Space Complexity of Fault-Tolerant Register Emulations

Gregory Chockler*
Department of Computer Science
Royal Holloway, University of London
Egham, United Kingdom TW20 0EX
Gregory.Chockler@rhul.ac.uk

Alexander Spiegelman†
Viterbi Department of Electrical Engineering
Technion, Haifa, Israel
sashas@tx.technion.ac.il

ABSTRACT

Driven by the rising popularity of cloud storage, the costs associated with implementing reliable storage services from a collection of fault-prone servers have recently become an actively studied question. The well-known ABD result shows that an f-tolerant register can be emulated using a collection of $2f+1$ fault-prone servers each storing a single read-modify-write object, which is known to be optimal. In this paper we generalize this bound: we investigate the inherent space complexity of emulating reliable multi-writer registers as a function of the type of the base objects exposed by the underlying servers, the number of writers to the emulated register, the number of available servers, and the failure threshold.

We establish a sharp separation between registers, and both max-registers (the base object type assumed by ABD) and CAS in terms of the resources (i.e., the number of base objects of the respective types) required to support the emulation; we show that no such separation exists between max-registers and CAS. Our main technical contribution is lower and upper bounds on the resources required in case the underlying base objects are fault-prone read/write registers. We show that the number of required registers is directly proportional to the number of writers and inversely proportional to the number of servers.

CCS CONCEPTS

• **Theory of computation → Distributed algorithms**;

KEYWORDS

Faulty shared memory; reliable emulations; space complexity

*Gregory Chockler is supported in part by Royal Society International Exchanges and Coleman-Cohen Exchange Programme Grants.
†Alexander Spiegelman is grateful to the Azrieli Foundation for the award of an Azrieli Fellowship.

PODC '17, July 25-27, 2017, Washington, DC, USA

ACM ISBN 978-1-4503-4992-5/17/07…$15.00
https://doi.org/10.1145/3087801.3087824

ACM Reference format:
Gregory Chockler and Alexander Spiegelman. 2017. Space Complexity of Fault-Tolerant Register Emulations. In *Proceedings of PODC '17, Washington, DC, USA, July 25-27, 2017,* 10 pages.
https://doi.org/10.1145/3087801.3087824

1 INTRODUCTION

Reliable storage emulations seek to construct fault-tolerant shared objects, such as read/write registers, using a collection of *base objects* hosted on failure-prone servers. Such emulations are core enablers for many modern storage services and applications, including cloud-based online data stores [15, 26, 32–34] and Storage-as-a-Service offerings [17, 35, 38, 40].

Most existing storage emulation algorithms are constructed from storage services capable of supporting custom-built read-modify-write (RMW) primitives [3, 5, 16, 19, 19, 22, 23, 31, 36], and perhaps the most famous one is ABD [5]. This algorithm emulates an f-tolerant atomic wait-free register, accessed by an unbounded number of processes (readers and writers), on top of $2f + 1$ servers, each of which stores a single RMW object. Since f-tolerant register emulation is impossible with less than $2f + 1$ servers [8, 30], the ABD algorithm's space complexity is optimal.

However, support for atomic RMW is not always available: the operations exposed by network-attached disks are typically limited to basic reads and writes, and the interfaces exposed by cloud storage services sometimes augment this with simple conditional update primitives similar to Compare-and-Swap (CAS). A natural question that arises is therefore how the ABD results generalize when only weaker primitives (e.g., read/write registers) are available. More specifically, we are interested whether reliable storage emulations are possible with weaker primitives, and if so, what their space complexity is, and in particular, whether it depends on the number of writers and the number of servers.

To answer these questions, we consider a collection of n fault-prone servers, each of which stores base objects supporting the given primitives. The failure granularity is servers, meaning that a server crash causes all base objects it stores to crash as well. We study three primitives: read/write register, max-register [4], and CAS. For each primitive, we are interested in the number of base objects required to emulate an f-tolerant register for k writers using n servers.

Table 1 summarizes our results. To strengthen our lower bounds, we prove them under weak liveness and safety guarantees, namely, *obstruction freedom* and *write-sequential safety (WS-Safety)*. The latter is a weak generalization of Lamport's notion of safety [29] to multi-writer registers, which we define

Table 1: The number of base objects used by f-tolerant register emulations with k writers and $n > 2f$ servers.

Base object	Lower bound (WS-Safe, obstruction-free)	Upper bound (WS-Regular, wait-free)
max-register	$2f+1$	$2f+1$
CAS	$2f+1$	$2f+1$
read/write register	$kf + \left\lceil \frac{k}{\frac{n-(f+1)}{f}} \right\rceil (f+1)$	$kf + \lceil \frac{k}{\lfloor \frac{n-(f+1)}{f} \rfloor} \rceil (f+1)$

in Section 2. Since atomicity usually requires readers to write, which may induce a dependency on the number of readers, we consider regularity for our upper bound; we define in Section 2 *write-sequential regularity (WS-Regularity)*, which is a weaker form of multi-writer regularity defined in [36]. The lower bound of $n > 2f$ of servers required for f-tolerant register emulations [8, 30] can be easily generalized for WS-Safe obstruction-free emulations. Therefore, we assume that $n > 2f$ throughout the paper.

Interestingly, even though both read/write registers and max-registers have the lowest consensus number of 1 in Herlihy's hierarchy [24], we show that they are clearly separated with respect to their power to support a reliable multi-writer register in a space-efficient fashion. On the other hand, no such separation exists between CAS, which has an infinite consensus number, and max-register. As an aside, we note that this separation has implications for the standard shared memory model (without base object failures); for example, it implies that a k-writer max-register cannot be implemented from less than k read/write registers (proven in Theorem 3.2) closing the gap between the known lower and upper bounds of $k-1$ [28] and k [4] respectively.

Results. Despite the fact that the original ABD emulation [5] assumes a general RMW base object on every server, we observe that the code executed by each server in the multi-writer ABD protocol [23, 31, 36] can be encapsulated into the *write-max* (for handling update messages) and *read-max* (for handling read messages) primitives of max-register. Therefore, the upper bound of $n = 2f+1$ applies to max-registers as well. In Appendix A we show how to emulate a max-register from a single CAS in a wait-free manner. Thus, the upper bound for max-register also applies to CAS.

Our main technical contribution is a lower bound on the number of read/write registers required to emulate an f-tolerant WS-Safe obstruction-free register for k clients using n servers. While the ABD [5] space complexity does not depend on the number of writers or the number of servers, we show in Section 3 (Theorem 3.1) that when servers support only read/write registers, the lower bound increases linearly with the number of writers and decreases (up to a certain point) with the number of available servers. In particular, our lower bound implies that at least $kf + f + 1$ registers are needed regardless of the number of available servers. We exploit asynchrony to construct a write-sequential failure-free run in which each write completes while leaving f pending writes on base registers, forcing the next write to use a different set of registers, and so on.

In Section 4, we present a new upper bound construction that closely matches our lower bound (Theorem 4.1). Note that the two bounds are closely aligned, and in particular, coincide in the two important cases of $n = 2f+1$ and $n \geq kf + f + 1$ where they are equal to $kf + k(f+1)$ and $kf + f + 1$ respectively. An interesting open question is to close the remaining small gap.

Another open question is whether our lower bound is tight for stronger regularity definitions [36]. In the special case of $n = 2f+1$ servers and k writers, a matching upper bound of $(2f+1)k$ registers can be achieved by simply having each server implement a single k-writer max-register from k base registers [4]. The question of the general case of $n \geq 2f+1$ remains open.

In the full paper [14], we prove the following three additional results implied by an extended variant of our main lower bound construction: (1) a lower bound of k registers per server for $n = 2f+1$ (Theorem 3.4); (2) a lower bound on the number of servers when the maximum number of registers stored on each server is bounded by a known constant (Theorem 3.5); and (3) impossibility of constructing fault-tolerant multi-writer register emulations adaptive to point contention [1, 7] (Theorem 3.6).

Related work. The space complexity of fault-tolerant register emulations has been explored in a number of prior works. Aguilera et al. [2] show that certain types of multi-writer registers cannot be reliably emulated from a fixed number of fault-prone ones if the number of writers is not a priori known. They however, do not provide precise bounds on the number of base registers as a function of the number of writers and servers, and the failure threshold as we do in this paper. The space complexity of reliable register emulations in terms of the amount of data stored on fault-prone RMW servers was studied in [13, 20], and more recently in [11, 12, 39]. Since we are only interested in the number of stored registers and not their sizes, these results are orthogonal to ours.

Basescu et al. [9] describe several fault-tolerant multi-writer register emulations from a collection of fault-prone read/write data stores. Their algorithms incorporate a garbage-collection mechanism that ensures that the storage cost is adaptive to the write concurrency, provided that the underlying servers can be accessed in a synchronous fashion. Our results show that asynchrony has a profound impact on storage consumption by exhibiting a *sequential* failure-free run where the

number of registers that need to be stored grows linearly with the number of writers.

The proof of our main result (see Lemma 3.7) further extends the adversarial framework of [39] to exploit the notion of *register covering* (originally due to [10]) extended to fault-prone base registers as in [2]. Covering arguments have been successfully applied to proving numerous space lower bound results in the literature (see [6] for a survey) including the recent tight bounds for obstruction-free consensus [18, 21, 41].

2 PRELIMINARIES

2.1 Shared Objects

A *shared object* supports concurrent execution of *operations* performed by some set, $\mathbb{C} = \{c_1, c_2, \dots\}$, of client processes. Each operation has an *invocation* and *response*. An object *schedule* is a sequence of the operation invocations and their responses. An invoked operation is *complete* in a given schedule if the operation's response is also present in the schedule, and *pending* otherwise. For a schedule σ, $ops(\sigma)$ denotes the set of all operations that were invoked in σ, and $complete(\sigma)$ (resp., $pending(\sigma)$) denotes the subset of $ops(\sigma)$ consisting of all the complete (resp., pending) operations. Also, for a set X of operations, we use $\sigma|X$ to denote the subsequence of σ consisting of all the invocation and responses of the operations in X.

An operation *op precedes* an operation op' in a schedule σ, denoted $op \prec_\sigma op'$, iff op' is invoked after op responds in σ. Operations op and op' are *concurrent* in σ, if neither one precedes the other. A schedule with no concurrent operations is *sequential*. Given a schedule σ, we use $\sigma|i$ to denote the subsequence of σ consisting of all the actions executed by the client c_i. The schedule is *well-formed* if $\sigma|i$ is sequential for all $i > 0$. In the following, we will only consider well-formed schedules.

The object's *sequential specification* is a collection of the object's sequential schedules in which all operations are complete. For an object schedule σ, a *linearization* L_σ of σ is a sequential schedule consisting of all operations in $complete(\sigma)$ along with their responses and a subset of $pending(\sigma)$, each of which being assigned a matching response, so that L_σ satisfies both the σ's operation precedence relation ($\prec_\sigma$) and the object sequential specification.

Consistency conditions specify the shared object behaviour when accessed concurrently by the clients. They are expressed as a set of schedules satisfying a desired property. The basic consistency condition for shared objects of any type is *atomicity* [25] defined formally as follows:

Definition 2.1 (Atomicity). A set of schedules C satisfies atomicity if for all schedules $\sigma \in C$, σ has a linearization.

2.2 Object types

Registers. A *read/write register* object (or simply a *register*) supports two operations of the form *write*(v), $v \in \mathbb{V}$, and *read*() returning *ack* and $v \in \mathbb{V}$ respectively where $\mathbb{V}$ is the register value domain. Its sequential specification consists of all sequential schedules in which every *read* returns the value written by the last preceding *write* or an initial value $v_0 \in \mathbb{V}$ if no such write exists.

A register is *multi-writer (MW)* (resp., *multi-reader (MR)*) if it can be written (resp., read) by an *unbounded* number of clients. A k-writer register, or simply, k-register, is a register that can be written by at most $k > 0$ distinct clients. A register is *single-writer (SW)* (resp., *single-reader (SR)*) if only one process can write (resp., read) the register. For a register schedule σ, we write $writes(\sigma)$ (resp., $reads(\sigma)$) to denote the sets of all write (resp., read) operations invoked in σ. We say that σ is *write-sequential* if no two writes in $writes(\sigma)$ are concurrent.

Let C be a set of register schedules. The following weaker consistency conditions will be considered for registers in addition to atomicity:

Definition 2.2 (Write-Sequential Regularity (WS-Regularity)). For all $\sigma \in C$, if σ is write-sequential, then for each $rd \in reads(\sigma) \cap complete(\sigma)$ there is a linearization L_{rd} of $\sigma|writes(\sigma) \cup \{rd\}$.

Definition 2.3 (Write-Sequential Safety (WS-Safety)). As WS-Regularity, but only required to hold for complete reads that are not concurrent with any writes.

Max-registers. Given an ordered set of values $\mathbb{V}$, a *max-register* [4] supports two operations: *write-max*(v), $v \in \mathbb{V}$, that returns *ok*, and *read-max* that returns $v \in \mathbb{V}$. Its sequential specification consists of all sequential schedules in which every *read-max* returns the highest value written by a preceding *write-max*, or an initial value $v_0 \in \mathbb{V}$ if no such *write-max* exists.

Compare-and-Swap (CAS). A CAS object supports a single operation $C\&S(v, v')$, $v, v' \in \mathbb{V}$, and returns $v'' \in \mathbb{V}$ where $\mathbb{V}$ is a value domain. Its sequential specification consists of all sequential schedules in which every $C\&S(v, v')$ operation returns the current object value (which is initialized to v_0), and sets it to v' if it equals v.

2.3 System Model

We consider an *asynchronous fault-prone shared memory system* [27] consisting of a set of shared *base* objects $\mathcal{B} = \{b_1, b_2, \dots\}$. The objects are accessed by a collection of clients in the set $\mathbb{C} = \{c_1, c_2, \dots\}$.

We consider a slight generalization of the model in [27] where the objects are mapped to a set $\mathcal{S} = \{s_1, s_2, \dots s_n\}$ of n 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., $\delta^{-1}(S) = \{b : \delta(b) \in S\}$. Note that for all $B \subseteq \mathcal{B}$, $|\delta(B)| \leq |B|$, and conversely, for all $S \subseteq \mathcal{S}$, $|\delta^{-1}(S)| \geq |S|$. Both servers and clients can fail by crashing. A crash of a server causes all objects mapped to that server to instantaneously crash (i.e.,, stop responding to the client invocations)[1].

[1] Note that the original faulty shared model of [27] can be derived from our model by choosing δ to be an injective function.

Emulation algorithms. We study algorithms emulating a reliable k-register to a set of clients from a collection of fault-prone atomic base objects. 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 invocations, 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, we assume that the clients can trigger several operations in a row without waiting for the previously triggered operations to respond.

An emulation algorithm A defines the behaviour 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.

Runs. A *run* of 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 crashes at some time in r, and correct, otherwise.

Fairness. 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.

2.4 Properties of Emulation Algorithms

Safety The emulation algorithm safety will be expressed in terms of the write-sequential consistency conditions as given by Definitions 2.2 and 2.3. An emulation algorithm A *satisfies* a consistency condition C if for all runs r of A, the subsequence of r consisting of the invocations and responses of the high-level read and write operations is a schedule in C.

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. An *obstruction-free* object guarantees that every high-level operation invoked by a correct client that is not concurrent to any other operation by a correct client eventually returns.

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* [27] of A is the maximum resource consumption of A in all its runs.

3 LOWER BOUNDS

We characterize the minimum resource complexity of the algorithms implementing an f-tolerant obstruction-free WS-Safe k-register as a function of the number n of available servers. First, it is easy to see that if $n \leq 2f$, then no such algorithm can exist. This result is implied by an extended statement of Lemma 3.7 (proven in the full paper [14]), and can also be shown directly by a straightforward application of a partitioning argument as discussed in [8, 30]. For the case $n > 2f$, we prove the following main theorem:

THEOREM 3.1. *For all $k > 0$, $f > 0$, let A be an f-tolerant algorithm emulating an obstruction-free WS-Safe k-register using a collection $\mathcal{S}$ of servers such that $n \geq 2f + 1$. Then, A uses at least $kf + \left\lceil \frac{kf}{n-(f+1)} \right\rceil (f + 1)$ base registers (i.e., $|\delta^{-1}(\mathcal{S})| \geq kf + \left\lceil \frac{kf}{n-(f+1)} \right\rceil (f + 1)$).*

This result implies a new lower bound of k registers on the resource complexity of emulating a single (i.e., non-fault-tolerant) max-register for k writers that tightens the previously known lower bound of $k - 1$ [28]:

THEOREM 3.2 (RESOURCE COMPLEXITY OF k-WRITER MAX-REGISTER). *For all $k > 0$, any algorithm implementing a wait-free k-writer max-register from a collection of wait-free MWMR atomic base registers uses at least k base registers.*

PROOF (SKETCH). Observe that an f-tolerant obstruction-free WS-Safe k-register emulation A can be obtained (using an ABD-style algorithm) from $n = 2f+1$ servers each storing a single max-register object. It therefore, follows that A's resource complexity using max-registers implemented from $< k$ registers will be $< (2f + 1)k$ contradicting Theorem 3.1. The detailed proof appears in the full paper [14]. □

Since a k-writer max-register can be implemented from k read/write registers [4], we have the following

COROLLARY 3.3. *k wait-free multi-writer atomic registers are necessary and sufficient to implement a wait-free k-writer max-register in the standard shared memory model.*

The following lower bounds are proven in the full paper [14]:

THEOREM 3.4. *Let $n = 2f+1$. For all $k > 0$, $f > 0$, every f-tolerant algorithm emulating an obstruction-free WS-Safe k-register stores at least k registers on each server in $\mathcal{S}$.*

THEOREM 3.5. *Let $m > 0$ be an upper bound on the number of registers mapped to each server in $\mathcal{S}$ (i.e., $\forall s \in \mathcal{S}, |\delta^{-1}(\{s\})| \leq m$). For all $f > 0$ and $k > 0$, every f-tolerant algorithm emulating an obstruction-free WS-Safe k-register from a collection $\mathcal{S}$ of servers such that $n > 2f + 1$ uses at least $\lceil kf/m \rceil + f + 1$ servers (i.e., $n \geq \lceil kf/m \rceil + f + 1$).*

THEOREM 3.6. *For any $f > 0$, there is no f-tolerant algorithm that emulates an WS-Safe obstruction-free k-register with resource complexity adaptive to point contention [1, 7].*

We now present the proof of our main result.

3.1 Proof of Theorem 3.1

Overview. Our proof exploits the fact that the environment is allowed to prevent a pending low-level write from taking effect for arbitrarily long [2]. As a result, a client executing a high-level WRITE operation cannot reliably store the requested value in a base register that has a pending write as this write may take effect at a later time thus erasing the stored value. At the same time, the client cannot wait for all base registers on which it triggers low-level operations to respond, since up to f of them may reside on faulty servers. It therefore must be able to complete a high-level write without waiting for responses from up to f registers. Consequently, the next high-level WRITE (by a different client) cannot reliably use these registers (as they might have outstanding low-level writes), and is therefore forced to use additional registers thus causing the total number of registers grow with each subsequent WRITE.

In our main lemma (Lemma 3.7), we formalize this intuition as follows: Starting from a run r_0 consisting of an initial configuration, we build a sequence of consecutive extensions $r_1, \ldots, r_k$ so that r_i is obtained from r_{i-1} by having a new client invoke a high-level WRITE W_i of some (not previously used) value. We then let the environment behave in an adversarial fashion (Definition 3.10) by blocking the responses from the writes triggered on at most f base registers as well as the prior pending low-level writes. In Lemma 3.13, we show that W_i must terminate without waiting for these responses to arrive. Furthermore, in Lemma 3.14, we show that W_i must invoke low-level writes on at least $2f + 1$ base registers (residing on $\geq 2f + 1$ different servers) that do not have any prior pending writes. This, combined with Lemma 3.13, implies that by the time W_i completes, there are at least f more registers on at least f servers with pending writes after W_i completes. Thus, by the time the kth high-level WRITE completes, the total number of covered registers is at least kf (see Lemma 3.7(a)).

To obtain a stronger bound, our construction is parameterized by an *arbitrary* subset F of servers such that $|F| = f+1$. We show that the extra storage available on these servers cannot in fact, be utilized by an emulation (see Lemma 3.7(b)) forcing it to use at least kf registers on the remaining $\mathcal{S} \setminus F$ servers to accommodate the same number k of writers. We use this result in the proof of Theorem 3.1, to show that the number of base registers required for the emulation is a function of k and n.

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

- *Covering write*: Let w be a low-level write triggered on a base register b at times $\leq t$. We will refer to w as *covering at* t, and to b as being *covered by* w *at* t if w is pending at t.
- $C(t) \subseteq \mathbb{C}$: the set of all clients that have completed a high-level WRITE operation at times $\leq t$.
- $Cov(t) \subseteq \mathcal{B}$: the set of all base registers being covered by some low-level write at time t.
- $V(t) \subseteq \mathbb{V}$: the set of all values written by high-level WRITEs at times $\leq t$.

We first prove the following key lemma:

LEMMA 3.7. *For all $k > 0$, $f > 0$, let A be an f-tolerant algorithm that emulates a WS-Safe obstruction-free k-register using a collection $\mathcal{S}$ of servers storing a collection $\mathcal{B}$ of wait-free MWMR atomic registers. Then, for every $F \subseteq S$ such that $|F| = f + 1$, there exist $k + 1$ failure-free runs r_i, $0 \leq i \leq k$, of A such that (1) r_0 is a run consisting of an initial configuration and $t_0 = 0$ steps, and (2) for all $i \in [k]$, r_i is a write-only sequential extension of r_{i-1} ending at time $t_i > 0$ that consists of i complete high-level WRITEs of i distinct values $v_1, \ldots, v_i$ by i distinct clients $c_1, \ldots, c_i$ such that:*

a) $|Cov(t_i)| \geq if$ *b)* $\delta(Cov(t_i)) \cap F = \emptyset$

Fix arbitrary $k > 0$, $f > 0$, and a set F of servers such that $|F| = f + 1$. We proceed by induction on i, $0 \leq i \leq k$. **Base:** Trivially holds for the run r_0 of A consisting of $t_0 = 0$ steps. **Step:** Assume that r_{i-1} exists for all $i \in [k-1]$. 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 r_i. For the remainder of the Lemma 3.7's proof, we will limit our attention to the runs in which every low-level write operation is linearized simultaneously with its respond step. In particular, this implies that no low-level write w that is covering some register b at a time t in r will be observed by any low-level reads from b as having taken effect until after w's respond event occurs. Formally:

ASSUMPTION 1 (WRITE LINEARIZATION). *For every extension r of r_{i-1} and a base object $b \in \mathcal{B}$, let $L_{r|b}$ be a linearization of $r|b$. Then, $L_{r|b}$ does not include any low-level write operations in pending$(r|b)$, and for any two low-level writes $w_1, w_2 \in$ complete$(r|b)$ such that respond$(w_1) \prec_{r|b}$ respond(w_2), $w_1 \prec_{L_{r|b}} w_2$.*

Note that the above does not affect generality of our lower bound since the stipulated base register behaviour is allowed by atomicity, and therefore, must be tolerated by *every* emulation algorithm.

We proceed by introducing the following notation:

Definition 3.8. Let r be an extension of r_{i-1}. For all times $t \geq t_{i-1}$ in r, let

(1) $Tr_i(t)$: the set of all base registers which have a low-level write triggered on between t_{i-1} and t.
(2) $Rr_i(t) \subseteq Tr_i(t)$ be the set of all base registers which had a low-level write triggered on and responded (took effect) between t_{i-1} and t.

(3) $Cov_i(t) = Cov(t) \setminus Cov(t_{i-1})$ be the set of all base registers that have been newly covered between t_{i-1} and t. Note that $Cov_i(t) \subseteq Tr_i(t)$.
(4) $Q_i(t) \subseteq \mathcal{S}$ be the set of all 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. In other words, $Q_i(t)$ is the set of the first f servers not in F that have at least one register newly covered between t_{i-1} and t.
(5) $F_i(t) \triangleq \{s \in F \mid \delta^{-1}(\{s\}) \cap Rr_i(t) \neq \emptyset\}$, i.e., $F_i(t)$ is the set of all servers in F having a register that responded to a low-level write invoked after t_{i-1}.
(6) $M_i(t) \triangleq \delta(Cov_i(t)) \cap (F \setminus F_i(t))$, i.e., $M_i(t)$ is the set of all servers in F with at least one register covered by a low-level write invoked after t_{i-1} and without registers that have responded to the low-level writes invoked after t_{i-1}.
(7) $G_i(t) \subseteq \mathcal{S}$ be the set of all servers such that $G_i(t) = M_i(t)$ if $|Q_i(t)| < |F_i(t)|$, and $G_i(t) = \emptyset$, otherwise.

Below we will introduce the adversary Ad_i, which causes A to gradually increase the number of base registers covered after t_{i-1} by delaying the respond actions of some of the previously triggered low-level writes.

Definition 3.9 (Blocked Writes). Let r be an extension of r_{i-1}. For all times $t \geq t_{i-1}$ in r, let $\text{BlockedWrites}_i(t)$ be the set of all low-level covering writes w satisfying either one of the following two conditions:

(1) w was triggered by a client in $C(t_{i-1})$, or
(2) w was triggered on a base register in $\delta^{-1}(Q_i(t) \cup G_i(t))$.

We say that a pending low-level write w is *blocked* in an extension r of r_{i-1} if there exists a time $t \geq t_{i-1}$ such that for all $t' > t$ in r, $w \in \text{BlockedWrites}_i(t')$. The following definition specifies the set of the environment behaviours that are allowed by Ad_i in all extensions of r_{i-1}:

Definition 3.10 (Ad_i). For an extension r of r_{i-1} we say that the environment *behaves like* Ad_i after r_{i-1} in r if the following holds:

(1) There are no failures after time t_{i-1} in r.
(2) For all $t \geq t_{i-1}$ in r, if a low-level write $w \in \text{BlockedWrites}_i(t)$, then w does not respond at t.
(3) If r is infinite then:
   (a) Every pending low-level read or write that is not blocked in r eventually responds.
   (b) Every trigger or return action that is ready to be executed at a client c in r is eventually executed.

For a finite extension r of r_{i-1}, we will write $\langle r, Ad_i \rangle$ to denote the set of all extensions of r in which the environment behaves like Ad_i after r_{i-1}; and we will write $\langle r, Ad_i, t \rangle$ to denote the subset of $\langle r, Ad_i \rangle$ consisting of all runs having exactly t steps. For $X \in \{Q_i, F_i, M_i\}$ and a run $r \in \langle r_{i-1}, Ad_i, t \rangle$, we say that X is *stable* after r if for all $t' \geq t$ for all extensions $r' \in \langle r, Ad_i, t' \rangle$, $X(t') = X(t)$.

The following lemma asserts several facts implied directly by Definitions 3.8 and 3.10. The proof is very technical and for space limitation appears in the full paper [14].

LEMMA 3.11. *For all $t \geq t_{i-1}$ and $r \in \langle r_{i-1}, Ad_i, t \rangle$, all of the following holds at time t in r:*

(1) $Q_i(t) \subseteq \delta(Cov_i(t)) \setminus F$
(2) $Q_i(t) \subseteq Q_i(t+1)$
(3) $F_i(t) \subseteq F_i(t+1)$
(4) $|F_i(t)| - |Q_i(t)| \leq 1$
(5) $|Q_i(t)| \leq f$
(6) $|F_i(t)| \leq f + 1$
(7) $F_i(t) = F_i(t+1) \implies M_i(t) \subseteq M_i(t+1)$
(8) $|M_i(t)| \leq f + 1$
(9) $|\delta(Cov_i(t)) \setminus F| \geq f \implies |Q_i(t)| \geq f$
(10) $|\delta(Cov_i(t)) \setminus F| < f \implies \delta(Rr_i(t)) \setminus F = \emptyset$
(11) $(Q_i(t) \cup M_i(t)) \cap \delta^{-1}(Rr_i(t)) = \emptyset$

The following corollary follows immediately from the claims 2–3 and 5–8 of Lemma 3.11.

COROLLARY 3.12. *There exists a run $r \in \langle r_{i-1}, Ad_i \rangle$ such that F_i, Q_i, and M_i are all stable after r.*

We first show that r_{i-1} can be extended with a complete high-level WRITE W_i of a value $v_i \notin V(t_{i-1})$ by a client $c_i \notin \{c_1, \ldots, c_{i-1}\}$ such that the environment behaves like Ad_i until W_i returns. Roughly, the reason for this is that Ad_i guarantees that after r_{i-1}, c_i would only miss responses from the writes invoked on at most f servers (see Claim 1) as well as those that might have been invoked in r_{i-1} by other clients $\{c_1, \ldots, c_{i-1}\}$, which c_i is unaware of. As a result, the involved servers and clients would appear to c_i as faulty after r_{i-1}, and therefore, to ensure obstruction freedom, it must complete W_i without waiting for the outstanding writes to respond.

LEMMA 3.13. *Let $r \in \langle r_{i-1}, Ad_i \rangle$ be a run consisting of r_{i-1} followed by a high-level WRITE invocation W_i by client $c_i \notin C(t_{i-1})$. Then, there exists a run $r_r \in \langle r, Ad_i \rangle$ in which W_i returns.*

By Corollary 3.12, there exists an extension $r' \in \langle r, Ad_i, t' \rangle$ where $t' > t_{i-1}$ such that Q_i, F_i, and M_i are all stable after r'. If W_i returns in r', we are done. Otherwise, we will first bound the number of servers $|Q_i(t') \cup M_i(t')|$ controlled by Ad_i as follows:

CLAIM 1. *Consider a time $t > t_{i-1}$, and a run $r \in \langle r_{i-1}, Ad_i, t \rangle$. If M_i is stable after r, then $|Q_i(t) \cup M_i(t)| \leq f$.*

PROOF. By Lemma 3.11.5, $|Q_i(t)| \leq f$. Thus, if $M_i(t) = \emptyset$, then $|Q_i(t) \cup M_i(t)| \leq f$ as needed. Otherwise ($M_i(t) \neq \emptyset$), we show that $|F_i(t)| = |Q_i(t)| + 1$. Suppose to the contrary that $|F_i(t)| \neq |Q_i(t)| + 1$. Since by Lemma 3.11.4, $|F_i(t)| \leq |Q_i(t)| + 1$, the only possibility for $|F_i(t)| \neq |Q_i(t)| + 1$ is if $|F_i(t)| \leq |Q_i(t)|$. Thus, by Definition 3.8.7, $G_i(t) = \emptyset$, and hence, by Definition 3.9, no writes on the registers in $\delta^{-1}(M_i(t))$ are blocked. However, since $M_i(t) \neq \emptyset$, at least one register in $\delta^{-1}(M_i(t))$ must have an outstanding write. Therefore, by Definition 3.10.3(a), there exists time t', and an extension $r' \in \langle r, Ad_i, t' \rangle$ such that one of the registers on some server $s \in M_i(t)$ responds at time t'. Thus, $s \in F_i(t')$, and therefore, $s \notin M_i(t')$. Hence, M_i is not stable after r. A contradiction to the assumption.

Since $F_i(t) \subseteq F$, $|F \setminus F_i(t)| + |F_i(t)| = |F| = f + 1$. Hence, $|F \setminus F_i(t)| = f + 1 - |F_i(t)| = f - |Q_i(t)|$. Since by Definition 3.8.6, $M_i(t) \subseteq (F \setminus F_i(t))$, $|M_i(t)| \leq |F \setminus F_i(t)| = f - |Q_i(t)|$. Thus, we receive $|Q_i(t) \cup M_i(t)| \leq |Q_i(t)| + |M_i(t)| \leq f$ as needed. □

We are now ready to complete the proof of Lemma 3.13:

Proof of Lemma 3.13. By Claim 1, $|Q_i(t') \cup M_i(t')| \leq f$, and by Lemma 3.11.11, no base registers on servers in $Q_i(t') \cup M_i(t')$ have ever responded to any low-level writes issued after t_{i-1}. Thus, there exists a finite run r'', which is identical to r' except all servers in $Q_i(t') \cup M_i(t')$ crash immediately after r and each client $c_1, \ldots, c_{i-1}$ fails before any of its covering writes on registers in $Cov(t_{i-1})$ responds. By f-tolerance and obstruction freedom, there exists a fair extension $r''\sigma$ of r'' (i.e., $r''\sigma \notin \langle r'', Ad_i \rangle$) such that W_i returns in $r''\sigma$. Since $Q_i \cup M_i$ is stable after r', the set of registers precluded by Ad_i from responding in r' is identical to that in r'', and by Assumption 1, no write with a missing response is linearized, r' is indistinguishable from r'' to c_i. Thus, $r'\sigma \in \langle r, Ad_i \rangle$, and since σ includes the return event of W_i, $r'\sigma$ satisfies the lemma.

□

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

Lemma 3.14. *Consider a run $r \in \langle r_{i-1}, Ad_i, t_r \rangle$ where $t_r > t_{i-1}$, consisting of r_{i-1} followed by a complete high-level* write *invocation $W_i =$ write(v_i), $v_i \notin V(t_{i-1})$, by client $c_i \notin C(t_{i-1})$ that returns at time t_r. Then, $|\delta(Tr_i(t_r) \setminus Cov(t_{i-1}))| > 2f$.*

Proof. Denote $X \triangleq \delta(Tr_i(t_r) \setminus Cov(t_{i-1}))$, and assume by contradiction that $|X| \leq 2f$. Let $S_1 = F_i(t_r)$, $S_2 = Q_i(t_r)$, $S_3 = X \cap (F \setminus F_i(t_r))$ and $S_4 = X \setminus (S_1 \cup S_2 \cup S_3)$. Note that S_1, S_2, S_3, S_4 are pairwise disjoint, and $X = S_1 \cup S_2 \cup S_3 \cup S_4$.

We first show that $|S_1 + S_4| \leq f$. By Lemma 3.11.6, $|S_1| \leq f + 1$. However, if $|S_1| = f + 1$, then by Lemma 3.11.4, $|S_1| - |S_2| = f + 1 - |S_2| \leq 1$, and therefore, $|S_1| + |S_2| \geq 2f + 1$ violating the assumption. Hence, $|S_1| \leq f$. By Lemma 3.11.5, $|S_2| \leq f$. If $|S_2| = f$, then by assumption, $|S_1 \cup S_3 \cup S_4| = |S_1| + |S_3| + |S_4| \leq f$, and therefore, $|S_1 + S_4| = |S_1| + |S_4| \leq f$. And if $|S_2| < f$, then by Definitions 3.8.4 and 3.10, $|S_4| = 0$. Hence, $|S_1 + S_4| = |S_1| + |S_4| \leq f$.

The proof proceeds by applying the partitioning argument to the sets S_i, $i \in [4]$, as illustrated in Figure 1. let r' be a fair extension of r_{i-1} consisting of t'_c steps in which r_{i-1} is followed by (1) the crash events of all servers in $S_1 \cup S_4$, and (2) the respond steps of all the covering writes in r_{i-1} and (and no other steps). Extend r' with an invocation of a high-level read operation R by client $c_{rd} \neq c_i$ at time t'_c. Since $|S_1 + S_4| \leq f$, by obstruction freedom and f-tolerance, there exists time $t_{rd} > t_{i-1}$ at which R returns. Since r' is write-sequential, by WS-Safety, R must return v_{i-1}.

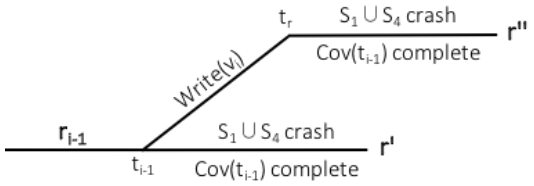

Figure 1: Construction for the proof of Lemma 3.14.

Next, let r'' be an extension of r consisting of all steps in r up to the time t_r followed by (1) the crash events of all servers in the set $S_1 \cup S_4$, and (2) the respond steps of all covering writes in r_{i-1} (and no other steps). Let $t''_c > t_r$ be the number of steps in r''. By Assumption 1, the values that can be read from the base registers in $Cov(t_{i-1})$ at time t''_c in r'' are identical to those that can be read at time t'_c in r'. Furthermore, by definitions 3.8.5 and 3.10, low-level writes triggered on registers in $\delta^{-1}(S_2 \cup S_3)$ do not respond before t_r in r. Thus, by Assumption 1, the values that can be read from the base registers in $\delta^{-1}(S_2 \cup S_3)$ at time t'_c in r' are also the same as those that can be read at time t''_c in r''. Thus, all registers in non-faulty servers at time t'_c in r' will appear to the subsequent reads as having the same content as at the time t''_c in r''_c.

We now extend r'' by letting client c_{rd} invoke high-level read R at time t''_c. Since r' is indistinguishable from r'' to c_{rd}, and R has no concurrent high-level operations, we get that R returns v_{i-1} in r''. However, since W_i is the last complete write preceding R in r'', by WS-Safety, the R's return value must be $v_i \neq v_{i-1}$. A contradiction.

□

The following corollary follows immediately from Lemma 3.14, Definitions 3.8.4 and 3.10, and the choice of $|F| = f + 1$:

Corollary 3.15. *Consider a run $r \in \langle r_{i-1}, Ad_i, t_r \rangle$ where $t_r > t_{i-1}$, consisting of r_{i-1} followed by a complete high-level* write *invocation $W_i =$ write(v_i), $v_i \notin V(t_{i-1})$, by client $c_i \notin C(t_{i-1})$ that returns at time t_r. Then, $|Q_i(t_r)| = f$.*

We are now ready to complete the proof of the induction step of Lemma 3.7:

Proof of the induction step (Lemma 3.7).
By Lemma 3.13, there exists a run $r \in \langle r_{i-1}, Ad_i, t_r \rangle$, $t_r > t_{i-1}$, in which r_{i-1} is followed by a complete high-level write invocation W_i by client $c_i \neq c_{i-1}$ writing a value $v_i \notin V(t_{i-1})$ and returning at time t_r. By Corollary 3.15, $|Q_i(t_r)| = f$, and therefore, by Lemma 3.11.4, $|F_i(t_r)| = f + 1$. Since $F_i(t_r) \subseteq F$ and $|F| = f + 1$, we conclude that $F_i(t_r) = F$. Hence, by Definition 3.8.6, $M_i(t_r) = \emptyset$, which by Definition 3.8.7, implies that $G_i = \emptyset$. Thus, by Definition 3.9, no writes on the registers in $\delta^{-1}(F)$ triggered after t_{i-1} are blocked.

Hence, by Definition 3.10.3(a), there exists an extension $r' \in \langle r, Ad_i, t' \rangle$, for some $t' \geq t_r$, such that $\delta(Cov_i(t')) \cap F = \emptyset$. We now show that $r_i = r'$ and $t_i = t'$ satisfy the lemma. By

the induction hypothesis and the construction of extension r', r' is a write-only failure-free sequential extension of r_{i-1} ending at time t' that consists of i complete high-level WRITEs of i distinct values $v_1, \ldots, v_i$ by i distinct clients $c_1, \ldots, c_i$. It remains to show that the implications (a) and (b) hold for $t_i = t'$:

a) $|Cov(t')| \geq if$: By the induction hypothesis $|Cov(t_{i-1})| \geq (i-1)f$, and by Definition 3.10, $Cov(t_{i-1}) \subseteq Cov(t')$. Therefore, we left to show that $|Cov(t') \setminus Cov(t_{i-1})| \geq f$. Since by Corollary 3.15, $|Q_i(t_r)| = f$, and by Lemma 3.11.2, $Q_i(t_r) \subseteq Q_i(t')$, we get $|Q_i(t')| = f$. By Definition 3.8.4, $|Cov_i(t')| \geq |Q_i(t')|$, and by Definition 3.8.3, $|Cov(t') \setminus Cov(t_{i-1})| = |Cov_i(t')|$. Therefore, we get $|Cov(t') \setminus Cov(t_{i-1})| \geq f$.

b) $\delta(Cov(t')) \cap F = \emptyset$: By the induction hypothesis we get that $\delta(Cov(t_{i-1})) \cap F = \emptyset$, and by construction of r' we get that $\delta(Cov_i(t')) \cap F = \emptyset$. By Definition 3.8.3, $Cov(t') = Cov_i(t') \cup Cov(t_{i-1})$. Hence, $\delta(Cov(t')) \cap F = \emptyset$. □

We are now ready to prove the main theorem:

PROOF OF THEOREM 3.1. Let $G \subseteq \mathcal{S}$ be the set consisting of all servers that store at least $\left\lceil \frac{kf}{n-(f+1)} \right\rceil$ base registers (i.e., $\forall s \in G,\ |\delta^{-1}(\{s\})| \geq \left\lceil \frac{kf}{n-(f+1)} \right\rceil$ and $\forall s \in \mathcal{S} \setminus G, |\delta^{-1}(\{s\})| < \left\lceil \frac{kf}{n-(f+1)} \right\rceil$). We first show that $|G| \geq f+1$. Suppose toward a contradiction that $|G| < f+1$, and pick a set F, such that $|F| = f+1$ and $\mathcal{S} \supset F \supset G$. By Lemma 3.7, there exists a run r of A consisting of t steps such that $|Cov(t)| \geq kf$ and $\delta(Cov(t)) \cap F = \emptyset$. Thus, by the pigeonhole argument, and since $|S \setminus F| = n - (f+1)$, there is at least one server in $\mathcal{S} \setminus F$ that stores at least $\left\lceil \frac{kf}{n-(f+1)} \right\rceil$. Therefore, since $F \supset G$ and the number of objects stored on a server is an integer, we get that there is at least one server in $\mathcal{S} \setminus G$ that stores at least $\left\lceil \frac{kf}{n-(f+1)} \right\rceil$ base registers. A contradiction.

We get that $|\delta^{-1}(G)| \geq \left\lceil \frac{kf}{n-(f+1)} \right\rceil (f+1)$. Now, again by Lemma 3.7, there exists a run r' of A consisting of t' steps such that $|Cov(t')| \geq kf$ and $\delta(Cov(t')) \cap G = \emptyset$, meaning that $|\delta^{-1}(\mathcal{S} \setminus G)| \geq kf$. Therefore, we get that $|\delta^{-1}(\mathcal{S})| \geq kf + \left\lceil \frac{kf}{n-(f+1)} \right\rceil (f+1)$. □

4 UPPER BOUND

In this section we present an f-tolerant construction emulating a *wait-free WS-Regular* k-register for all combinations of values of the parameters $k > 0$, $f > 0$, and n where $n > 2f$. Our construction is carefully crafted to deal with the adversarial behaviour (Definition 3.10) that was exploited in the proof of Lemma 3.7 while minimizing the resource complexity. Similarly to multi-writer ABD [23, 31, 37], our algorithm uses *read* and *write quorums* to read from and write to registers. However, since RMW objects are replaced with read/write registers, and covering low-level writes belonging to old WRITEs can overwrite registers at any time, the quorums in our case must have a larger intersection.

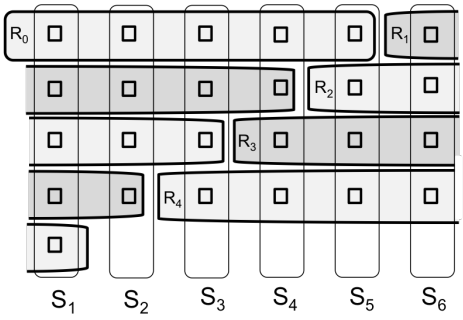

Figure 2: A possible mapping from $\mathcal{R}$ to $\mathcal{S}$ for $n = 6, k = 5$, and $f = 2$.

Let $z \triangleq \left\lfloor \frac{n-(f+1)}{f} \right\rfloor$ and $y \triangleq zf + f + 1$, we construct a collection $\mathcal{R}$ of $m = \left\lfloor \frac{k}{z} \right\rfloor$ disjoint sets $R_0, \ldots, R_{m-1}$, each of which consist of y registers, and if k/z is not an integer, then we add to $\mathcal{R}$ another disjoint set R_m of $(k - \left\lfloor \frac{k}{z} \right\rfloor z)f + f + 1$ registers. Intuitively, z is the maximum number of writers that can be supported by a single set of y registers as can be deduced from Lemma 3.7's argument. If z divides k, then exactly k/z such sets are needed to accommodate the total of k writers. Otherwise, the remaining $k \mod z$ writers are moved to an overflow set R_m. Note that for all $R_i \in \mathcal{R}$, $2f+1 \leq |R_i| \leq n$. Then, we distribute the registers in each set R_i on servers in $\mathcal{S}$ so that every register in R_i is mapped to a different server (i.e., $|\delta(R_i)| = |R_i|$). Figure 2 demonstrates a possible mapping from registers to servers. All in all, we use $\Sigma_{R_i \in \mathcal{R}} |R_i| = \left\lfloor \frac{k}{z} \right\rfloor y + (k - \left\lfloor \frac{k}{z} \right\rfloor z)f + (f+1)(\left\lceil \frac{k}{z} \right\rceil - \left\lfloor \frac{k}{z} \right\rfloor) = \cdots = kf + \left\lfloor \frac{k}{z} \right\rfloor (f+1) = kf + \left\lceil \frac{k}{\left\lfloor \frac{n-(f+1)}{f} \right\rfloor} \right\rceil (f+1)$ registers.

The resulting layout is then used to derive the *read* and *write quorums* as follows: for every set $R_i \in \mathcal{R}$, any subset of R_i of size $|R_i| - f$ is a write quorum for all writers c_j such that $i = \left\lfloor \frac{j}{z} \right\rfloor$; and any subset of registers consisting of *all* registers mapped to $n - f$ servers is a read quorum (i.e., the set of the read quorums is $\{B \subseteq \mathcal{B} : \exists S \in \mathcal{S}$ s.t. $|S| = 2f + 1 \wedge \delta^{-1}(S) = B\}$).

The construction pseudocode appears in Algorithm 1. The registers store values in $\mathbb{V}$ each of which is associated with a unique timestamp. (Note that since safety is required only in write-sequential runs, we do not need to use client identifiers for breaking ties.) To write a value v to the emulated register, a client c_i first accesses a read quorum (via collect() in lines 23–27) and selects a new timestamp ts, which is higher than any other timestamp that has been returned. It then proceeds to trigger low-level writes of $\langle ts, v \rangle$ on all registers in R_j such that $j = \left\lfloor \frac{i}{z} \right\rfloor$, so as to ensure that (1) $\langle ts, v \rangle$ is stored in a write quorum wq (lines 9–12), and (2) no more than f registers in R_j are left covered by writes that have been triggered by c_i in the course of either the current or one of the preceding WRITE invocations. The latter is achieved by preventing c_i from triggering a new low-level write on every register that has not yet responded to the previously triggered one (lines 10–11). To read a value, a client simply

Algorithm 1 f-tolerant k-register emulation from registers for all $f > 0$, $k > 0$, and $n = |\mathcal{S}| > 2f$.

Types:
$TSVal = \mathbb{N} \times \mathbb{V}$, with selectors ts and val.
$States = TSVal \times 2^{TSVal} \times 2^{\mathcal{B}} \times 2^{\mathcal{B}}$
with selectors $tsVal$, $rdSet$, $wrSet$ and $coverSet$.

Base Objects and Servers:
$\forall b \in \delta^{-1}(\mathcal{S})$, $b \in TSVal$, initially, $\langle 0, v_0 \rangle$.
Let $z \triangleq \lfloor \frac{n-(f+1)}{f} \rfloor$, $y \triangleq zf + f + 1$, and $m \triangleq \lceil \frac{k}{z} \rceil$.
$R = \{R_0, \ldots, R_{m-1}\} \subset 2^{\delta^{-1}(\mathcal{S})}$ s.t.
1. $\forall i \in \{0, \ldots, m-2\}$, $|R_i| = y$. If $\lceil \frac{k}{z} \rceil = \lfloor \frac{k}{z} \rfloor$, then $|R_{m-1}| = y$. Else, $|R_{m-1}| = (k - \lfloor \frac{k}{z} \rfloor z)f + f + 1$.
2. $\forall R_i, R_j \in R$, $R_1 \cap R_j = \emptyset$.
3. $\forall R_i \in R$, $|\delta(R_i)| = |R_i|$.

Clients states:
$\forall i \in [k]$, $State_i \in States$, initially,
$\langle \langle \langle 0, 1 \rangle, v_0 \rangle, \emptyset, R_j, \emptyset \rangle$, where $j = \lfloor \frac{i}{z} \rfloor$.

Code for client c_i, $i \in [k]$

```
1:  operation WRITE(v)
2:      value ← collect()
3:      State_i.tsVal.val ← v
4:      State_i.tsVal.ts ← value.ts + 1
5:      j ← ⌊i/z⌋
6:      // do not handle responds between lines 7 to 11
7:      State_i.coverSet ← R_j \ State_i.wrSet
8:      State_i.wrSet ← ∅
9:      || for each b ∈ R_j
10:         if b ∉ State_i.coverSet
11:             trigger b.write(State_i.tsVal)
12:     wait until |State_i.wrSet| ≥ |R_j| − f
13:     return ack

14: operation READ()
15:     value ← collect()
16:     return value.val

17: scan(s)
18:     for each b ∈ δ^{-1}(s) do
19:         trigger b.read()
20:         wait for the matching response

21: collect()
22:     State_i.rdSet ← ∅
23:     || for each s ∈ S do
24:         scan(s)
25:     wait for n − f scans to complete
26:     ts ← max({ts' | ⟨ts', ∗⟩ ∈ State_i.rdSet})
27:     return ⟨v, ts'⟩ ∈ State_i.rdSet : ts' = ts

28: upon receiving b.read() respond res do
29:     State_i.rdSet ← State_i.rdSet ∪ {res}

30: upon receiving b.write(∗) respond do
31:     if b ∈ State_i.coverSet then
32:         State_i.coverSet ← State_i.coverSet \ {b}
33:         trigger b.write(State_i.tsVal)
34:     else
35:         State_i.wrSet ← State_i.wrSet ∪ {b}
```

reads all registers in a read quorum, via collect(), and returns the value having the highest timestamp.

Observe that by construction of $\mathcal{R}$, for every set $R_i \in \mathcal{R}$, (1) the number of clients mapped to write quorums in R_i is $\lfloor \frac{|R_i|-(f+1)}{f} \rfloor = \frac{|R_i|-(f+1)}{f}$, and (2) any write quorum in R_i intersects with any read quorum on at least $|R_i| - f$ registers. This, along with the algorithm's guarantee that no more than f low-level writes remain pending upon completion of every high-level WRITE, ensures that in a write-sequential run, the latest value written by a high-level WRITE is always available to the subsequent READs. In addition, since the registers in every (read or write) quorum are mapped to exactly $n - f$ servers, each quorum access is guaranteed to terminate, and thus, the algorithm is wait-free. Thus, we have the following (see [14] for a full proof):

THEOREM 4.1. *For all $k > 0$, $f > 0$, and $n > 2f$, there exists an f-tolerant algorithm emulating a wait-free WS-Regular k-register using a collection of n servers storing $kf + \lceil \frac{k}{z} \rceil (f + 1)$ wait-free z-writer/multi-reader atomic base registers where $z = \lfloor \frac{n-(f+1)}{f} \rfloor$.*

5 DISCUSSION AND FUTURE WORK

We studied space complexity of emulating an f-tolerant register from fault-prone base objects as a function of the base object type, the number of writers k, the number of available servers n, and the failure threshold f. For the three object types considered, we established a sharp separation (by factor k) between registers and both max-registers and CAS in terms of the number of objects of the respective types required to support the emulation; we showed that no such separation between max-registers and CAS exists. Interestingly, these results shed light on the resource complexity bounds in the standard shared memory model (i.e., without object failures) as evidenced by our proof of a lower bound on the number of registers required for implementing a k-writer max-register. Our main technical contribution comprises the lower bound of $\left\lceil \frac{k}{\frac{n-(f+1)}{f}} \right\rceil (f + 1) + kf$ and the upper bound of $\left\lceil \frac{k}{\lfloor \frac{n-(f+1)}{f} \rfloor} \right\rceil (f + 1) + kf$ on the resource complexity of emulating an f-tolerant k-writer register from n fault-prone servers storing read/write registers. To strengthen our lower bound, it was proved for emulations satisfying weak liveness and safety properties.

Future directions. First, for some choices of k and n, our bounds leave a small gap that can be closed. Second, an interesting question that arises is whether our lower bound is tight for stronger properties. In the special case of $n = 2f + 1$ servers, emulation with stronger regularity [36] is possible with $(2f + 1)k$ registers (tight to our lower bound). However, the question of the general case ($n \geq 2f+1$) remains open. In addition, since atomicity usually requires readers to write, it is interesting to investigate whether the space complexity (assuming read/write registers) in this case also linearly depends on the number of readers.

Our results suggest a new classification of the data types based on space complexity of fault-tolerant emulations built from base objects of these types, which is fundamentally different from those established by [24] and [18]. A promising future direction is to extend this classification with additional types (e.g., multiple assignment), and potentially generalize it into a full-fledged hierarchy of its own.

Another possible direction is to consider the time complexity of the emulations. For example, we showed that although

a max-register can be implemented from a single CAS, the time complexity of the implementation is high. An interesting open question is to determine whether this tradeoff is inherent.

ACKNOWLEDGMENTS

We are thankful to Idit Keidar for reading many drafts and helpful discussions, Dan Dobre and Alex Shraer for their contribution to earlier versions of this paper, and PODC '17 anonymous reviewers for valuable feedback.

REFERENCES

[1] Yehuda Afek, Hagit Attiya, Arie Fouren, Gideon Stupp, and Dan Touitou. 1999. Long-lived renaming made adaptive. In *PODC '99*.
[2] Marcos K. Aguilera, Burkhard Englert, and Eli Gafni. 2003. On Using Network Attached Disks As Shared Memory. In *PODC '03*.
[3] Marcos K. Aguilera, Idit Keidar, Dahlia Malkhi, Jean-Philippe Martin, and Alexander Shraer. 2010. Reconfiguring Replicated Atomic Storage: A Tutorial. *Bulletin of the EATCS* 102 (2010).
[4] James Aspnes, Hagit Attiya, and Keren Censor. 2009. Max-registers, counters, and monotone circuits. In *PODC '09*.
[5] Hagit Attiya, Amotz Bar-Noy, and Danny Dolev. 1995. Sharing memory robustly in message-passing systems. *J. ACM* 42, 1 (1995).
[6] Hagit Attiya and Faith Ellen. 2014. Impossibility Results for Distributed Computing. *Synthesis Lectures on Distributed Computing Theory* 5, 1 (2014).
[7] Hagit Attiya and Arie Fouren. 2003. Algorithms adapting to point contention. *J. ACM* 50, 4 (July 2003).
[8] Hagit Attiya and Jennifer Welch. 2004. *Distributed Computing: Fundamentals, Simulations, and Advanced Topics* (2nd ed.). Wiley, Chapter 10.4, 234.
[9] Cristina Basescu, Christian Cachin, Ittay Eyal, Robert Haas, Alessandro Sorniotti, Marko Vukolic, and Ido Zachevsky. 2012. Robust data sharing with key-value stores. In *DSN '12*. 1–12.
[10] James E. Burns and Nancy A. Lynch. 1993. Bounds on Shared Memory for Mutual Exclusion. *Inf. Comput.* 107, 2 (1993).
[11] Viveck R. Cadambe, Nancy A. Lynch, Muriel Médard, and Peter M. Musial. 2017. A coded shared atomic memory algorithm for message passing architectures. *Distributed Computing* 30, 1 (2017), 49–73.
[12] Viveck R. Cadambe, Zhiying Wang, and Nancy A. Lynch. 2016. Information-Theoretic Lower Bounds on the Storage Cost of Shared Memory Emulation. In *PODC '16*.
[13] Gregory Chockler, Rachid Guerraoui, and Idit Keidar. Amnesic Distributed Storage. In *DISC '07*.
[14] Gregory Chockler and Alexander Spiegelman. 2017. Space Complexity of Fault-Tolerant Register Emulations. arXiv:1705.07212
[15] Brian F. Cooper et al. 2008. PNUTS: Yahoo!'s hosted data serving platform. *Proc. VLDB Endowment* 1, 2 (2008).
[16] Partha Dutta, Rachid Guerraoui, Ron R. Levy, and Marko Vukolic. 2010. Fast Access to Distributed Atomic Memory. *SIAM J. Comput.* 39, 8 (2010), 3752–3783.
[17] Amazon DynamoDB. http://aws.amazon.com/dynamodb/.
[18] Faith Ellen, Rati Gelashvili, Nir Shavit, and Leqi Zhu. 2016. A Complexity-Based Hierarchy for Multiprocessor Synchronization. In *PODC '16*.
[19] Burkhard Englert and Alexander A. Shvartsman. 2000. Graceful Quorum Reconfiguration in a Robust Emulation of Shared Memory. In *ICDCS '2000*.
[20] Rui Fan and Nancy Lynch. 2003. Efficient Replication of Large Data Objects. In *DISC '03*.
[21] Rati Gelashvili. 2015. On the Optimal Space Complexity of Consensus for Anonymous Processes. In *DISC '15*.
[22] Chryssis Georgiou, Nicolas C. Nicolaou, and Alexander A. Shvartsman. 2009. Fault-tolerant semifast implementations of atomic read/write registers. *Journal of Parallel Distributed Computing* 69, 1 (2009), 62–79.
[23] Seth Gilbert, Nancy A. Lynch, and Alexander A. Shvartsman. 2010. Rambo: a robust, reconfigurable atomic memory service for dynamic networks. *Distributed Computing* 23, 4 (2010), 225–272.
[24] Maurice Herlihy. 1991. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 13, 1 (1991), 124–149.
[25] Maurice Herlihy and Jeannette M. Wing. 1990. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Trans. Program. Lang. Syst.* 12, 3 (1990).
[26] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. 2010. ZooKeeper: wait-free coordination for internet-scale systems. In *USENIX ATC '10*.
[27] Prasad Jayanti, Tushar Chandra, and Sam Toueg. 1998. Fault-tolerant wait-free shared objects. *J. ACM* 45, 3 (1998), 451–500.
[28] Prasad Jayanti, King Tan, and Sam Toueg. 2000. Time and Space Lower Bounds for Nonblocking Implementations. *SIAM J. Comput.* 30, 2 (April 2000), 438–456. https://doi.org/10.1137/S0097539797317299
[29] Leslie Lamport. 1986. On interprocess communication: Parts I and II. *Distributed computing* 1, 2 (1986).
[30] Nancy Lynch. 1996. *Distributed Algorithms*. Morgan Kaufman.
[31] Dahlia Malkhi and Michael K. Reiter. 1998. Byzantine Quorum Systems. *Distributed Computing* 11, 4 (1998), 203–213.
[32] MongoDB. http://www.mongodb.org/.
[33] Jun Rao, Eugene J. Shekita, and Sandeep Tata. 2011. Using Paxos to Build a Scalable, Consistent, and Highly Available Datastore. *PVLDB* (2011).
[34] Riak. http://basho.com/riak.
[35] Amazon Simple Storage Service (Amazon S3). http://aws.amazon.com/s3/.
[36] Cheng Shao, Jennifer L Welch, Evelyn Pierce, and Hyunyoung Lee. 2011. Multiwriter consistency conditions for shared memory registers. *SIAM J. Comput.* 40, 1 (2011).
[37] C. Shao, J. L. Welch, E. Pierce, and H. Lee. 2011. Multiwriter Consistency Conditions for Shared Memory Registers. *SIAM J. Comput.* 40, 1 (2011).
[38] Amazon SimpleDB. http://aws.amazon.com/simpledb/.
[39] Alexander Spiegelman, Yuval Cassuto, Gregory Chockler, and Idit Keidar. 2016. Space Bounds for Reliable Storage: Fundamental Limits of Coding. In *PODC '16*.
[40] Microsoft Azure Storage. http://www.windowsazure.com/en-us/manage/services/storage.
[41] Leqi Zhu. 2016. A Tight Space Bound for Consensus. In *STOC '16*.

A MAX-REGISTER FROM CAS

We present here a wait-free emulation of an atomic max-register on top of a single CAS object. The pseudocode appears in Algorithm 2, and the correctness proof in the full paper [14].

Algorithm 2 max-register from a single CAS object

Types:
$\mathbb{V}$: set of ordered values.
Base Objects:
$b \in \mathbb{V}$, CAS object, initially v_0.
Local variables:
$tmp \in \mathbb{V}$, initially v_0.

1: **operation** *write-max*(v)
2: **while** true **do**
3: $tmp \leftarrow b.C\&S(v_0, v_0)$
4: **if** $tmp \geq v$ **then**
5: return ok
6: $b.C\&S(tmp, v)$
7: **operation** *read-max*()
8: $tmp \leftarrow b.C\&S(v_0, v_0)$
9: return tmp