Consistency and Complexity Tradeoffs for Highly-Available Multi-Cloud Store

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

Abstract

Cloud-based storage services have established themselves as a paradigm of choice for supporting bulk storage needs of modern networked services and applications. Although individual storage service providers can be trusted to do their best to reliably store the user data, exclusive reliance on any single provider or storage service leaves the users inherently at risk of being locked out of their data due to outages, connectivity problems, and unforeseen alterations of the service contracts. An emerging multi-cloud storage paradigm addresses these concerns by replicating data across multiple cloud storage services, potentially operated by distinct providers. In this paper, we study the impact of the storage interfaces and consistency semantics exposed by individual clouds on the complexity of the reliable multi-cloud storage implementation. Our results establish several inherent space and time tradeoffs associated with emulating reliable objects over a collection of unreliable storage services with varied interfaces and consistency guarantees.

1 Introduction

A rapidly growing number of Internet companies offer Storage-As-A-Service to their customers. These include big corporations such as Amazon, Google, Microsoft, Apple, EMC, HP, IBM, AT&T, as well as numerous smaller providers such as Dropbox, Box, Rackspace, Nirvanix and many others. The popularity of cloud storage stems from its flexible deployment, convenient pay-per-use model, and little (if any) administrative overhead. This is especially attractive for smaller businesses, who cannot afford the costs of deploying and administering enterprise-scale storage infrastructure on their premises, and would rather outsource this task to an external entity.

Although cloud storage providers make tremendous investments into ensuring reliability and security of the service they offer, most of them have suffered from well-publicized outages where the integrity and/or availability of data have been compromised for prolonged periods of time [36, 17, 31]. In addition, even in the absence of outages, the customers can still lose access to their data due to connectivity problems, or unexpected alterations in the service contract. In fact, the problem of *data lock-in* [8] in which the customers become critically dependent on a specific cloud provider for all their data storage needs has long been considered a major roadblock to a wider adoption of cloud storage for sensitive data such as banking, medical, or critical infrastructure domains.

To address these concerns, *multi-cloud* storage systems whereupon the data is replicated across multiple cloud storage services (potentially operated by distinct providers) have recently become a hot topic in the systems community [37, 2, 46, 13, 12]. Note that since in this setup, the individual stores can be hosted by different cloud providers, the service implementation becomes the sole responsibility of a *user-side* proxy (such as a client library, or a middle tier) whose goal is to mediate between the users and the individual cloud stores so as to ensure data availability in the face of asynchrony, concurrency, and failures of both individual services and users.

Although a significant progress has so far been made in building practical multi-cloud storage systems [2, 13, 12], as of today, little is known about their fundamental capabilities and limitations. The primary challenge lies in a wide variety of the storage interfaces and consistency semantics offered by different cloud providers to their external users. For example, whereas Amazon S3 [40] supports a simple read/write interface, other storage services (such as, Amazon SimpleDB [42], Amazon DynamoDB [22], Microsoft Azure Storage [43], Yahoo's PNUTS [20], Apache ZooKeeper [29], Spinnaker [38], and Riak [39]) also expose a selection of more advanced transactional primitives, such as conditional writes.

In this paper, we initiate a rigorous study aimed to shed light on complexity trade-offs involved in building reliable storage services in fault-prone multi-cloud environments. Our departure point is to model a multi-cloud storage system as an asynchronous fault-prone shared memory system of Jayanti et al. [30] in which individual storage services are abstracted as fault-prone shared *base* objects, cloud users as processes accessing these objects, and a reliable multi-cloud storage service as a *fault-tolerant object emulation* consisting of the process algorithms interacting with the base objects. We then explore the space and time complexity associated with building such emulations as a function of the emulated object *type*, *the number of supported processes*, and the *safety properties* offered by the individual base objects being used.

Our first result establishes a lower bound on the space required to emulate a reliable wait-free k-writer/single-reader register supporting *safe* consistency [33, 32, 41], which is one of the most basic guarantees one could expect from a storage service[1]. We assume underlying storage services supporting read/write

[1]It remains open whether a similar bound also applies to weaker forms of consistency, such as e.g., sequential [34] or timeline [20] consistency.

and list primitives[2], which we model model as multi-writer/multi-reader (MWMR) *atomic snapshot* objects [3, 6, 7]. We prove (see Section 4) that irrespective of the number of failures being tolerated by the emulation, the *maximum* number of distinct processes (i.e., k) that could *ever* write the emulated register (either concurrently or not) is bounded by a quantity, which is *linear* in the number of entries supported by the underlying snapshot objects.

In other words, the space overhead associated with storing each individual data item (such as e.g., a single key/value) is proportional to the maximum number of clients that could ever update this item, and in particular, cannot be optimized under the assumption of bounded maximum concurrency (i.e., point contention [4, 10]). In practice, this means that architects of the multi-cloud storage services should avoid using speculative techniques that stipulate bounded peak load (such as e.g., memory overcommit [45, 14, 28]), but rather focus on limiting the *total* number of writers (e.g., through deploying proxies, or access control mechanisms) as the means of optimizing the space usage.

An important theoretical consequence of our lower bound is that it shows that shared memory algorithms based on reliable multi-writer registers are subject to a linear (in the number of writers) space blow-up when transformed to a fault-prone shared memory. This means that failures cause multi-writer registers to "lose" their ability to support multiple writers using constant space, rendering them equivalent to single-writer registers. Note that since our proof assumes base objects supporting multi-writer atomic snapshot, which is in a sense, the strongest possible read/write memory abstraction, our result also carries over to other multi-writer object emulations in this model, such as e.g., consensus, and multi-writer snapshots.

We next turn to emulating reliable registers over storage services supporting transactional update primitives. First, it is well known that a constant number of *read-modify-write* objects is indeed sufficient to reliably emulate a multi-writer atomic register [9, 26]. This implies that strengthening the underlying object semantics is indeed essential to avoid linear dependency on the number of writers implied by our lower bound. However, the read-modify-write objects employed by the existing implementations are too specialized to be exposed by the commodity storage interfaces. Instead, the primitives typically supported are variants of *conditional writes* (see e.g., [42, 22, 43, 20, 29, 38, 39]), which are essentially equivalent to *compare-and-swap (CAS)*. We therefore, adopt a shared memory system with fault-prone *CAS objects* as our model of cloud storage services supporting conditional writes, and study reliable object emulations in this environment.

In Section 5, we present a *constant* space implementation of a multi-writer atomic register, which requires the underlying clouds to only support a *single CAS* object per stored value, is adaptive to point contention (i.e., the maximum number of clients executing concurrently with the operation), and tolerates up to a minority of base object failures. In Section 6, we also show a constant space implementation of a Ranked Register (RR) [19] using a single fault-prone *CAS* object. A collection of such Ranked Registers can be used to construct a reliable Ranked Register, from which agreement is built [19]. Our construction thus obtains a multi-cloud state machine replication service capable of supporting infinitely many clients using constant space.

Finally, we believe that our work opens several interesting new directions for future research. To this end, in Section 7, we enumerate several open questions naturally arising from, or extending the results presented in this paper.

[2]The list primitive is supported by Amazon S3

2 Related Work

There are numerous algorithms emulating reliable shared memory objects using unreliable read-modify-write objects [9, 23, 23, 26, 21, 25, 5]. The specific read-modify-write functionality being assumed often depends on the implementation specifics, and therefore, incompatible with the existing cloud storage interfaces.

Notable exceptions are the SWMR regular register emulation by Gafni and Lamport [24], its Byzantine variant by Abraham et al. [1] and the recent algorithms by Basescu et al. [12] and Ye et al. [46], which use read/write registers. Standard techniques for transforming SWMR registers to support multiple writers (e.g., [44, 11, 16]) are expensive as they incur space complexity linear in the number of writers.

Basescu et. al [12] and Ye et al. [46] present an implementation of a reliable multi-cloud data store supporting multi-wrtiter atomic registers using storage primitives equivalent to atomic snapshot objects. Their algorithms incur worst-case space complexity proportional to the number of writers (regardless of concurrency), which matches our lower bound. In addition, [12] never uses less than 2 registers per written value, which is twice as much as required by our *CAS*-based algorithm. Other work in the systems community considered implementing multi-cloud stores resilient to data corruption using approaches such as erasure coding, and external coordination services [2, 13].

3 System Model

We consider an asynchronous shared memory system consisting of a collection of processes interacting with a finite collection of objects. Objects and processes are modeled as I/O automata [35]. An I/O automaton's state transitions are triggered by *actions*. Actions are classified as *input*, *output*, and *internal*. The automaton's interface is determined by its input and output actions, which are collectively called *external* actions. An action π of an automaton A is said to be *enabled* in state *s* if A has a state transition of the form (s, π, s'). The transitions triggered by input actions are always enabled, whereas those triggered by output and internal actions, (collectively called *locally controlled* actions), depend solely on the automaton's current state.

Execution, traces, and properties Let A be an I/O automaton. An execution α of A is a (finite or infinite) sequence of alternating states and actions $s_0, \pi_1, s_1, \ldots$, where s_0 is A's initial state, and each triple (s_{i-1}, π_i, s_i) is a state transition of A. The trace of an execution α of A is the subsequence of α consisting of the external actions in α. An infinite execution α of A is *fair* if every locally controlled action of A either occurs infinitely often in α or is disabled infinitely often in α. A finite execution α of A is *fair* if no locally controlled action of A is enabled at the end of α. A *fair trace* of A is the trace of a fair execution of A. An automaton's external behavior is specified in terms of the properties of its traces. Liveness properties are required to hold only in fair traces.

Object type An object automaton's interface is determined by its *type*, which is a tuple consisting of the following components: (1) a set V of values; (2) a set of *invocations*; (3) a set of *responses*; and (4) a *sequential specification*, which is a function from *invocations* $\times V$ to *responses* $\times V$, specifying the object's semantics in sequential executions.

Shared memory system An asynchronous shared memory system is a composition of a (possibly infinite) collection of process automata $P_1, P_2, \ldots$ and a finite collection of object automata $O_1, O_2, \ldots O_n$. Let O_j be an object of type $\mathcal{T}$, and a (b) be an invocation (resp. response) of $\mathcal{T}$. Process P_i interacts with O_j using actions of the form $O_j.a_i$ (resp. $O_j.b_i$), where a_i is an output of P_i and an input of O_j (resp. b_i is an output

of O_j and an input of P_i). For an execution α (resp., trace τ) of a shared memory system A, we will write $\alpha|i$ (resp., $\tau|i$) to denote a subsequence of α (resp., τ) consisting of only the invocation and return events occurring at P_i.

Well-formedness We say that the interaction between a process and an object is *well-formed* if it consists of alternating invocations and responses, starting from an invocation. In this paper, we only consider systems in which the interaction between P_i and O_j is well-formed for all i and j. Well-formedness allows an invocation occurring in an execution α to be paired with a unique response (when such exist). If an invocation has a response in α, the invocation is said to be *complete*; otherwise, it is *incomplete*. If two invocations are incomplete after some prefix of α, then they are said to be *overlapping* in α. Note that well-formedness does not rule out concurrent operation invocations on the same object by different processes. Nor does it rule out parallel invocations by the same process on different objects, which can be performed in separate threads of control.

Object failures Objects may suffer NR-Crash (non-responsive crash) failures [30]. An object experiencing an NR-Crash failure behaves correctly until it fails, and, once it fails, it never responds to any invocation. We consider t-tolerant implementations [30], which remain correct (in the sense that the emulated object satisfies its specification) whenever at most t base objects suffer NR-Crash failures.

Process failures Any number of the processes may fail by stopping. The failure of a process P_i is modeled using a special external event $stop_i$. Once $stop_i$ occurs, all locally controlled actions of P_i become disabled indefinitely. A process that does not fail in an execution is *correct* in that execution.

Atomicity and wait freedom We now define the Atomicity and Wait Freedom properties for an object of arbitrary type $\mathcal{T}$. Let σ be a well-formed sequence of $\mathcal{T}$'s invocations and responses.

Atomicity. Let σ' be a sequence obtained from σ by (1) assigning responses to a subset Φ of the invocations which are incomplete in σ, and (2) removing all incomplete invocations, which are not in Φ. σ is *atomic* [27] if all matching invocation and response pairs in σ' can be reordered so that the resulting ordering both preserves the order of the non-overlapping invocations, and satisfies the sequential specification of $\mathcal{T}$. An object O of type $\mathcal{T}$ is *atomic* if each trace τ of O is atomic.

Wait Freedom. σ satisfies *wait freedom* if every invocation in σ is complete. An object O of type $\mathcal{T}$ is *wait-free* if for each fair trace τ of O, if a process P_i is correct in τ, then $\tau|i$ satisfies wait freedom.

Below, we introduce several object types that will be used throughout the paper.

Registers The *read/write register* object type (or simply, a register) supports an arbitrary set of values V, and the initial value v_0. Its invocations are *read* and *write*(v), $v \in V$. Its responses are $v \in V$ and ack. Its sequential specification, f, requires that every WRITE overwrites the object's value with v and returns ack (i.e., $f(write(v), w) = (ack, v)$), and every READ returns the current object's value (i.e., $f(read, v) = (v, v)$).

A register is k-*writer* (resp., k-*reader*) denoted kW (resp., kR) if it can be written (resp., read) by at most $k > 0$ processes. We refer to a register as a *single-* or *multi-* writer (*SW* or *MW*) or reader (*SR* or *MR*) in the special cases of when it can be accessed (either for writing or reading) by 1 or any number of processes respectively.

Multi-writer safe register For our impossibility proofs, we will use a weak notion of multi-writer *safe* consistency for registers adapted from the weakest multi-writer regular consistency condition defined in [41]. Specifically, a well-formed sequence σ of invocations and responses of reads and writes is *safe* if each read invocation r that does not overlap any other write invocations returns the value of some write w that precedes r in σ, as long as no other write falls completely between w and r; or the register's initial value if no such w exists. A MWMR register is called *safe* if it has only safe traces.

Snapshot Objects Given an arbitrary set of values V, and an integer $m > 0$, the *Snapshot* [3, 6, 7] object type supports the set of values W, which are vectors of elements of V of length $m > 0$ with the initial value w_0, $w_0[i] = v_0$ for all $1 \leq i \leq m$. Its invocations are *write*(i, v), $v \in V$, and *list*. Its responses are ack and $w \in W$. Its sequential specification, f, requires that every *write*$(i, \cdot)$ overwrites the value stored in the object's ith component with v, and returns ack (i.e., $f($*write*$(i, v), w) = (ack, w')$ where $w'[i] = v$, and $w'[j] = w[j]$ for all $j \neq i$); and every *list* returns the current object's value (i.e., $f($*list*$, w) = (w, w)$).

A snapshot object is *multi-writer (MW)* if each individual component of its vector value can be written by any number of processes. It is *multi-reader (MR)* if *list* can be invoked by any number of processes.

Compare-and-Swap Objects The *compare-and-swap* object type supports an arbitrary set of values V, and the initial value v_0. Its invocations are *CAS*(u, v), $u, v \in V$, and *read*. Its responses are elements of V. Its sequential specification, f, requires that every *CAS* overwrites the object's value with v if it is equal to u, and leaves it intact otherwise returning the original object value in either case (i.e., $f($*CAS*$(u, v), w) = (w, v)$ if $u = w$; and (w, w), otherwise.); and every *read* returns the current object's value (i.e., $f($*read*$, w) = (w, w)$)[3]. All compare-and-swap objects considered in this paper can be updated and queried by any number of processes.

4 Space Requirements for Implementing MWSR Register

In this section, we prove that any implementation of a multi-writer safe register out of a collection of crash-prone atomic snapshot objects has space complexity linear in the *maximum* number of writers that could ever write to the emulated register. Specifically, our Lemma 4.2 shows that the amount of space used by the base objects may grow indefinitely even in executions where all writes are strictly serialized. This implies that the maximum number of clients that can be supported is limited by the amount of the available storage even in executions where only one of them can be simultaneously active at every given time. Thus, no implementation can be adaptive to the number of concurrently active writers (or point contention) in terms of its space consumption.

Let A be a t-tolerant implementation of a wait-free k-writer/1-reader safe register, supporting a set of values V, $|V| > k$, with the initial value v_0, using a collection of $n > t$ base wait-free atomic MWMR snapshot objects in the set O each of which can store vectors of length $m > 0$.

The following theorem asserts that the maximum number of k of clients that can be supported by A is limited by the amount of the storage space m available at each base object.

Theorem 4.1 $k \leq \lfloor (nm - t - 1)/t \rfloor$.

Given a snapshot object o, and an execution α of A, we say that a pair (j, o) where $1 \leq j \leq m$ is *covered* in α if o.*write*(j, v) is incomplete in α. We will write *Covered*(α) to denote the set of all integer/object pairs

[3]Note that *read* is simply a shortcut for *CAS*(v, v) for any $v \in V$. It is introduced here solely for presentation purposes.

which are covered in α. To distinguish the emulated register invocations from those of the underlying registers, we will henceforth use WRITE and READ to refer the writes and reads of the emulated register respectively.

Our proof strategy is based on the *coverage* argument of Burns and Lynch [15]. We first construct a failure-free execution ζ (see Lemma 4.2) in which $k-1$ writers take turns to write a unique value to the emulated registers in a strictly serialized fashion (i.e., the next write is not invoked until the previous one completes) so that at the end of each WRITE there exist at least t snapshot objects with 1 newly covered entry. We then show (see Lemma 4.5) that at least $2t+1$ non-covered base objects must exists after any failure-free finite execution with no incomplete WRITE invocations. We then show (see Theorem 4.1) that after ζ the total number of non-covered entries is $\leq 2t$ thus obtaining a contradiction.
For the lack of space, we will only present the outline of the proof. The full details can be found in Section A of the Appendix.

Let $\zeta_0 = s_0$ where s_0 is an initial state of A in which the initial value of the register is v_0, and all snapshot objects and processes are correct. We prove the following:

Lemma 4.2 *For all $1 \leq l \leq k$, there exists an execution $\zeta_l = \zeta_{l-1}, \text{WRITE}(v_l)_l, \gamma_l, ack_l$ such that*

1. *$v_l \neq v_j$ for all $1 \leq j < l$,*
2. *ζ_l is failure-free,*
3. *γ_l does not include any* WRITE *or* READ *invocations,*
4. *$|Covered(\zeta_l)| \geq lt$.*

The proof is by induction on $1 \leq l \leq k$. We start by proving the base case. Since $k \geq 1$ and $|V| > k$, there exists an input action $W_1 = \text{WRITE}(v_1)_1$ of the A's interface such that $v_1 \neq v_0$. Since all input actions of A are always enabled, s_0, W_1 is an execution of A.

The following lemma shows that W_1 can invoke *write* invocations on t distinct base objects without waiting for the previously issued invocations to respond.

Lemma 4.3 *Let $\alpha_0 = \beta_0 = s_0, W_1$. Then, for all $1 \leq i \leq t$, there exist two executions $\alpha_i = \alpha_{i-1}, \gamma_i, o_i.write$ and $\beta_i = \beta_{i-1}, \gamma_i, o_i.write, NR\text{-}crash(o_i)$ such that*

1. *γ_i is failure-free,*
2. *γ_i does not include any $o_j.ack_i$ responses for all $1 \leq j < i$,*
3. *$o_i \neq o_j$ for all $1 \leq j < i$.*

We proceed by extending the execution β_t constructed by Lemma 4.3 with an execution fragment γ' obtained by running A from the state reached after β_t, and until W_1 returns. Note that since all base objects $o_1, \ldots, o_t$ that were covered in β_t have crashed, they remain covered at the end of γ' as well. Thus, the execution $\zeta_1 = \zeta_0, \alpha_t, \gamma'$ obtained by grafting γ' after α_t satisfying the base case of Lemma 4.2. The full proof appears in Section A.1 of the Appendix.

We now turn to proving the inductive step of Lemma 4.2. Assume that $1 \leq l < k$. By the inductive hypothesis, there exists an execution ζ_l satisfying Lemma 4.2. We show how to construct ζ_{l+1} from ζ_l.

Since $k \geq 1$ and $|V| > k$, there exist $v_{l+1} \neq \ldots \neq v_0$ such that $W_{l+1} = \text{WRITE}(v_{l+1})_{l+1}$ is an input action of A's interface. Since all input actions of A are always enabled, ζ_l, W_{l+1} is an execution of A.

The following lemma asserts that W_{l+1} can cover t previously non-covered entries each of which residing on a distinct snapshot object (see Lemma 4.4.2-4) without waiting for responses from any snapshot objects covered in the course of the WRITE invocations preceding W_{l+1} (see Lemma 4.4.5).

Lemma 4.4 *Let $\alpha_0 = \zeta_l, W_{l+1}$, and $\beta_0 = \zeta_l, stop_1, \ldots, stop_l, W_{l+1}$. Then, for all $1 \leq i \leq t$, there exist three executions $\alpha_i = \alpha_{i-1}, \gamma_i, o_i.write(\ell_i, \cdot)_{l+1}$; and $\beta_i = \beta_{i-1}, \gamma_i, o_i.write(\ell_i, \cdot)_{l+1}, NR\text{-}crash(o_i)$ s.t.*

1. *γ_i is failure-free,*
2. *$o_i \neq o_j$ for all $1 \leq j < i$,*
3. *$(\ell_i, o_i) \notin Covered(\zeta_l)$,*
4. *γ_i does not include any ack_{l+1} responses from objects o_j, $1 \leq j < i$, and*
5. *γ_i does not include any $o.ack$ responses from all objects o such that $(\cdot, o) \in Covered(\zeta_l)$.*

We proceed by extending the execution β_t constructed by Lemma 4.4 with an execution fragment γ' obtained by running A from the state reached after β_t until W_{l+1} returns. Let $\zeta_{l+1} = \alpha_t, \gamma', ack_{l+1}$. Note that the first three claims of Lemma 4.2 are satisfied in ζ_{l+1} by construction. Furthermore, by claims 3, 4, and 5 of Lemma 4.4, $|Covered(\alpha_t)| = |Covered(\zeta_l)| + t = (l+1)t$. Moreover, by construction, none of the *write* invocations made on pairs $(\ell, o) \in Covered(\alpha_t)$ returns in γ'. Therefore, $|Covered(\zeta_{l+1})| = |Covered(\alpha_t)| = (l+1)t$. Hence, claim 4 of Lemma 4.2 is also satisfied. This completes the proof of Lemma 4.2 (full details can be found in Section A.4 of the Appendix).

We now show that after each finite execution α of A in which all WRITE invocations are complete, there must be at least $2t+1$ base objects each of which has at least one non-covered entry. Intuitively, these $2t+1$ non-covered entries are needed in order to ensure that a WRITE invoked after α will be able to complete. This intuition is formalized by the following

Lemma 4.5 *Let α be a finite failure-free execution of A in which all* WRITE *invocations are complete. Then, $|\{o : (\cdot, o) \notin Covered(\alpha)\}| > 2t$.*

We are now ready to prove Theorem 4.1.

Proof: [Proof of Theorem 4.1] Assume by contradiction that $k = \lfloor (nm - t - 1)/t \rfloor + 1$. By Lemma 4.2, there exists an execution ζ of A in which $k-1$ writes by $k-1$ different processes are invoked such that $|Covered(\alpha)| \geq \lfloor (nm - t - 1)/t \rfloor t \geq nm - 2t$. Hence, at the time the kth WRITE is invoked, $|\{(\ell, o) \notin Covered(\alpha)\}| \leq 2t$. Therefore, $|\{o : (\cdot, o) \notin Covered(\alpha)\}| \leq 2t$ contradicting Lemma 4.5. □

5 Atomic Register Implementation

In this section we outline our wait-free implementation of a *MWMR* atomic register with space complexity 1. Unlike previous approaches that feature constant space complexity, our algorithm does not require support for any specific read-modify-write functionality besides *CAS*, i.e., conditional write, obviating the need for server code. In contention-free executions, our algorithm attains optimal constant step-complexity. Under contention, the step-complexity of an operation is bounded by $O(\mathcal{C}^2)$, where $\mathcal{C}$ denotes the *point contention* [4, 10], i.e., the maximum number of processes simultaneously concurrent with the operation.

Our implementation, shown in Algorithm 1, is a derivation of the multi-writer variation of the ABD protocol [9] which we now briefly describe. In ABD, each object stores a timestamp-value pair (ts, v). To write a value v, a process proceeds in two phases. In the first phase, the process queries the objects for their stored timestamp-value pair, waits for a majority to respond, and chooses a *unique* timestamp ts which is higher than any previous timestamp. In the second phase, the process updates the objects with (ts, v). To read a value, a process proceeds in a similar way. In the first phase, it queries the objects

```
1: Definitions
2:     TS: (ℕ0 × {ℕ0 ∪ ⊥}) with selectors num and pid                    // timestamps
3:     TSVal: (TS × 𝒱) with selectors ts and val
4: Compare-And-Swap base objects
5:     for 1 ≤ k ≤ n, x_k ∈ TSVal is a Compare-And-Swap object, initially ((0, ⊥), v0)
6: State
7:     x[k]: TSVal, for 1 ≤ k ≤ n, initially ((0, ⊥), v0)
8:     new: TSVal ∪{⊥}, initially ⊥
9:     R1, R2 : 2^ℕ0, initially ∅
```

```
10: operation WRITE(v)
11:     init()
12:     x_max ← query()
13:     new ← ((x_max.ts.num + 1, i), v)
14:     update(new, R1)
15:     return OK
16: operation READ()
17:     init()
18:     new ← query()
19:     update(new, R1)
20:     return new.val
21: procedure query()
22:     in parallel for 1 ≤ k ≤ n do
23:         invoke x_k.read()
24:     wait until |R1| ≥ ⌈(n + 1)/2⌉
25:     k← ARGMAX_{k∈R1}{x[k].ts}
26:     return x[k]
27: upon completion of x_k.read() returning v
28:     x[k] ← v
29:     if new = ⊥ then
30:         R1 ← R1 ∪ {k}
31:     else invoke rmw(k, new)
32: procedure update(new, T)
33:     in parallel for all k ∈ T do
34:         invoke rmw(k, new)
35:     wait until |R2| ≥ ⌈(n + 1)/2⌉
36: procedure rmw(k, new)
37:     DONE ← false
38:     exp ← x[k]
39:     if new.ts > exp.ts then
40:         repeat
41:             old ← x_k.CAS(exp, new)
42:             if old = exp ∨ old.ts > new.ts then
43:                 DONE ← true
44:             exp ← old
45:         until DONE
46:     R2 ← R2 ∪ {k}
47: procedure init()
48:     for 1 ≤ k ≤ n, x[k] ← ((0, ⊥), v0)
49:     new ← ⊥
50:     R1, R2 ← ∅
```

Algorithm 1: Atomic register implementation. Code for process P_i.

for their timestamp-value pair, waits for a majority to respond and picks the pair (ts, v) with the highest timestamp. In the second phase, the process writes-back (ts, v) to a majority of objects and returns v. An essential requirement is that an object never changes state to a value with a lower timestamp. In ABD, the object compares the new timestamp to the one currently stored, and updates the stored value *if and only if* the new timestamp is higher (notice that in practice, a server implementing the object has to do timestamp comparison, which requires server code).

In contrast, in our algorithm the comparison is not performed by the storage objects. It is performed by the processes using the *rmw* procedure, which is called in the second phase of an operation, for every object. For a timestamp-value pair (ts, v) and object x, *rmw*$(x, (ts, v))$ satisfies three properties: (P1) if *rmw* returns then $x.ts$ is at least ts, (P2) x is changed to (ts, v) only if ts is higher than $x.ts$ and (P3) *rmw* eventually returns. Intuitively, property P1 captures the "if" part in the ABD requirement, and property P2 captures the "only if" part. Finally, property P3 is needed for wait-freedom.

To perform *rmw* with timestamp-value pair (ts, v) on object x, the process enters a loop of *CAS* interactions with x, from which it only exits after successfully updating x with (ts, v) or after finding a higher timestamp in $x.ts$ (satisfying P1). Notice that x is always changed using *CAS*. Furthermore, notice that ev-

ery invocation *CAS*(*exp*,*new*) satisfies *exp* $<$ *new*. Since *CAS* changes x to *new* only if the current value of x equals to *exp*, $x.ts$ is strictly monotonically increasing (satisfying P2). Finally, since only operations with timestamp lower than ts may obstruct the process on object x, the process does not execute the loop forever (satisfying P3). In fact, we prove that the step complexity of each operation adapts to its point contention, as we explain below.

Note that for the sake of readability Algorithm 1 does not include the (standard) implementation details required to guarantee the following properties: (1) each process has at most one pending invocation on any object (well-formedness); (2) after an operation o by process P_i completes, object responses received by P_i for invocations made during o are ignored and do not change the state of P_i, and (3) at most one operation may be invoked by P_i on any object after o completes.

Atomicity Proof Sketch The formal analysis of our algorithm is given in Appendix B. In the proof we associate a timestamp with each operation. For a read (write) this is the timestamp of the returned (resp., written) value. Lemma B.2 proves property P2 (essentially, this property implies timeline consistency, i.e., that the timestamp of every object cannot go back in time). Lemma B.3 shows that if an operation o completes before another operation o' is invoked, then the timestamp of o' will be at least as high as that of o, and if o' is a write operation, it will be strictly higher. We then show that different write operations have different timestamps (Lemma B.4) and that a read always returns a value written by some previously invoked write with the same associated timestamp (Lemma B.5). Finally, Theorem B.6 proves that Algorithm 1 implements an atomic read/write register, by explicitly constructing the sequential permutation required by the definition of atomicity. For a trace of the object σ produced by the algorithm, we construct σ' from σ by completing all WRITE operations of the form WRITE(v), where v has been returned by some complete READ operation. We then construct a sequential permutation π by ordering all operations in σ' according to their associated timestamps and by placing all READ operations immediately after the WRITE operation with the same timestamp or in the beginning of the execution if the timestamp of the read is $(0, \perp)$. Reads with the same timestamp are ordered according to their invocation order in σ. We show that π satisfies the two properties required by the definition of atomicity, namely the operation precedence relation and the sequential specification of a read/write register.

Step Complexity In the best case, our algorithm requires two operations per object, a *read* and *CAS*, and therefore its step-complexity is 2. Since *CAS* requires the current value as an input, the process first invokes a *read* on an object during the first round and only then a CAS on the same object during the second round. It is possible, however, that an object responding to a *read* fails before responding to the CAS. We therefore execute the first and the second round of an operation concurrently – even after a process receives a majority of *read* responses and awaits to complete *rmw* on a majority of the objects, it will process the completion of a *read* and invoke a *rmw* on that object (this 'late' *rmw* may be necessary to form a majority of *rmw* completions). Notice that during an operation, *rmw* is called at most once for each object.

In the worst case, the step-complexity of an operation is bounded by $O(\mathcal{C}^2)$, where $\mathcal{C}$ denotes the point contention during the operation. To see why, suppose that o is an operation that invokes *rmw* on object x at time t. There are three types of operations that can obstruct o on x: (Type 1) operations that complete before time t, (Type 2) operations that start but do not complete before time t and (Type 3) operations that start at time t or later. Any number of operations of Type 1 can obstruct o at most twice (by Lemma B.8 in Appendix B). By definition of point contention, there are at most $\mathcal{C}$ operations of Type 2. Finally, there are at most $(\mathcal{C}+2)\mathcal{C}$ operations of Type 3. To understand why, notice that the difference between the sequence number in the timestamp of o and the the sequence number in the timestamp of an operation of Type 3 lies

between 0 and $k+1$, where k is the number of Type 2 operations (by Lemma B.7 in Appendix B). Recall that there are at most $\mathcal{C}$ operations of Type 2, and so $k \leq \mathcal{C}$. For each sequence number num in this range of at most $\mathcal{C}+2$ sequence numbers, there are at most $\mathcal{C}$ operations whose timestamps share num (Lemma B.9 in Appendix B). Therefore, the number of Type 3 operation is at most $(\mathcal{C}+2)\mathcal{C}$. Since each operation obstructs o only once (per object), the number of *CAS* invocations during o is bounded by $O(\mathcal{C}^2)$ (*read* is invoked once for each object). The algorithm invokes operations on different objects in parallel, in separate threads of control.

6 Ranked Register Implementation

In this section we outline our implementation of a single wait-free shared ranked register. As shown in [19], the ranked register is a sufficient building block for Consensus. Our algorithm can be used as a substitute of the single ranked register implementation in [19] to obtain a fault-tolerant implementation of any linearizable object from a collection of fault-prone ranked registers. Unlike the implementation outlined in [19], which requires objects with read-modify-write functionality specific to ranked registers, our implementation builds on basic *CAS* objects.

6.1 Preliminaries

We start by giving a formal specification of a ranked register. Let *Ranks* be a totally ordered set of ranks with an initial rank r_0 such that for all $r \in Ranks$, $r > r_0$. A ranked register is a *MWMR* shared object with two operations: *rr-read*(r) with $r \in$ *Ranks*, whose return value is $(r, v) \in$ *Ranks* $\times \mathcal{V}$, and *rr-write*((r, v)) with $(r, v) \in$ *Ranks* $\times \mathcal{V}$, whose return value is either *commit* or *abort*. We say that an *rr-write* operation *committs* (rep. *aborts*) when its reply is *commit* (reps. *abort*). We assume that ranks used in *rr-write* operations are unique. Typically, this is implemented by using unique process ids and a sequence number. An implementation of ranked register has to satisfy the properties of *Safety*, *Non-triviality* and *Liveness* (adopted from [19]).

Definition 6.1 (Safety) *Every rr-read operation returns a value and rank that was written in some rr-write invocation or (r_0, v_0). Additionally, let $W =$ rr-write$((r_1, v_1))$ be a write operation that commits, and let $R =$ rr-read(r_2), such that $r_2 > r_1$. Then R returns (r, v) where $r \geq r_1$.*

Definition 6.2 (Non-Triviality) *If a rr-write operation W invoked with rank r_1 aborts, then there exists an operation with rank $r_2 > r_1$ which returns before W is invoked, or is concurrent with W.*

Definition 6.3 (Liveness) *If an operation is invoked by a correct process, then eventually it returns.*

We now proceed to describing the implementation given in Algorithm 2. The implementation makes use of a single shared *CAS* object $x \in X$ with three fields $x.rR$, $x.wR$ and $x.val$. The field $x.rR$ holds the highest rank of any invoked operation, whereas $x.wR$ holds the highest rank of any invoked *rr-write* operation and $x.val$ the corresponding value.

To *rr-write* a rank-value pair (r, v), the client enters a loop from which is only exits (a) after successfully updating x with the triple (r,r,v), in which case the operation commits, or (b) after finding a higher rank in $x.rR$, in which case the operation aborts. To *rr-read* with a rank r, the client enters a similar loop from which it only exists (a) after successfully recording its rank in the $x.rR$ field of the object or (b) after finding

a higher rank in $x.rR$. Finally, the operation returns the rank and value currently stored in the fields $x.wR$ and $x.val$.

We now briefly describe the intuition behind Algorithm 2 by informally arguing that it satisfies Safety, Non-Triviality and Liveness. To see why Safety is satisfied, consider a write $W =$ *rr-write*$((r_1, v_1))$ that commits and a read $R =$ *rr-read*(r_2), such that $r_2 > r_1$. Since W commits, x changes state to (r_1, r_1, v_1). Notice that by the way the algorithm invokes $x.CAS$, x never changes its state to a lower rank. By Algorithm 2, if R returns (r, v), then the value of x changed to (r', r, v) such that $r' \geq r_2 > r_1$. Since W commits, x transitions to (r', r, v) only after it changed to (r_1, r_1, v_1). Since ranks never go back in time, it follows that $r \geq r_1$. Intuitively, Non-triviality is satisfied because if a write aborts, then x has been previously changed to a higher rank by some other operation. Finally, Liveness is satisfied because an operation with rank r can be prevented from termination only by operations with lower ranks, and the number of such operations is bounded by r.

```
1: Types
2:     X: (Ranks × Ranks × V) ∪ {(r_0, r_0, v_0)} with selectors rR and wR and val
3: Base-objects
4:     x ∈ X shared, initially x = (r_0, r_0, v_0)
---------------------------------------------------------------------------
5: operation rr-write((r, v))
6:     DONE ← false
7:     exp ← (r_0, r_0, v_0)
8:     new ← (r, r, v)
9:     repeat
10:        old ← x.CAS(exp, new)
11:        if old = exp ∨ old.rR > r then
12:            DONE ← true
13:        exp ← old
14:    until DONE
15:    if old.rR > r then
16:        return abort
17:    else return commit

18: operation rr-read(r)
19:    DONE ← false
20:    exp ← (r_0, r_0, v_0)
21:    new ← (r, r_0, v_0)
22:    repeat
23:        old ← x.CAS(exp, new)
24:        if old = exp ∨ old.rR > r then
25:            DONE ← true
26:        exp ← old
27:        new ← (r, old.wR, old.val)
28:    until DONE
29:    return (old.wR, old.val)
```

Algorithm 2: Implementation of a single ranked register. Code for process P_i.

6.2 Correctness of Algorithm 2

Lemma 6.4 *Algorithm 2 satisfies Safety.*

Proof: Is is clear from the code that a *rr-read* operation can only return a rank-value pair that was used in a *rr-write* operation or (r_0, v_0). Suppose that $W =$ *rr-write*$((r_1, v_1))$ is a *rr-write* operation that commits. Furthermore, suppose $R =$ *rr-read*(r_2) is a *rr-read* such that $r_2 > r_1$ and let (r, v) be the rank-value pair returned by R. We need to show that $(r, v) = (r_1, v_1)$ or $r > r_1$. For the purpose of contradiction, suppose that $r < r_1$. Since W committed, the first part of the condition in line 11 is satisfied, and it follows that W has set x to (r_1, r_1, v_1) at time t_1. Since R returns (r, v), by the condition in line 24, some operation has changed x to (r', r, v) where $r' \geq r_2$ at time t_2. Notice that the new value supplied to *CAS* in lines 10 and 23 has never a lower rank than the expected value. As such, the ranks of x never decrease. Since $r' \geq r_2 > r_1$, it follows that $t_2 > t_1$. Furthermore, since x changes to (r', r, v) only after it changed to (r_1, r_1, v_1), and

ranks never decrease, it follows that $r \geq r_1$. As no two different values are written with the same rank, if $r = r_1$ then $v = v_1$. □

Lemma 6.5 *Algorithm 2 satisfies Non-Triviality.*

Proof: If W aborts, then according to the condition in line 15 some operation has previously changed $x.rR$ to a rank $r_2 > r_1$. This can happen only as a result of some previously returned or concurrent *rr-read* or *rr-write* operation with rank $r_2 > r_1$, as required. □

Lemma 6.6 *Algorithm 2 satisfies Liveness.*

Proof: Let o be a *rr-read* (resp. *rr-write*) operation with rank r, and for the purpose of contradiction, suppose that o never returns. Hence, o is stuck in an infinite loop, which means that the condition in line 24 (resp. 11) never holds. This implies that an infinite number of operations with rank lower than r prevent o from updating x in line 23 (resp. 10). Notice however that by rank uniqueness, the number of operations o' with rank $r' < r$ is upper bounded by r, a contradiction. □

7 Open Questions

This work raises numerous open questions for future research:

- The step complexity of Algorithm 1 is quadratic in point contention. Is this optimal for emulating atomic registers? Is this optimal for weaker shared object types?
- Our lower bound in Section 4 is shown for safe registers which, despite being weaker than atomic and regular registers, still require that a read returns the value of a most recently completed write (if it does not overlap the write). It would be interesting to show a similar lower bound for consistency conditions that allow returning stale values, such as sequential, timeline, or causal consistency, etc.
- Our space bound proof in Section 4 critically depends on a capability to terminate each WRITE without waiting for the responses from the objects covered by the prior WRITE invocations. Note that this capability is no longer available if each WRITE is guaranteed to terminate, or in other words, all writers are correct. Since the writer reliability can be enforced in many practical settings, it will be interesting to see whether a constant memory algorithm can be constructed under the assumption of reliable writers, or the space bound can be further strengthened to also apply in this case.
- We conjecture that there is a simple algorithm implementing a $k*m$-writer/multi-reader atomic register out of $n > (k+1)t$ atomic snapshot objects of length m. This algorithm will use each "row" of n snapshot slots to support k writers using an algorithm similar to ABD. Thus, the number of supported writers can be bounded from below by $(\lceil (n-2t)/t \rceil)m$.
- We further conjecture that the above bound is tight since it is possible to choose $(\lceil (n-2t)/t \rceil)$ subsets S_i out of $n > t$ snapshot objects each of which consisting of t objects so that any implementation will have a run where each WRITE invocation terminates while leaving *write* invocations pending on all objects in one of the sets S_i.

Conclusions

In this paper, we initiated a rigorous study aimed to shed light on the consistency and complexity trade-offs involved in building reliable storage services in fault-prone multi-cloud environments. We proved that when the interfaces of constituent cloud services are limited to read/write primitives, every multi-cloud replication solution must use space proportional to the maximum number of writers to reliably emulate multi-writer safe register (or a stronger shared storage primitive). This matches the complexity of existing algorithms and shows that their worst-case space complexity is optimal. We then show that by leveraging conditional writes (modeled as compare-and-swap), readily available with most existing cloud storage interfaces, one can implement a reliable atomic register using a single replicated object per emulated data item. The step complexity of our algorithm adapts to point contention. Finally, our work opens a number of interesting new directions for future research which we list in Section 7.

References

[1] Ittai Abraham, Gregory Chockler, Idit Keidar, and Dahlia Malkhi. Byzantine disk Paxos: Optimal resilience with Byzantine shared memory. *Distributed Computing*, 18(5):387–408, 2006.

[2] Hussam Abu-Libdeh, Lonnie Princehouse, and Hakim Weatherspoon. RACS: a case for cloud storage diversity. In *Symposium on Cloud Computing (SoCC)*, pages 229–240, 2010.

[3] Yehuda Afek, Hagit Attiya, Danny Dolev, Eli Gafni, Michael Merritt, and Nir Shavit. Atomic snapshots of shared memory. *J. ACM*, 40(4):873–890, 1993.

[4] Yehuda Afek, Hagit Attiya, Arie Fouren, Gideon Stupp, and Dan Touitou. Long-lived renaming made adaptive. In *Proceedings of the eighteenth annual ACM symposium on Principles of distributed computing*, PODC '99, pages 91–103, New York, NY, USA, 1999. ACM.

[5] Marcos K. Aguilera, Idit Keidar, Dahlia Malkhi, Jean-Philippe Martin, and Alexander Shraer. Reconfiguring replicated atomic storage: A tutorial. *Bulletin of the EATCS*, 102:84–108, 2010.

[6] James H. Anderson. Composite registers. *Distributed Computing*, 6(3):141–154, 1993.

[7] James H. Anderson. Multi-writer composite registers. *Distributed Computing*, 7(4):175–195, 1994.

[8] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. A view of cloud computing. *Commun. ACM*, 53(4):50–58, April 2010.

[9] Hagit Attiya, Amotz Bar-Noy, and Danny Dolev. Sharing memory robustly in message-passing systems. *J. ACM*, 42(1):124–142, January 1995.

[10] Hagit Attiya and Arie Fouren. Algorithms adapting to point contention. *J. ACM*, 50(4):444–468, July 2003.

[11] Hagit Attiya and Jennifer Welch. *Distributed Computing: Fundamentals, Simulations and Advanced Topics*. John Wiley & Sons, 2004.

[12] Cristina Basescu, Christian Cachin, Ittay Eyal, Robert Haas, Alessandro Sorniotti, Marko Vukolic, and Ido Zachevsky. Robust data sharing with key-value stores. In *DSN*, pages 1–12, 2012.

[13] Alysson Bessani, Miguel Correia, Bruno Quaresma, Fernando André, and Paulo Sousa. Depsky: Dependable and secure storage in a cloud-of-clouds. In *European Conference on Computer Systems (EuroSys)*, 2011.

[14] David Breitgand and Amir Epstein. Improving consolidation of virtual machines with risk-aware bandwidth oversubscription in compute clouds. In *INFOCOM*, pages 2861–2865, 2012.

[15] James E. Burns and Nancy A. Lynch. Bounds on shared memory for mutual exclusion. *Inf. Comput.*, 107(2):171–184, 1993.

[16] Christian Cachin, Rachid Guerraoui, and Luís Rodrigues. *Introduction to Reliable and Secure Distributed Programming (Second Edition)*. Springer, 2011.

[17] Rory Cellan-Jones. The Sidekick Cloud Disaster. http://www.bbc.co.uk/blogs/technology/2009/10/the_sidekick_cloud_disaster.html, 2009.

[18] Gregory Chockler, Dan Dobre, and Alex Shraer. Consistency and complexity tradeoffs for highly-available multi-cloud store. https://pure.rhul.ac.uk/admin/files/16996922/main.pdf, 2013.

[19] Gregory Chockler and Dahlia Malkhi. Active disk paxos with infinitely many processes. *Distrib. Comput.*, 18(1):73–84, July 2005.

[20] Brian F. Cooper, Raghu Ramakrishnan, Utkarsh Srivastava, Adam Silberstein, Philip Bohannon, Hans-Arno Jacobsen, Nick Puz, Daniel Weaver, and Ramana Yerneni. Pnuts: Yahoo!'s hosted data serving platform. *Proc. VLDB Endow.*, 1(2):1277–1288, August 2008.

[21] Partha Dutta, Rachid Guerraoui, Ron R. Levy, and Marko Vukolic. Fast access to distributed atomic memory. *SIAM Journal on Computing*, 39(8):3752–3783, 2010.

[22] Amazon DynamoDB. http://aws.amazon.com/dynamodb/.

[23] Burkhard Englert and Alexander A. Shvartsman. Graceful quorum reconfiguration in a robust emulation of shared memory. In *International Conference on Distributed Computing Systems (ICDCS)*, pages 454–463, 2000.

[24] Eli Gafni and Leslie Lamport. Disk Paxos. *Distributed Computing*, 16(1):1–20, 2003.

[25] Chryssis Georgiou, Nicolas C. Nicolaou, and Alexander A. Shvartsman. Fault-tolerant semifast implementations of atomic read/write registers. *Journal of Parallel Distributed Computing*, 69(1):62–79, 2009.

[26] Seth Gilbert, Nancy A. Lynch, and Alexander A. Shvartsman. Rambo: a robust, reconfigurable atomic memory service for dynamic networks. *Distributed Computing*, 23(4):225–272, 2010.

[27] Maurice Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990.

[28] Michael R. Hines, Abel Gordon, Márcio Silva, Dilma Da Silva, Kyung Dong Ryu, and Muli Ben-Yehuda. Applications know best: Performance-driven memory overcommit with ginkgo. In *CloudCom*, pages 130–137, 2011.

[29] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. Zookeeper: wait-free coordination for internet-scale systems. In *Proceedings of the 2010 USENIX annual technical conference (ATC)*, Berkeley, CA, USA, 2010.

[30] P. Jayanti, T. Chandra, , and S. Toueg. Fault-tolerant wait-free shared objects. *Journal of the ACM*, 45(3):451–500, 1998.

[31] Michael Krigsman. MediaMax / The Linkup: When the cloud fails. http://blogs.zdnet.com/projectfailures/?p=999, 2008.

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

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

[34] Leslie Lamport. How to make a correct multiprocess program execute correctly on a multiprocessor. *IEEE Trans. Computers*, 46(7):779–782, 1997.

[35] N. A. Lynch and M.R. Tuttle. An introduction to Input/Output Automata. *CWI Quarterly*, 2(3):219–246, 1989.

[36] Richard MacManus. More Amazon S3 Downtime: How Much is Too Much? `http://readwrite.com/2008/07/20/more_amazon_s3_downtime`, 2008.

[37] TClouds Project. Privacy and resilience for Internet-scale critical infrastructures. `http://www.tclouds-project.eu`.

[38] Jun Rao, Eugene J. Shekita, and Sandeep Tata. Using paxos to build a scalable, consistent, and highly available datastore. *PVLDB*, 4(4):243–254, 2011.

[39] Riak. `http://basho.com/riak`.

[40] Amazon Simple Storage Service (Amazon S3). `http://aws.amazon.com/s3/`.

[41] C. Shao, E. Pierce, and J. L. Welch. Multi-writer consistency conditions for shared memory objects. In *DISC 2003*, pages 106–120, 2003.

[42] Amazon SimpleDB. `http://aws.amazon.com/simpledb/`.

[43] Microsoft Azure Storage. `http://www.windowsazure.com/en-us/manage/services/storage`.

[44] Paul M. B. Vitányi and Baruch Awerbuch. Atomic shared register access by asynchronous hardware (detailed abstract). In *FOCS*, pages 233–243, 1986.

[45] Dan Williams, Hani Jamjoom, Yew-Huey Liu, and Hakim Weatherspoon. Overdriver: handling memory overload in an oversubscribed cloud. In *Proceedings of the 7th ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, VEE '11, pages 205–216, New York, NY, USA, 2011. ACM.

[46] Yunqi Ye, Liangliang Xiao, I-Ling Yen, and Farokh Bastani. Secure, dependable, and high performance cloud storage. In *Proceedings of the 29th Symposium on Reliable Distributed Systems (SRDS)*, pages 194–203, 2010.

A Proof of Theorem 4.1

In this section, we will present full formal proofs of the claims in Section 4.

A.1 Proof of Lemma 4.3

Lemma A.1 *Let $\alpha_0 = \beta_0 = s_0, W_1$. Then, for all $1 \leq i \leq t$, there exist two executions $\alpha_i = \alpha_{i-1}, \gamma_i, o_i.write$ and $\beta_i = \beta_{i-1}, \gamma_i, o_i.write, NR\text{-}crash(o_i)$ such that*

1. *γ_i is failure-free,*
2. *γ_i does not include any $o_j.ack_i$ responses for all $1 \leq j < i$,*
3. *$o_i \neq o_j$ for all $1 \leq j < i$.*

Proof: The proof is by induction on $1 \leq i \leq t$.

Base Case: Since, A is wait-free, there exists an execution fragment γ such that $\alpha = s_0, W_1, \gamma, ack_1$ is an execution of A. Note that at least one write must be invoked on some snapshot object in γ for otherwise, α will be indistinguishable from s_0 to a process invoking a read operation R after α causing R to return $v_0 \neq v_1$ violating safety. Let $w_1 = o_1.write_p$ be the first such write in γ, and γ_1, w_1 be the prefix of γ ending with w_1. Since no snapshot objects are crashed in α_0, and γ_1 is failure-free, $\alpha_1 = \alpha_0, \gamma_1$, and $\beta_1 = \beta_0, \gamma_1, NR\text{-}crash(o_1))$ satisfy the lemma's requirements.

Inductive Step: Assume that $1 \leq i < t$. We use our inductive hypothesis for α_i and β_i, and show how to construct α_{i+1} and β_{i+1} required by the lemma.

By wait freedom, there exists a failure-free execution fragment γ such that $\beta = \beta_i, \gamma, ack_1$ is an execution of A. Note that γ must include *write* invoked on a snapshot object, which is not in $\{o_j : j \leq i\}$. Indeed, since all snapshot objects, which have been written in the course of β_i, have crashed, not writing to any not previously written object will result in β and s_0 being indistinguishable to a process invoking a read operation R after β causing R to return $v_0 \neq v_1$ violating safety.

Let $w_{i+1} = o_{i+1}.write_p$ be the first write invoked in γ on an object $o_{i+1} \notin \{o_j : j \leq i\}$, and γ_{i+1}, w_{i+1} be the prefix of γ ending with w_{i+1}. Since α_i and β_i are indistinguishable to process 1, and the response of w_{i+1} can be arbitrarily delayed, γ_{i+1}, w_{i+1} is also a valid continuation of α_i. Furthermore, since the number of snapshot objects that have crashed in β_i is $< t$, $\gamma_{i+1}, w_{i+1}, NR\text{-}crash(o_{i+1})$ is a valid continuation of β_i.

Hence, $\alpha_{i+1} = \alpha_i, \gamma_{i+1}, w_{i+1}$, and $\beta_{i+1} = \beta_i, \gamma_{i+1}, w_{i+1}, NR\text{-}crash(o_{i+1})$ satisfy the lemma's requirements. □

A.2 Proof of the Base Case of Lemma 4.2

The following lemma asserts that the base case of Lemma 4.2 is true:

Lemma A.2 *(*Base Case of Lemma 4.2*) There exists an execution ζ_1 satisfying Lemma 4.2 for $l = 1$.*

Proof: Let α_t and β_t be the two executions of A whose existence is asserted by Lemma 4.3. Note that by Lemma 4.3, $|Covered(\alpha_t)| = |Covered(\beta_t)| = t$.

Since A is wait-free, and process 1 is correct in β_t, there exists a failure-free execution fragment γ' such that $\beta = \beta_t, \gamma', ack_1$ is an execution of A. Since α_t and β_t are indistinguishable to process

1, and the responses of all outstanding *write* invocations in α_t can be arbitrarily delayed, γ' is also a valid continuation of α_t. Moreover, since all snapshot objects on which a *write* operation has been invoked in both α_t and β_t have crashed in β_t, none of the *write* invocations in α_t return in γ'. Therefore, $|\textit{Covered}(\alpha_t, \gamma', ack_p)| \geq |\textit{Covered}(\alpha_t)|$ so that $|\textit{Covered}(\alpha_t, \gamma', ack_p)| \geq t$. Hence, $\zeta_1 = \alpha_t, \gamma', ack_p$ satisfy the lemma's requirements. □

A.3 Proof of Lemma 4.4

In this section we present details of the proof of the inductive step of Lemma 4.2.

The proof is by induction on $1 \leq i \leq t$. We start by proving the base case:

Proof: [Base Case of Lemma 4.4]

By wait freedom, there exists an execution fragment γ such that $\beta = \beta_0, \gamma, ack_{l+1}$ is an execution of A. Since β_0 and α_0 are indistinguishable to process $l+1$, and incomplete operations are allowed to take effect arbitrarily in the future, $\alpha = \alpha_0, \gamma, ack_{l+1}$ is also an execution of A.

Note that at least 1 *write* must be invoked in γ as otherwise, α_0 and α will be indistinguishable to a process invoking a READ operation R after α causing R to return $v_l \neq v_{l+1}$ violating safety. Moreover, at least one of the *write*'s in γ must be invoked on a pair $(\ell, o) \notin \textit{Covered}(\zeta_l)$ as otherwise, the extensions α' and α'_0 obtained from α and α_0 by completing all *write* operations invoked on pairs in $\textit{Covered}(\zeta_l)$ will be indistinguishable to a subsequent reader causing it to return $v_l \neq v_{l+1}$ violating safety.

Let $w_1 = o_1.\textit{write}(\ell_1, \cdot)$ be the first *write* invocation in γ such that $(\ell_1, o_1) \notin \textit{Covered}(\zeta_l)$, and γ_1, w_1 be the prefix of γ ending with w_1. In addition, by construction, γ_1 does not include responses for all *write* invocations made in ζ_l. Therefore, $\alpha_1 = \alpha_0, \gamma_1, w_1$; and $\beta_1 = \beta_0, w_1, \textit{NR-crash}(o_1)$ satisfy the base case of Lemma 4.4. □

We now prove the inductive step of Lemma 4.4. Assume that $1 \leq i < t$. We use our inductive hypothesis for α_i, and β_i and show how to construct α_{i+1}, and β_{i+1} required by the lemma.

By wait freedom, there exists an execution fragment γ such that $\beta = \beta_i, \gamma, ack_{l+1}$ is an execution of A. Since β_i and α_i are indistinguishable to process $l+1$ (and both the effects of incomplete invocations and object responses can be indefinitely postponed), $\alpha = \alpha_i, \gamma, ack_{l+1}$ is also an execution of A. We prove the following

Lemma A.3 *At least* 1 *write invoked in* γ *must be on a pair* (ℓ, o) *such that* $o \notin \{o_j : 1 \leq j \leq i\}$ *and* $(\ell, o) \notin \textit{Covered}(\zeta_l)$.

Proof: Assume by contradiction that for each (ℓ, o) written in γ either $o \in \{o_j : 1 \leq j \leq i\}$ or $(\ell, o) \in \textit{Covered}(\zeta_l)$. Let α' be an extension of α where all *write* invocations on the pairs $(\ell', o') \in \textit{Covered}(\zeta_l)$, such that o' is correct in γ_i, complete. Note that after α', each base object o on which *write* was invoked in the course of W_{l+1} either has the written value overwritten by a *write* invoked before W_{l+1} started or has an incomplete *write* invocation.

Let ζ'_l be an extension of ζ_l obtained by completing all *write* invocations on the pairs $(\ell', o') \in \textit{Covered}(\zeta_l)$, and crashing all objects in $\{o_j : 1 \leq j \leq i\}$ (this is possible since ζ_l is failure-free, and $|\{o_j : 1 \leq j \leq i\}| = i < t$). We extend ζ'_l with an execution fragment ρ consisting of a complete READ invocation R. By safety the R's response must be v_l.

Next, we extend α' with ρ. By atomicity, all the base object invocations issued in the course of R are allowed to be serialized before all the incomplete *write* invocations occurring in the course of W_{l+1}. Given

this serialization, the state perceived by R will be identical to that perceived by R after ζ_l' causing it to return v_l. However, by safety, R's return value in α' must be $v_{l+1} \neq v_l$. A contradiction. □

The proof is by induction on $1 \leq i \leq t$. We start by proving the base case:

Proof: [Base Case of Lemma 4.4]

By wait freedom, there exists an execution fragment γ such that $\beta = \beta_0, \gamma, ack_{l+1}$ is an execution of A. Since β_0 and α_0 are indistinguishable to process $l+1$, and incomplete operations are allowed to take effect arbitrarily in the future, $\alpha = \alpha_0, \gamma, ack_{l+1}$ is also an execution of A.

Note that at least 1 *write* must be invoked in γ as otherwise, α_0 and α will be indistinguishable to a process invoking a READ operation R after α causing R to return $v_l \neq v_{l+1}$ violating safety. Moreover, at least one of the *write*'s in γ must be invoked on a pair $(\ell, o) \notin$ *Covered*(ζ_l) as otherwise, the extensions α' and α_0' obtained from α and α_0 by completing all *write* operations invoked on pairs in *Covered*(ζ_l) will be indistinguishable to a subsequent reader causing it to return $v_l \neq v_{l+1}$ violating safety.

Let $w_1 = o_1.write(\ell_1, \cdot)$ be the first *write* invocation in γ such that $(\ell_1, o_1) \notin$ *Covered*(ζ_l), and γ_1, w_1 be the prefix of γ ending with w_1. In addition, by construction, γ_1 does not include responses for all *write* invocations made in ζ_l. Therefore, $\alpha_1 = \alpha_0, \gamma_1, w_1$; and $\beta_1 = \beta_0, w_1,$ *NR-crash*(o_1) satisfy the base case of Lemma 4.4. □

We now prove the inductive step of Lemma 4.4. Assume that $1 \leq i < t$. We use our inductive hypothesis for α_i, and β_i and show how to construct α_{i+1}, and β_{i+1} required by the lemma.

By wait freedom, there exists an execution fragment γ such that $\beta = \beta_i, \gamma, ack_{l+1}$ is an execution of A. Since β_i and α_i are indistinguishable to process $l+1$ (and both the effects of incomplete invocations and object responses can be indefinitely postponed), $\alpha = \alpha_i, \gamma, ack_{l+1}$ is also an execution of A. We prove the following

Lemma A.4 *At least* 1 *write invoked in* γ *must be on a pair* (ℓ, o) *such that* $o \notin \{o_j : 1 \leq j \leq i\}$ *and* $(\ell, o) \notin$ *Covered*(ζ_l).

Proof: Assume by contradiction that for each (ℓ, o) written in γ either $o \in \{o_j : 1 \leq j \leq i\}$ or $(\ell, o) \in$ *Covered*(ζ_l). Let α' be an extension of α where all *write* invocations on the pairs $(\ell', o') \in$ *Covered*(ζ_l), such that o' is correct in γ_i, complete. Note that after α', each base object o on which *write* was invoked in the course of W_{l+1} either has the written value overwritten by a *write* invoked before W_{l+1} started or has an incomplete *write* invocation.

Let ζ_l' be an extension of ζ_l obtained by completing all *write* invocations on the pairs $(\ell', o') \in$ *Covered*(ζ_l), and crashing all objects in $\{o_j : 1 \leq j \leq i\}$ (this is possible since ζ_l is failure-free, and $|\{o_j : 1 \leq j \leq i\}| = i < t$). We extend ζ_l' with an execution fragment ρ consisting of a complete READ invocation R. By safety the R's response must be v_l.

Next, we extend α' with ρ. By atomicity, all the base object invocations issued in the course of R are allowed to be serialized before all the incomplete *write* invocations occurring in the course of W_{l+1}. Given this serialization, the state perceived by R will be identical to that perceived by R after ζ_l' causing it to return v_l. However, by safety, R's return value in α' must be $v_{l+1} \neq v_l$. A contradiction. □

We are now ready to complete the proof of Lemma 4.4.

Proof: [Lemma 4.4] Let $w_{i+1} = o_{i+1}.write(\ell_{i+1}, \cdot)_{l+1}$ be the first *write* whose existence is asserted by Lemma A.4. Let γ_{i+1}, w_{i+1} be the prefix of γ ending with w_{i+1}. By construction, γ_{i+1}, w_{i+1} is a valid

continuation of β_i. In addition, since α_i and β_i are indistinguishable to process $i+1$ (and the responses of all *write* invocations in α_i can be arbitrarily postponed) γ_{i+1}, w_{i+1} is also a valid continuation of α_i. We conclude that $\alpha_{i+1} = \alpha_i, \gamma_{i+1}, w_{i+1}$; and $\beta_{i+1} = \beta_i, \gamma_{i+1}, w_{i+1}, \textit{NR-crash}(o_{i+1})$ are executions of A.

By construction, both executions satisfy the claims 1 and 2 of the lemma. We now show that the claims 3, 4, and 5 also hold. Indeed, By Lemma A.4, $o_{i+1} \notin \{o_j : 1 \leq j \leq i\}$ and $(\ell_{i+1}, o_{i+1}) \notin \textit{Covered}(\zeta_l)$. Therefore, claims 3 and 4 are both satisfied for α_{i+1}, and β_{i+1}. Finally, since γ_{i+1} is derived from an execution fragment that follows the stop events for all processes in $\{1, \ldots, l\}$, γ_{i+1} does not include ack responses for any prior invocations of *write*. Hence, the lemma's claim 6 is also satisfied for both α_i and β_i. This concludes the proof of the induction step of Lemma 4.4. □

A.4 Proof of the Inductive Step of Lemma 4.2

We now complete the proof of the inductive step of Lemma 4.2.

Proof: Inductive step of Lemma 4.2 Let α_t, and β_t be the two executions of A whose existence is asserted by Lemma 4.4.

By wait freedom, there exists an execution fragment γ' such that $\beta = \beta_t, \gamma', ack_{l+1}$ is an execution of A. Since β_t and α_t are indistinguishable to process $l+1$ (and all incomplete writes are allowed to take effect arbitrarily far in the future as well as the responses to previously invoked writes can be arbitrarily postponed) γ', ack_{l+1} is also a valid continuation of α_t. Hence, $\alpha_t, \gamma', ack_{l+1} = \zeta_{l+1}$ for $\gamma = \gamma_1, w_1, \ldots, \gamma_t, w_t, \gamma'$ is an execution of A. It remains to show that ζ_{l+1} constructed in this fashion satisfies the lemma's requirements.

Indeed, the first three claims of Lemma 4.2 are satisfied by construction. Furthermore, by claims 3, 4, and 5 of Lemma 4.4, $|\textit{Covered}(\alpha_t)| = |\textit{Covered}(\zeta_l)| + t = (l+1)t$. Moreover, by construction, none of the *write* invocations made on pairs $(\ell, o) \in \textit{Covered}(\alpha_t)$ returns in γ'. Therefore, $|\textit{Covered}(\zeta_{l+1})| = |\textit{Covered}(\alpha_t)| = (l+1)t$. Hence, claim 4 of Lemma 4.2 is also satisfied. □

A.5 Proof of Lemma 4.5

Lemma A.5 *Let α be a finite failure-free execution of A in which all* WRITE *invocations are complete. Then, $|\{o : (\cdot, o) \notin \textit{Covered}(\alpha)\}| > 2t$.*

Proof: Assume by contradiction that $|\{(\ell, o) \notin \textit{Covered}(\alpha)\}| \leq 2t$. We proceed by constructing an extension of α violating safety thus obtaining a contradiction.

Let v be the value written by the last complete WRITE invocation in α if exists, or v_0, otherwise. We first extend α with a WRITE invocation $W = \text{WRITE}(v_1)_1$ by process 1 such that $v_1 \neq v$. This is possible since all WRITE invocations in α are complete, $k \geq 1$, and α is failure-free.

Next, we continue by extending α, W with an execution fragment γ obtained by repeating the following 2 steps in a loop for each $1 \leq i \leq t$ until either i reaches t, or W returns: (1) run A until the first occurrence of *write* invocation $w_i = o_i.\textit{write}(\ell_i, \cdot)_i$ such that $(\ell_i, o_i) \notin \textit{Covered}(\alpha)$; and (2) append $\textit{NR-crash}(o_1)$ to the resulting execution. If by the end of this procedure W is still in progress, continue executing A until W returns with ack. This is guaranteed to eventually happen since A is wait-free. Let $S_1 = \{o_i\}$.

Let S be the set of all snapshot objects $o' \in \{o : (\cdot, o) \notin \textit{Covered}(\alpha)\}$ such that $o'.\textit{write}$ is an event in α'. Note that $S_1 \subseteq S$. Let $S_2 = S \setminus S_1$. Let β be the subsequence of γ, which includes all events in γ omitting the crashes of all the objects in S_1. Since γ and β are indistinguishable to process 1, and object responses can be arbitrarily postponed, α, W, β, ack_1 is an execution of A. By assumption, $|S| \leq 2t$.

Let α_2 be an execution obtained by extending α with the following event sequence: (1) *NR-crash*(o) for all $o \in S_2$, and (2) completions of the *write* invocations on all $(\ell, o) \in$ *Covered*(α). We extend α_2 with an execution segment γ consisting of READ invocation R by a correct process. Since $|S_2| \leq t$, by wait freedom, R must eventually return. By safety, the return value of R must be v.

Finally, we obtain an execution α_1 by extending α, W, β, ack_1 with ack responses of the *write* invocations on all $(\ell, o) \in$ *Covered*(α). Note that by the end of α_1, all values written in the course of W in α_1 (i.e., in β) are either overwritten by prior writes, written by incomplete writes to the objects in S_1, or written to the objects in S_2. Since all incomplete writes are allowed to take effect indefinitely far in the future, the execution α_1', which is identical to α_1 except for all write invocations on objects in S_1 being removed, is a valid execution of A. Since all *list* operations invoked on objects in S_2 are allowed to return arbitrarily far in the future, γ is a valid continuation of α_1'. However, the return value of the read R in γ is $v \neq v_1$ violating safety. A contradiction. □

B Correctness of Algorithm 1

Definition B.1 (Timestamp of an operation) *Let o be a* READ *or* WRITE *operation. We define $ts(o)$, the timestamp of o, as follows: if o is a* READ *operation (resp. a* WRITE *operation), then $ts(o)$ is new.ts when its assignment completes in line 18 (resp. in line 13).*

Lemma B.2 (Timeline consistency) *Consider an object x and let ts and ts' be the value of $x.ts$ at time t and t' respectively. If $t' > t$ then $ts' \geq ts$.*

Proof: Assume by contradiction that that $ts' < ts$. This implies that some *CAS* operation has set object x to a value with lower timestamp. Notice that an object can only be changed by invoking *CAS* in line 41. For its first invocation, the condition in line 39 must hold. Then, *CAS* may be invoked again, but only if the condition in line 42 evaluates to false. In both cases, when $x.CAS(exp, new)$ is invoked, *exp.ts* $<$ *new.ts*. Contradiction follows from the fact that *CAS* changes x to *new* and does so only if the current value of x equals to *exp*. □

Lemma B.3 (Partial Order) *If o and o' are two* READ *or* WRITE *operations with associated timestamps $ts(o)$ and $ts(o')$, respectively, such that o completes before o' is invoked, then $ts(o) \leq ts(o')$. If o' is a* WRITE *operation then $ts(o) < ts(o')$.*

Proof: Since o completes and o' has an associated timestamp, procedure *update* returns during o and procedure *query* returns during o'. This implies that both o and o' receive responses from a majority of objects. Suppose that object x_k belongs to the intersection of the majority accessed by *rmw* during o and the majority accessed by *read* during o'. By the termination condition of the loop in *rmw* in line 42, some operation changed x_k to a value with timestamp at least $ts(o)$ before o completes. Since o' is invoked after o completes, and since by Lemma B.2 the timestap of x_k never decreases, the *read* that accesses x_k during o' returns a value with timestamp at least $ts(o)$ and assigns it to $x[k]$ in line 28. Hence, after *query* returns during o', $x[k] \geq ts(o)$.

There are two cases to consider. If o' is a READ operation, then by Definition B.1, $ts(o') = new.ts$ when the assignment in line 18 completes. Since *new* is set in line 18 to the value with the highest timestamp in x, which is at least $ts(o)$, it follows that $ts(o') \geq ts(o)$.

If o' is a WRITE operation, by Definition B.1, $ts(o') = new.ts$ after the assignment in line 13, where process i sets *new*.ts to $(x_{max}.ts.num + 1, i)$. Notice that $x_{max}.ts$ has been previously selected as the highest timestamp in x, which is at least $ts(o)$. Therefore, it follows that $ts(o') = (x_{max}.ts.num + 1, i) > x_{max}.ts \geq ts(o)$, which completes the proof. □

Lemma B.4 *Let o and o' are two* WRITE *operations with timestamps $ts(o)$ and $ts(o')$, respectively. Then $ts(o) \neq ts(o')$.*

Proof: If o and o' are executed by different processes, then the *pid* parts of their timestamps are different. If o and o' are executed by the same process, then they are executed sequentially and the proof follows from Lemma B.3. □

Lemma B.5 *Let r be a* READ *operation with timestamp $ts(r)$ returning value v. If $v \neq v_0$ then there exists a single* WRITE *operation w writing the value v s.t. $ts(r) = ts(w)$. Moreover, w does not follow r in the execution.*

Proof: Since r returns v and has an associated timestamp $ts(r)$, the *query()* method during r reads $(ts(r), v)$ from one of the objects. Since objects are changed only through *CAS* in line 41, we conclude that for some operation w, *new* in line 41 is equal to $(ts(r), v)$. Since *new* is only set once during the execution of a WRITE and that happens in line 13, we have that $ts(w) = ts(r)$. Moreover, by Lemma B.4 no other write has the same timestamp. Since w invokes *CAS* before r reads the same object, r does not complete before w is invoked, in other words, w does not follow r in the execution. □

Theorem B.6 (Atomicity) *Algorithm 1 implements an atomic RW register.*

Proof: Let σ be a trace produced by the algorithm. Note that the timestamp of a READ operation either has been written by some WRITE operation or is $(0, \perp)$, in which case the READ returns v_0 (Lemma B.5). We first construct σ' from σ by completing all WRITE operations of the form WRITE(v), where v has been returned by some complete READ operation. Then we construct a sequential permutation π by ordering all operations in σ' according to their timestamps and by placing all READ operations immediately after the WRITE operation with the same timestamp or in the beginning of the execution if the timestamp of the read is $(0, \perp)$. Note that concurrent READ operations with the same timestamp may appear in any order, whereas all other READ operations appear in the same order as in σ'.

To prove that π preserves the sequential specification of a MWMR register we must show that a READ always returns the value written by the closest write which appears before it in π, or the initial value of the register v_0 if there is no preceding write in π. Let r be a READ operation returning a value v. If $v = v_0$, then since the value and timestamp are always assigned atomically together in lines 13 and 18, we have that $ts(r) = (0, \perp)$, in which case or is ordered before any WRITE in π. Otherwise, $v \neq v_0$ and by Lemma B.5 there exists a *write*(v) operation, which has the same associated timestamp, $ts(r)$. In this case, this write is placed in π before r, by construction. By Lemma B.4, other write operations in π have a different associated timestamp and thus appear in π either before *write*(v) or after r.

It remains to show that π preserves real-time order. Consider two complete operations o and o' in σ' s.t. o precedes o'. By Lemma B.3, $ts(o') \geq ts(o)$. If $ts(o') > ts(o)$ then o' appears after o in π by construction. Otherwise $ts(o') = ts(o)$ and by Lemma B.3 it follows that o' is a READ operation. If o is a write operation, then o' appears after o since we placed each read after the WRITE with the same timestamp. Otherwise, if o is a READ, then it appears before o' as in σ'. □

Lemma B.7 *Let o be an operation that invokes rmw at time t and let o' be another operation that starts at time $t' \geq t$. If k operations are invoked but do not complete before time t then $ts(o').num \geq ts(o).num - k - 1$*

Proof: Let o'' be the operation with the highest timestamp that commits before time t. By Lemma B.3 $ts(o') \geq ts(o'')$. Therefore it is sufficient to prove that $ts(o'').num \geq ts(o).num - k - 1$. Suppose, for the purpose of contradiction, that $ts(o'').num = ts(o).num - k - 2$. Since every operation increments num by at most one and the timestamp of o is $ts(o)$, at least $k + 1$ operations must be invoked before time t with timestamps strictly greater than $ts(o).num - k - 2$. At least one of these operations commits before time

t by the statement of our Lemma. This is a contradiction since o'' was chosen to be the operation with the highest timestamp that commits before t. □

Lemma B.8 *Let o be an operation that invokes rmw at time t and o' be another operation that obstructs o on some object x but is not one of the first two operations to obstruct o on x. Then o' does not complete by time t.*

Proof: Since o is obstructed at least three times, the following sequence of invocations on x must occur (we denote an invocation of *op* on register r during an operation o by $o.r.op$): $o.x.read() \ldots o.x.CAS \ldots o.x.CAS \ldots o.x.CAS$. Since all three invocations of $o.x.CAS$ fail (the third one due to o'), we know that there are at least three invocations of $x.CAS$ by other operations that succeed: $o.x.read() \ldots o'''.x.CAS \ldots o.x.CAS \ldots o''.x.CAS \ldots o.x.CAS \ldots o'.x.CAS \ldots o.x.CAS$.

Since $o'.x.CAS$ succeeds, o' reads x (or invokes a failed *CAS* on x) after it was changed by $o''.x.CAS$, and hence after the first invocation of $o.x.CAS$, which in turn must occur after *rmw* is invoked during o, i.e., after time t. Hence, o' does not complete by time t. □

Lemma B.9 *Let $\mathcal{C}$ be the point contention of an operation o. For any constant n the number of operations o' that are concurrent with o and s.t. $ts(o').num = n$ is at most $\mathcal{C}$.*

Proof: Suppose for the purpose of contradiction that there exists a constant n such that there are $\mathcal{C}+1$ operations concurrent with o with the first component of their timestamp equal to n. Since there are $\mathcal{C}+1$ operations and at most $\mathcal{C}$ processes executing operations concurrently with o at any single point in time (by definition of point contention), there is a process that executes two operations, both of which have the same first component of the timestamp. Since each process executes operations sequentially, this contradicts Lemma B.3. □

Theorem B.10 (Step Complexity) *The step complexity of an operation o invoked by a correct process is $O(\mathcal{C}^2)$, where $\mathcal{C}$ is the point contention of o.*

Proof: Procedure *query* invokes *read* on every object and returns once it receives replies from a majority of objects. Since at most a minority of objects may fail, it follows that *query* returns. Let t be the time when o invokes *rmw*. There are three types of operations that can obstruct o: (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 o.

By Lemma B.8 at most two operation completing before time t can obstruct o on a given register. Thus, at most two operations fall into the first category. By definition of $\mathcal{C}$, the number of operations of the second type is at most $\mathcal{C}$. By Lemma B.7, this also implies that any operation o' of the third type, that is, starting at time t or later, satisfies $ts(o).num - ts(o').num \leq \mathcal{C}+1$. Since operations with timestamps higher than $ts(o)$ cannot obstruct o (see line 42), we only care about the case $0 \leq ts(o).num - ts(o').num$. There are at most $\mathcal{C}+2$ numbers in this range. Since all operations that start at time t or later and obstruct o are concurrent with o, by Lemma B.9 there are at most $\mathcal{C}$ such operations whose first timestamp component is each of the numbers in the range described above. Overall, there are at most $(\mathcal{C}+2) * \mathcal{C}$ operations with timestamps in this range, and in total there are $\mathcal{C}^2 + 3\mathcal{C} + 2$ operations that may obstruct o.

Notice that an operation o' can obstruct o on an object r only by changing the value of r using *CAS* on line 41. By the specification of *CAS*, the old value of r was the expected value passed to *CAS* in this invocation during o'. By the conditions on lines 42 and 45, once this *CAS* returns, *rmw* completes. This means that o' can obstruct o at most once. Since each operation can obstruct o at most once, $\mathcal{C}^2 + 3\mathcal{C} + 2$ is an upper bound on the number of times a CAS invocation during o can fail (for each object). The interaction with different objects during o is done using different instances of *rmw* executed concurrently. □