. Since each operation can obstruct *op* at most once, PntCont$(op)^2 +$ 3PntCont$(op) + 2$ is an upper bound on the number of times a CAS invocation during *op* can fail (for each object). □