Robust Data Sharing with Key-Value Stores

Cristina Băsescu[†] Christian Cachin[*] Ittay Eyal[§] Robert Haas[*] Marko Vukolić[‡]

Abstract

A key-value store (KVS) offers functions for storing and retrieving values associated with unique keys. KVSs have become the most prominent way to access Internet-scale "cloud" storage systems.

We present a fault-tolerant wait-free efficient algorithm that emulates a multi-reader multi-writer register from a set of KVS replicas in an asynchronous environment. Our implementation serves an unbounded number of clients that use the storage. It tolerates crashes of a minority of the KVSs and crashes of any number of clients. We provide two variants of our algorithm: one implementing an atomic register and one implementing a regular register; the latter does not require read operations to store data at the underlying KVSs.

We note that applying textbook reliable storage solutions to this scenario is either impossible or prohibitively inefficient. Our algorithms maintain *two* copies of the stored value per KVS in the common case (in the absence of faults and concurrency) and we show this is indeed necessary. In runs with concurrency, we prove that the algorithms' maximum space complexity is linear in the number of concurrent operations.

1 Introduction

1.1 Background

In the recent years, the *key-value store* (*KVS*) abstraction has become the most prominent way to access Internet-scale "cloud" storage systems. Such systems provide storage and coordination services for online platforms [14, 32, 7, 26, 39], ranging from web search to social networks, but they are also available to consumers directly [6, 40, 34, 17, 33].

A KVS offers a range of simple functions for manipulation of unstructured data objects, called *values*, each one identified by a unique *key*. While different services and systems offer various extensions to the KVS interface, the common denominator of existing KVS services implements an associative array: A client may *store* a value by associating the value with a key, *retrieve* a value associated with a key, *list* the keys that are currently associated, and *remove* a value associated with a key.

Existing KVS services provide high availability and reliability using replication internally, such that the system tolerates the failure of individual machines and disks. However, as the KVS service is managed by one provider, there are many further common components (and thus failure modes) that may affect the KVS. A failure of such a component may lead to service outage or even to data being lost, as witnessed during an Amazon S3 incident [4], Google's temporary loss of email data [22], and Amazon's recent service disruption [5]. As a remedy, a client may increase data reliability by replicating it among several storage providers

[†]Vrije Universiteit Amsterdam. cbu200@few.vu.nl.
[*]IBM Research, Zurich Research Laboratory, CH-8803 Rüschlikon, Switzerland. {cca, rha}@zurich.ibm.com.
[§]Department of Electrical Engineering, The Technion – Israel Institute of Technology. ittay@tx.technion.ac.il.
[‡]Eurécom, 2229, Route des Crêtes, BP 193, F-06904 Sophia Antipolis cedex, France. vukolic@eurecom.fr.

(all offering a KVS interface), using the guarantees offered by *robust* distributed storage algorithms [21, 8]. Such an algorithm relies on multiple storage providers, called *base objects* here, and emulates a single, more reliable shared storage abstraction, which we model as a *read/write register*. The register tolerates asynchrony, concurrency, and faults among the clients and the base objects.

Many well-known robust distributed storage algorithms exist (for an overview see [11]). Perhaps surprisingly, none of them directly exploits key-value stores as base objects. The problem arises because existing solutions are either (1) unsuitable for KVSs since they rely on storage nodes that perform custom *computation*, which a KVS cannot[1] do, or (2) prohibitively expensive, in the sense that they require as many base objects as there are clients [19, 1]. We return to this issue in Section 2. In the following, we describe the challenges behind running robust storage algorithms over a set of KVS base objects.

This work is motivated by our project to build a dependable *Intercloud storage* solution from multiple unreliable KVSs [12]. Our algorithms emulate a register because this represents a basic form of storage, from which a KVS service or more elaborate abstractions may be implemented.

1.2 Challenges

Many previous robust register emulations are based on versioning [37], i.e., they associate each stored value with a version (also called a timestamp [27]). Consider the multi-writer version of the seminal register implementation by Attiya, Bar-Noy and Dolev (ABD) [8]. A writer checks which version is the largest in some majority of the base objects, and then stores a value with a larger version in a majority. The base object then *performs computation* and stores the new version only if it is larger than the locally stored version. However, the KVS interface does not include such an operation. Therefore, if an algorithm like ABD is cast as-is into the KVS context, all values are put with the same key, and large versions might be overwritten by small ones. We refer to this as the *old-new overwrite* problem.

A naïve KVS-enabled solution is to store each version in its own key; however, this would result in infinite space complexity, as all versions that were ever stored are accumulated. Removing small versions from a KVS once larger ones are put, could in turn jeopardize wait-freedom: a read operation has to list the existing keys, and then get the latest version. If this version is removed between the time it appears in the result of a list operation and the time the operation tries to get it, the read operation fails. We refer to this as the *garbage collection (GC) racing* problem.

1.3 Contribution

In this paper, we formally define a key-value store (KVS) object. Our KVS interface is very basic because it represents the common denominator of existing commercial KVSs. This is crucial for replicating data across independent KVS providers.

We provide two robust, asynchronous, and efficient emulations of a register over a set of KVSs, which may fail by crashing. Inspired by Internet-scale systems, both emulations are designed for an unbounded number of clients and are multi-reader multi-writer (MWMR). Both emulations are *wait-free* [24], i.e., all operations invoked by a correct client eventually complete. They are also *optimally resilient*, i.e., tolerate the failure of any minority of the KVSs and of any number of clients.

The two emulations differ in their consistency semantics. The first emulates a regular register [36] (where, roughly speaking, a read operation may return a value written by the latest write that precedes it or one of the concurrently written values [28, 36]). This algorithm does not require read operations to write

[1] A recent *active* KVS research prototype explores this option [35].

to the KVSs and to change their state. Precluding readers from writing is practically appealing, since the clients may belong to different domains and not all readers may have write access to the shared memory. But it also poses a challenge because of the GC racing problem. Our solution is to have the write algorithm store the same value *twice* in every KVS: (1) under an *eternal* key, which is never removed by a garbage collector, and therefore is vulnerable to the old-new overwrite and (2) under a *temporary* key, named according to the version; old temporary keys are garbage-collected by write operations, making them vulnerable to the GC racing problem. In a sense, the eternal and temporary copies complement each other and, together, guarantee the desirable properties of our emulation outlined above.

The second algorithm emulates an *atomic* [28] register, i.e., each read and write operation can be placed at a single point in time between its invocation and response. This emulation requires read operations to write, but this is necessary [28, 18, 9]. We derive our atomic emulation from the regular one, by employing the method that readers write back the returned value [8].

Both algorithms maintain only two copies of the stored value per KVS in the common case (in the absence of faults and concurrency). We show that this is necessary. In the worst case, a stored value exists in every KVS once for every concurrent write operation, in addition to the one stored under the eternal key.

Note that some of the available KVSs export proprietary versioning information [6, 39]. However, one cannot exploit this for a data replication algorithm before the format and semantics of those versions has been harmonized.

Roadmap. The rest of the paper is organized as follows. We discuss related work in Section 2. We describe the system model in Section 3; this includes a formal specification of the key-value store interface. Then, in Section 4, we provide two robust algorithms that use KVS objects to emulate a read/write register. In Section 5 we prove the correctness of the algorithms and in Section 6 we discuss their efficiency. Section 7 concludes the paper.

2 Related Work

There is a rich body of literature on robust register emulations that provide guarantees similar to ours. However, virtually all of them assume read-modify-write functionalities, or computation at the base objects. These include the single-writer multi-reader (SWMR) atomic wait-free register implementation of Attiya et al. [8], its dynamic multi-writer counterparts by Lynch and Shvartsman [30, 31] and Englert and Shvartsman [16], wait-free simulations of Jayanti et al. [25], low-latency atomic wait-free implementations of Dutta et al. [15] and Georgiou et al. [20], and the consensus-free versions of Aguilera et al. [3]. These solutions are not directly applicable to our model where KVSs are used as base objects, due to the old-new overwrite problem, described in Section 1.2.

Notable exceptions that are applicable in our KVS context are SWMR regular register emulation by Gafni and Lamport [19] and its Byzantine variant by Abraham et al. [1] that use registers as base objects. However, transforming these SWMR emulations to support a large number of writers is inefficient: standard register transformations [9, 11] that can be used to this end require at least as many SWMR regular registers as there are clients, even if there are no faults. This is prohibitively expensive in terms of space complexity and effectively limits the number of supported clients. Chockler and Malkhi [13] acknowledge this issue and propose an algorithm that supports an unbounded number of clients (like our algorithm). However, their method uses base objects (called "active disks") that may carry out computations. In contrast, our emulation leverages the operations in the KVS interface, which is more general than a register due to its list and remove operations, and supports an unbounded number of clients. Ye et al. [41] overcome the GC racing problem by having the readers "reserve" the versions they intend to read, by storing extra values that signal to the garbage collector not to remove the version being read. This approach requires readers to have write access, which is not desirable.

Two recent works share our goal of providing robust storage from KVS base objects. Abu-Libdeh et al. [2] propose RACS, an approach that casts RAID techniques to the KVS context. RACS uses a model different from ours and relies on a proxy between the clients and the KVSs, which may become a single point-of-failure. Bessani et al. [10] propose a distributed storage system, called DepSky, which is closest to this work in spirit. It employs erasure coding and cryptographic tools to achieve robustness using multiple KVS objects, which are prone to even Byzantine faults. However, the basic version of DepSky allows only a single writer and thereby circumvents the old-new overwrite problem and the GC racing problem. An extension supports multiple writers through a locking mechanism that determines a unique writer using client-to-client communication. However, this extension may block a writing client, whereas our solution is wait-free. Furthermore, client-to-client communication is not possible in our model.

3 Model

After introducing the formal notation (Section 3.1), we provide a definition of the key-value store object (Section 3.2) and describe the system model (Section 3.3).

3.1 Preliminaries

We introduce here some basic concepts of distributed algorithms. First we describe a system execution, the specifications of objects, and the relations between them. Then we define register objects with regular and atomic semantics.

3.1.1 Executions

The system is comprised of *clients* and *(base) objects*. We model both as I/O automata [29], which contain state and potential transitions that are triggered by *actions*. The interface of an I/O automaton is determined by external (input and output) actions. A client may *invoke* an *operation*[2] on an object (with an output action of the client automaton that is also an input action of the object automaton). The object reacts to this invocation, possibly involving state transitions and internal actions, and returns a *response* (an output action of the object that is also an input action of the client). We consider an asynchronous system, i.e., there are no timing assumptions that relate invocations and responses. (Consult [29, 9] for details.)

Clients and objects may *fail* by stopping, i.e., *crashing*, which we model by a special action *stop*. When *stop* occurs at automaton A, all actions of A become disabled indefinitely and A no longer modifies its state. A client or base object that does not fail is called *correct*.

An *execution* σ of the system is a sequence of invocations and responses. We define a partial order among the operations. An operation o_1 *precedes* another operation o_2 (and o_2 *follows* o_1) if the response of o_1 precedes the invocation of o_2 in σ. We denote this by $o_1 \prec_\sigma o_2$. The two operations are *concurrent* if neither of them preceded the other. An operation o is *pending* in an execution σ if σ contains the invocation of o but not its response; otherwise the operation is *complete*. An execution σ is *well-formed* if every subsequence thereof that contains only the invocations and responses of one client on one object consists of alternating invocations and responses, starting with an invocation. A well-formed execution σ is *sequential*

[2]For simplicity, we refer to an *operation* when we should be referring to *operation execution*.

if every prefix of σ contains at most one pending operation; in other words, in a sequential execution, the response of every operation immediately follows its invocation.

A *real-time sequential permutation* π of an execution σ is a sequential execution that contains all operations that are invoked in σ and only those operations and in which for any two operations o_1 and o_2 such that $o_1 \prec_\sigma o_2$, it holds $o_1 \prec_\pi o_2$. Since a sequential execution is a sequence of pairs, each containing the invocation and the response of one operation, we slightly abuse the terminology and refer to π as the sequence of these operations.

A *sequential specification* of some object O is a prefix-closed set of sequential executions containing operations on O. It defines the desired behavior of O. A sequential execution π is *legal* with respect to the sequential definition of O if the subsequence of σ containing only operations on O lies in the sequential specification of O.

Finally, an object implementation is *wait-free* if it eventually responds to an invocation by a correct client [23].

3.1.2 Register Specifications

Sequential Register. A register [28] is an object that supports two operations: one for writing a value $v \in \mathcal{V}$, denoted by **write**(v), which returns ACK, and one for reading a value, denoted by **read**$()$, which returns a value in $\mathcal{V}$. The sequential specification of a register requires that every **read** operation returns the value written by the last preceding **write** operation in the execution, or the special value $\perp$ if no such operation exists. For simplicity, our description assumes that every distinct value is written only once.

Registers may exhibit different semantics under concurrent access, as described next.

Multi-Reader Multi-Writer Regular Register. The following semantics describe a *multi-reader multi-writer regular register* (*MRMW-regular*), adapted from [36]. A MRMW-regular register only guarantees that different **read** operations agree on the order of preceding **write** operations.

Definition 1 (MRMW-regular register). *A well-formed execution σ of a register is MRMW-regular if there exists a sequential permutation π of the operations in σ as follows: for each* ***read*** *operation r in σ, let π_r by a subsequence of π containing r and those* ***write*** *operations that do not follow r in σ; furthermore, let σ_r be the subsequence of σ containing r and those* ***write*** *operations that do not follow it in σ; then π_r is a legal real-time sequential permutation of σ_r. A register is MRMW-regular if all well-formed executions on that register are MRMW-regular.*

Atomic Register. A stronger consistency notion for a concurrent register object than regular semantics is atomicity [28], also called linearizability [24]. In short, atomicity stipulates that it should be possible to place each operation at a singular point (linearization point) between its invocation and response.

Definition 2 (Atomicity). *A well-formed execution σ of a concurrent object is* atomic *(or* linearizable*), if σ can be extended (by appending zero or more responses) to some execution σ', such that there is a legal real-time sequential permutation π of σ'. An object is atomic if all well-formed executions on that object are atomic.*

3.2 Key-Value Store

A *key-value store* (*KVS*) object is an associative array that allows storage and retrieval of *values* in a set $\mathcal{X}$ associated with *keys* in a set $\mathcal{K}$. The size of the stored values is typically much larger than the length of a key, so the values in $\mathcal{X}$ cannot be translated to elements of $\mathcal{K}$ and be stored as keys.

A KVS supports four operations: (1) *Storing* a value x associated with a key *key* (denoted **put**(key, x)), (2) *retrieving* a value x associated with a key ($x \leftarrow$ **get**(key)), which may also return FAIL if *key* does not exist, (3) *listing* the keys that are currently associated ($list \leftarrow$ **list**$()$), and (4) *removing* a value associated with a key (**remove**(key)).

Our formal sequential specification of the KVS object is given in Algorithm 1. This implementation maintains in a variable *live* the set of associated keys and values. The *space complexity* of a KVS at some time during an execution is given by the number of associated keys, that is, by the value $|live|$.

Algorithm 1: Key-value store object i

```
state
    $live \subseteq \mathcal{K} \times \mathcal{X}$, initially $\emptyset$
On invocation put$_i(key, value)$
    $live \leftarrow (live \setminus \{\langle key, x\rangle \mid x \in \mathcal{X}\}) \cup \langle key, value\rangle$
    return ACK
On invocation get$_i(key)$
    if $\exists x : \langle key, x\rangle \in live$ then
        return $x$
    else
        return FAIL
On invocation remove$_i(key)$
    $live \leftarrow live \setminus \{\langle key, x\rangle \mid x \in \mathcal{X}\}$
    return ACK
On invocation list$_i()$
    return $\{key \mid \exists x : \langle key, x\rangle \in live\}$
```

3.3 Register Emulation

The system is comprised of a finite set of clients and a set of n atomic wait-free KVSs as base objects. Each client is named with a unique identifier from an infinite ordered set $\mathcal{ID}$. The KVS objects are numbered $1, \ldots, n$. Initially, the clients do not know the identities of other clients or the total number of clients.

Our goal is to have the clients *emulate* a MRMW-regular register and an atomic register using the KVS base objects [29]. The emulations should be wait-free and tolerate that any number of clients and any minority of the KVSs may crash. Furthermore, an emulation algorithm should associate only few keys to values in every KVS (i.e., have low space complexity).

4 Algorithm

We present an algorithm for implementing a MRMW-regular register, where **read** operations do not store data at the KVSs, and an extension of this algorithm that provides an atomic register, but where **read** operations store data at the KVSs.

Inspired by previous work on fault-tolerant register emulations, our algorithm makes use of versioning. Clients associate versions with the values they store in the KVSs. In each KVS there may be several values stored at any time, with different versions. Roughly speaking, when writing a value, a client associates it with a version that is larger than the existing versions, and when reading a value, a client tries to retrieve the one associated with the largest version [8]. Since a KVS cannot perform computations and atomically store one version and remove another one, values associated with obsolete versions may be left around. Therefore our algorithm explicitly removes unused values, in order to reduce the space occupied at a KVS.

A version is a pair[3] $\langle seq, id \rangle \in \mathbb{N}_0 \times \mathcal{ID}$, where the first number is a sequence number and the second is the identity of the client that created the version and used it to store a value. When comparing versions with the $<$ operator and using the $\max$ function, we respect the lexicographic order on pairs. We assume that the key space of a KVS is the version space, i.e., $\mathcal{K} = \mathbb{N}_0 \times \mathcal{ID}$, and that the value space of a KVS allows clients to store either a register value from $\mathcal{V}$ or a version and a value in $(\mathbb{N}_0 \times \mathcal{ID}) \times \mathcal{V}$.[4]

At the heart of our algorithm lies the idea of using *temporary* keys that are created and later removed at the KVSs, and an *eternal* key ETERNAL that is never removed. Both store a register value and its version. The key of a temporary keys is the version, and it is associated with the register value. The eternal key is associated to a pair comprised of the version and the register value.

Hence, when a client writes to the emulated register, it determines a new version, accesses a majority of the KVSs, and stores the value and version twice at every KVS — once under a new temporary key, named according to the version, and once under the eternal key, overwriting its current value. Our algorithm lets every writer perform garbage collection (GC) to remove obsolete temporary versions. By performing GC before writing to a KVS, the algorithm is able to bound space complexity.

When a client invokes a read from the emulated register, it obtains a version and a value from a majority of the KVSs and returns the value associated with the largest obtained version. To obtain such a pair from a KVS, the reader first determines the currently largest stored version, denoted by ver_0, through a snapshot of temporary keys with a **list** operation. Then the reader enters a loop, from which it only exits after finding a value associated with a version that is at least ver_0. It first retrieves the value under the key representing the largest version. This may fail because a concurrent writer has meanwhile removed this key (GC racing). In this case, the reader retrieves the version/value pair associated with the eternal key and checks its version. If this version is greater than or equal to ver_0, the reader returns this value. However, if this version is smaller than ver_0, an old-new overwrite occurred. In the latter case, the reader again lists the keys in the KVS to obtain a new snapshot of temporary keys and repeats the loop. The loop eventually terminates, since either the number of writes is finite and retrieving a value from a temporary key succeeds, or, in case the number of writes is not finite, the old-new overwrite problem may appear only finitely many times.

We formally describe the operation of the MRMW-regular variant of the algorithm in Section 4.2. Later we extend our algorithm in Section 4.3 to provide atomic registers, by having the **read** operation write back the read value before returning it.

4.1 Notation for Algorithms

We formulate the algorithms using the concept of concurrent execution threads. Each function initiates one thread per base object, which manages the communication with this base object by invoking operations and handling the responses. The threads concurrently perform the same procedure on each of the base objects; the procedure consists of a series of internal client steps and operations on the base object. We use the **concurrently** keyword to express this execution flow. When the procedure completes on a majority of base objects, the client function progresses past the **concurrently** block. The threads whose base-object

[3]We denote by $\mathbb{N}_0$ the set $\{0, 1, 2, \dots\}$.

[4]In other words, $\mathcal{X} = \mathcal{V} \cup (\mathbb{N}_0 \times \mathcal{ID}) \times \mathcal{V}$. Alternatively one may assume that there exists a one-to-one transformation from the version space to the KVS key space, and from the set of values written by the clients to the KVS value space. In practical systems, where $\mathcal{K}$ and $\mathcal{X}$ are strings, this assumptions holds.

operations have not completed are discarded, but the system keeps track of pending operations at the base objects.

Although this description of the algorithm is convenient, it hides some concurrency issues of the shared-memory model that occur at the client, namely the coordination between the threads. A more detailed description would replace a **concurrently** block with a single loop that simulates the procedure in the block for each of the base objects. State variables would be maintained for each base object, and in each iteration, the function would handle one response, possibly changing the state variables of the relevant base object. The result would be collected and would serve to determine the exit condition of the loop. The state of pending operations would be maintained between **concurrently** blocks to assert well-formedness.

4.2 MRMW-Regular Register

We describe the algorithm for emulating a regular register. The pseudo code of its **regularRead** and **regularWrite** functions appears in Algorithms 3 and 5, respectively.

4.2.1 Read

The client's **read** implementation uses the function **getFromKVS**(i), shown in Algorithm 2. Roughly speaking, it retrieves from KVS i the value that was written most recently before it started, or a value that is written concurrently. The function first lists the keys in the KVS. If there are no temporary keys, then no **write** operation has completed, and the function returns a default version and value $\perp$ (lines 3–4). Otherwise, if there are temporary keys in the KVS, the client remembers the largest version corresponding to those keys, denoted ver_0 (line 5). The algorithm ensures that it returns a value associated with a version that is not smaller than ver_0.

The client then repeatedly tries to retrieve a value associated with a temporary key representing a large enough version, or to retrieve a suitable value stored under the eternal key from the KVS. More precisely, it determines the largest version in the list of temporary keys and tries to retrieve the value associated with the largest one from the KVS (lines 7–9). This step may fail if the particular version is removed by some writer between the times when the client issues the **list** and the **get** operations. In this case, the client retrieves the data stored under the eternal key, which consists of a value and a version. However, the algorithm returns this version-value pair to the client only if its version is not smaller than v_0 (lines 10–13). Note that the **get**(ETERNAL) call never returns FAIL, since the first **list** returned a non-empty list. As the eternal key is always stored first by a writer and never removed, it must exist. If the version retrieved from the eternal key is too small, then the algorithm lists again the keys currently in the KVS and repeats the steps above. As shown in the argument for liveness (Theorem 6), this loop exits after finitely many iterations.

The **regularRead** algorithm (Algorithm 3) implements the read operation from the register. It uses the **getFromKVS** function to obtain a version-value pair from a majority of the KVSs (lines 3–6). For each KVS, the obtained version has the property that it was the largest version associated with a value at this KVS at some time after the invocation of **read**. The algorithm then returns the value from the pair that contains the largest version among all pairs it obtained (line 7).

4.2.2 Write

The client's **regularWrite** algorithm (Algorithm 5) writes a value to the register. The operation is divided into two parts.

Algorithm 2: Retrieve a legal version-value pair from a KVS

```
function getFromKVS($i$)
    $list \leftarrow \textbf{list}_i() \setminus$ ETERNAL
    if $list = \emptyset$ then
        return $\langle\langle 0, \bot\rangle, \bot\rangle$
    $ver_0 \leftarrow \max(list)$
    while True do
        $val \leftarrow \textbf{get}_i(\max(list))$
        if $val \neq$ FAIL then
            return $\langle \max(list), val\rangle$
        $\langle ver, val\rangle \leftarrow \textbf{get}_i(\text{ETERNAL})$
        if $ver \geq ver_0$ then
            return $\langle ver, val\rangle$
        $list \leftarrow \textbf{list}_i() \setminus$ ETERNAL
```

Algorithm 3: Client c **read** operation of the MRMW-regular register

```
function regularRead$_c$()
    $results \leftarrow \emptyset$
    concurrently for each $1 \leq i \leq n$, until a majority completes
        if KVS $i$ is pending then wait until it responds
        $result \leftarrow$ getFromKVS($i$)
        $results \leftarrow results \cup \{result\}$
    return $val$ such that $\langle ver, val\rangle \in results$ and $ver' \leq ver$ for any $\langle ver', val'\rangle \in results$
```

In the first part, the client lists the temporary keys in each base object and determines the largest version for each of them (Algorithm 5, lines 3–6). After receiving answers from a majority, it takes the largest version among all the results, increments it, and associates it with the value for writing (lines 7–8). This part is the same as in many existing shared-memory algorithms [8].

The second part of Algorithm 5 is new and closely tied to our algorithm for reading. The client stores the value together with the associated new version by calling the function **putInKVS**, shown in Algorithm 4, for all KVSs. The function is also responsible for garbage collection. It obtains a list of all existing keys first (lines 2–3) and removes all obsolete temporary keys (excluding the key corresponding to the maximal version, lines 4–5). The function subsequently stores the value to be written under the eternal key ETERNAL together with the new version (line 6). Afterwards, it checks whether the new version is larger than the maximal version of an existing key. If yes, it also stores the new value under the temporary key corresponding to the new version and removes the key holding the previous maximal version. Once the function **putInKVS** finishes for a majority of the KVSs, the algorithm for writing to the register completes. It is important for ensuring termination of concurrent **read** operations that the writer first stores the value under the eternal key and later under the temporary key.

4.3 Atomic Register

Atomicity is achieved by having a client write back its read value before returning it. This is similar to the write-back procedure of ABD [8]. We implement an atomic register by replacing the **regularRead** algorithm with **atomicRead**, shown in Algorithm 6. The first phase of the function (lines 2–7) is similar to the **regularRead** function, which takes the value associated with the maximal version found among a majority of the KVSs. The second phase (lines 8–10) is similar to the second phase of the **regularWrite**

Algorithm 4: Store a value and a given version in a KVS

```
function putInKVS$(i, ver_w, val_w)$
    $list \leftarrow$ list$_i()$
    $obsolete \leftarrow \{v \mid v \in list \wedge v \neq \text{ETERNAL} \wedge v < \max(list)\}$
    foreach $ver \in obsolete$ do
        remove$_i(ver)$
    put$_i(\text{ETERNAL}, \langle ver_w, val_w\rangle)$
    if $ver_w > \max(list)$ then
        put$_i(ver_w, val_w)$
        remove$_i(\max(list))$
```

Algorithm 5: Client c **write** operation of the MRMW-regular register

```
function regularWrite$_c(val_w)$
    $results \leftarrow \{\langle 0, \perp\rangle\}$
    concurrently for each $1 \leq i \leq n$, until a majority completes
        if KVS $i$ is pending then wait until it responds
        $list \leftarrow$ list$_i()$
        $results \leftarrow results \cup list$
    $\langle seq_{\max}, id_{\max}\rangle \leftarrow \max(results)$
    $ver_w \leftarrow \langle seq_{\max} + 1, c\rangle$
    concurrently for each $1 \leq i \leq n$, until a majority completes
        if KVS $i$ is pending then wait until it responds
        putInKVS$(i, ver_w, val_w)$
    return ACK
```

function, which stores the versioned value to a majority of the KVSs. Finally, the function returns the read value (line 11). The atomic **write** operation is implemented by **regularWrite** from Algorithm 5.

Algorithm 6: Client c **read** operation of the atomic register

```
function atomicRead$_c()$
    $results \leftarrow \emptyset$
    concurrently for each $1 \leq i \leq n$, until a majority completes
        if KVS $i$ is pending then wait until it responds
        $result \leftarrow$ getFromKVS$(i)$
        $results \leftarrow results \cup \{result\}$
    choose $\langle ver, val\rangle \in results$ such that $ver' \leq ver$ for any $\langle ver', val'\rangle \in results$
    concurrently for each $1 \leq i \leq n$, until a majority completes
        if KVS $i$ is pending then wait until it responds
        putInKVS$(i, ver, val)$
    return $val$
```

5 Correctness

For establishing the correctness of the algorithms, note first that every client accesses the KVS objects in a well-formed manner, as ensured by the corresponding checks in Algorithm 3 (line 4), Algorithm 5 (lines 4 and 10), and Algorithm 6 (lines 4 and 8).

A global execution of the system consists of invocations and responses of two kinds: those of the emulated register and those of the KVS base objects. In order to distinguish between them, we let $\bar{\sigma}$ denote an execution of the register (with **read** and **write** operations) and let σ denote an execution of the KVS base objects (with **put**, **get**, **list**, and **remove** operations).

We say that a KVS-operation o is *induced* by a register operation $\bar{o}$ when the client executing $\bar{o}$ invoked o according to its algorithm for executing $\bar{o}$. Furthermore, a **read** operation *reads a version ver* when the returned value has been associated with *ver* (Algorithm 3 line 7), and a **write** operation *writes a version ver* when the induced **put** operation stores a value under a temporary key corresponding to *ver* (Algorithm 5 line 11).

At a high level, the register emulations are correct because the **read** and **write** operations always access a majority of the KVSs, and hence every two operations access at least one common KVS. Furthermore, each KVS stores two copies of a value under the eternal and under temporary keys. Because the algorithm for reading is carefully adjusted to the garbage-collection routine, every **read** operation returns a legitimate value in finite time. Section 5.1 below makes this argument precise for the regular register, and Section 5.2 addresses the atomic register.

5.1 MRMW-Regular Register

We prove safety (Theorem 3) and liveness (Theorem 6) for the emulation of the MWMR-regular register. Consider any execution $\bar{\sigma}$ of the algorithm, the induced execution σ of the KVSs, and a real-time sequential permutation π of σ (note that σ is determined by the operations on the atomic KVSs). Let π_i denote the sequence of actions from π that occur at some KVS replica i.

According to Algorithm 5, every **write** operation to the register induces exactly two **put** operations, one with a temporary key and one with the eternal key.

The following lemma shows that the largest version (corresponding to a temporary key) stored in every KVS always increases.

Lemma 1 (KVS version monotonicity). *Consider a write operation w that writes version ver and some operation **put**$_i$ in π_i induced by w with a temporary key. Then the response of any operation **list**$_i$ in π_i that follows **put**$_i$ contains at least one temporary key that corresponds to a version equal to or larger than ver.*

Proof. We show this by induction on the length of some prefix of π_i that is followed by an imaginary **list**$'$ operation. (Note that **list** does not modify the state of KVS i).

Initially, no versions have been written, and the claim is vacuously true for the empty prefix. According to the induction assumption, the claim holds for some prefix ρ_i. We argue that it also holds for every extension of ρ_i. When ρ_i is extended by a **put**$_i$ operation, the claim still holds. Indeed, the claim can only be affected when ρ_i is extended by an operation **remove**$_i$ with a key that corresponds to version *ver and* when no **put**$_i$ operation with a temporary key that corresponds to a larger version than *ver* exists in ρ_i.

A **remove**$_i$ operation is executed by some client that executes a **write** operation and function **putInKVS** in two cases. In the first case, when Algorithm 4 invokes operation **remove**$_i$ in line 5, it has previously executed **list**$_i$ and excluded from *obsolete* the temporary key corresponding to the largest version *ver*$'$. The induction assumption implies that *ver*$' \geq$ *ver*. Hence, there exists a temporary key corresponding to *ver*$' \geq$ *ver* also after **remove**$_i$ completes.

In the second case, when Algorithm 4 invokes **remove**$_i$ in line 9, then it has already stored a temporary key corresponding to a larger version than *ver* through operation **put**$_i$ (line 8), according to the algorithm. The claim follows. □

Lemma 2 (Partial order). *In an execution $\bar{\sigma}$ of the algorithm, the versions of the read and write operations in $\bar{\sigma}$ respect the partial order of the operations in $\bar{\sigma}$:*

*a) When a **write** operation w writes a version v_w and a subsequent (in $\bar{\sigma}$) **read** operation r reads a version v_r, then $v_w \leq v_r$.*

*b) When a **write** operation w_1 writes a version v_1 and a subsequent **write** operation w_2 writes a version v_2, then $v_1 < v_2$.*

Proof. For part a), note that both operations return only after receiving responses from a majority of KVSs. Suppose KVS i belongs to the majority accessed by the **putInKVS** function during w and to the majority accessed by r. Since $w \prec_{\bar{\sigma}} r$, the **put**$_i$ operation induced by w precedes the first **list**$_i$ operation induced by r. Therefore, the latter returns at least one temporary key corresponding to a version that is v_w or larger according to Lemma 1.

Consider now the execution of function **getFromKVS** (Algorithm 2) for KVS i. The previous statement shows that the client sets $v_0 \geq v_w$ in line 5. The function only returns a version that is at least v_0. As Algorithm 3 takes the maximal version returned from a KVS, the version v_r of r is not smaller than v_w.

The argument for the write operations in part b) is similar. Suppose that KVS i belongs to the majority accessed by the **putInKVS** function during w_1 and to the majority accessed by the **list** operation during w_2. As $w_1 \prec_{\bar{\sigma}} w_2$, the **put**$_i$ operation induced by w_1 precedes the **list**$_i$ operation induced by w_2. Therefore, the latter returns at least one temporary key corresponding to a version that is v_1 or larger according to Lemma 1. Hence, the computed previous maximum version $\langle seq_{\max}, id_{\max}\rangle$ of Algorithm 5 in w_2 is at least v_1. Subsequently, operation w_2 at client c determines its version $v_2 = \langle seq_{\max} + 1, c\rangle > \langle seq_{\max}, id_{\max}\rangle \geq v_1$. □

The two lemmas prepare the way for the following theorem. It shows that the emulation respects the specification of a multi-reader multi-writer regular register.

Theorem 3 (MRMW-regular safety). *Every well-formed execution $\bar{\sigma}$ of the MRMW-regular register emulation in Algorithms 3 and 5 is MRMW-regular.*

Proof. Note that a **read** only reads a version that was written by a **write** operation. We construct a sequential permutation $\bar{\pi}$ of $\bar{\sigma}$ by ordering all **write** operations of $\bar{\sigma}$ according to their versions and then adding all **read** operations after their matching **write** operation; concurrent **read** operations are added in arbitrary after, the others in the same order as in $\bar{\sigma}$.

Let r be a **read** operation in $\bar{\sigma}$ and denote by $\bar{\sigma}_r$ and by $\bar{\pi}_r$ the subsequences of $\bar{\sigma}$ and $\bar{pi}$ according to Definition 1, respectively. They contain only r and those **write** operations that do not follow r in $\bar{\sigma}$. We show that $\bar{\pi}_r$ is a legal real-time sequential permutation of $\bar{\sigma}_r$.

Due to the construction of $\bar{\pi}$, operation r returns the value written by the last preceding **write** operation or $\perp$ if there is no such **write**. The sequence $\bar{\pi}_r$ is therefore legal with respect to a register.

It remains to show that $\bar{\pi}_r$ respects the real-time order of $\bar{\sigma}_r$. Consider two operations o_1 and o_2 in $\bar{\sigma}_r$ such that $o_1 \prec_{\bar{\sigma}_r} o_2$. Hence, also $o_1 \prec_{\bar{\sigma}} o_2$. Note that o_1 and o_2 are either both **write** operations or o_1 is a **write** operation and o_2 is the **read** operation r. If o_1 is a **write** of a version v_1 and o_2 is a **write** of a version v_2, then Lemma 2a shows that $v_1 < v_2$. According to the construction of $\bar{\pi}$, we conclude that $o_1 \prec_{\bar{\pi}} o_2$. If o_1 is a **write** of a version v_1 and o_2 is a **read** of a version v_2, then Lemma 2b shows that $o_1 \prec_{\bar{\pi}} o_2$, again according to the construction of $\bar{\pi}$. By the construction of $\bar{\pi}_r$, this means that $o_1 \prec_{\bar{\pi}_r} o_2$. Hence, $\bar{\pi}_r$ is also a real-time sequential permutation of $\bar{\sigma}_r$. □

It remains to show that the register operations are also live. We first address the **read** operation, and subsequently the **write** operation.

Lemma 4 (Wait-free read). *Every **read** operation completes in finite time.*

Proof. The algorithm for reading (Algorithm 3) calls the function **getFromKVS** once for every KVS and completes after this call returns for a majority of the KVSs. As only a minority of KVSs may fail, it remains to show that when a client c invokes **getFromKVS** for a correct KVS i, it returns in finite time.

Algorithm 2 implements **getFromKVS**. It first obtains a list *list* of all temporary keys from KVS i and returns if no such key exists. If some temporary key is found, it determines the corresponding largest version ver_0 and enters a loop.

Towards a contradiction, assume that client c never exits the loop in some execution $\bar{\sigma}$ and consider the induced execution σ of the KVSs.

We examine one iteration of the loop. Note that since all operations of c are wait-free, the iteration eventually terminates. Prior to starting the iteration, the client determines *list* from an operation $\textbf{list}_i$. In line 8 the algorithm attempts to retrieve the value associated with key $v_c = \max(\textit{list})$ through an operation $\textbf{get}_c(v_c)$. This returns FAIL and the client retrieves the eternal key with an operation $\textbf{get}_c(\text{ETERNAL})$. We observe that $\textbf{list}_c \prec_\sigma \textbf{get}_c(v_c) \prec_\sigma \textbf{get}_c(\text{ETERNAL})$.

Since $\textbf{get}_c(v_c)$ fails, some client must have removed it from the KVS with a $\textbf{remove}(v_c)$ operation. Applying Lemma 1 to version v_c now implies that prior to the invocation of $\textbf{get}_c(v_c)$, there exists a temporary key in KVS i corresponding to a version $v_d > v_c$ that was stored by a client d. Denote the operation that stored v_d by $\textbf{put}_d(v_d)$. Combined with the previous observation, we conclude that

$$\textbf{list}_c \prec_\sigma \textbf{put}_d(v_d) \prec_\sigma \textbf{get}_c(v_c) \prec_\sigma \textbf{get}_c(\text{ETERNAL}). \tag{1}$$

Furthermore, according to Algorithm 4, client d has stored a tuple containing $v_d > v_c$ under the eternal key prior to $\textbf{put}_d(v_d)$ with an operation $\textbf{put}_d(\text{ETERNAL})$. But the subsequent $\textbf{get}_c(\text{ETERNAL})$ by client c returns a value containing a version *smaller* than v_c. Hence, there must be an *extra* client e writing concurrently, and its version-value pair has overwritten v_d and the associated value under the eternal key. This means that operation $\textbf{put}_e(\text{ETERNAL})$ precedes $\textbf{get}_c(\text{ETERNAL})$ in σ and stores a version $v_e < v_c$. Note that $\textbf{put}_e(\text{ETERNAL})$ occurs exactly once for KVS i during the write by e.

As client e also uses Algorithm 5 for writing, its *results* variable must contain the responses of **list** operations from a majority of the KVSs. Denote by $\textbf{list}_e$ its **list** operation whose response contains the largest version, as determined by e. Let $\textbf{list}_c^0$ denote the initial list operation by c that determined ver_0 in Algorithm 2 (line 5). We conclude that $\textbf{list}_e$ precedes $\textbf{list}_c^0$ in σ. Summarizing the partial-order constraints on e, we have

$$\textbf{list}_e \prec_\sigma \textbf{list}_c^0 \prec_\sigma \textbf{put}_e(\text{ETERNAL}) \prec_\sigma \textbf{get}_c(\text{ETERNAL}). \tag{2}$$

To conclude, in one iteration of the loop by reader c, some client d concurrently writes to the register according to (1). An extra client e concurrently writes as well and its **write** operation is invoked before $\textbf{list}_c^0$ and irrevocably makes progress after d invokes a **write** operation, according to (2). Therefore, client e may cause *at most one* extra iteration of the loop by the reader. Since there are only a finite number of such clients, client c eventually exits the loop. This contradicts the assumption that such an execution $\bar{\sigma}$ and the induced σ exist, and the lemma follows. □

Lemma 5 (Wait-free write). *Every **write** operation completes in finite time.*

Proof. The algorithm for writing (Algorithm 5) calls the function **list** for every KVS, and continues after this call returns for a majority of the KVSs. Then, it calls the function **putInKVS** for every KVS and returns after this call returns for a majority of the KVSs. As only a minority of KVSs may fail, it remains to show that when a client c invokes **putInKVS** for a correct KVS, it returns in finite time.

Algorithm 4 implements **putInKVS**. It calls **list**, possibly removes keys with **remove** and puts an eternal and possibly a temporary key in the KVS. Since all these operations are wait-free, the function returns in finite time. □

The next theorem summarizes these two lemmas and states that the emulation is wait-free.

Theorem 6 (MRMW-regular liveness). *Every **read** and **write** operation of the MRMW-regular register emulation in Algorithms 3 and 5 completes in finite time.*

5.2 Atomic Register

We state the correctness theorems for the atomic register emulation and sketch their proofs. The complete proofs are similar to the ones for the MRMW-regular register emulation.

Theorem 7 (Atomic safety). *Every well-formed execution $\bar{\sigma}$ of the atomic register emulation in Algorithms 6 and 5 is atomic.*

Proof sketch [8]. Note that a **read** operation can only read a version that has been written by some **write** operation. We therefore construct a sequential permutation $\bar{\pi}$ by ordering the operations in $\bar{\sigma}$ according to their versions, placing all **read** operations immediately after the **write** operation with the same version. Two concurrent **read** operations in $\bar{\sigma}$ that read the same version may appear in arbitrary order; all other **read** operations appear ordered in the same way as in $\bar{\sigma}$.

We show that $\bar{\pi}$ is a legal real-time sequential permutation of $\bar{\sigma}$. From the construction of $\bar{\pi}$, it follows that every **read** operation returns the value written by the last preceding **write** operation, after which it was placed. Therefore, $\bar{\pi}$ is a legal sequence of operations with respect to a register.

It remains to show that $\bar{\pi}$ respects the real-time order of $\bar{\sigma}$. Consider two operations o_1 and o_2 in $\bar{\sigma}$ such that $o_1 \prec_{\bar{\sigma}} o_2$. Operation o_1 is either a **write** or a **read** operation. In both cases, it completes only after storing its (read or written) version v_1 together with its value at a majority of the KVSs under a temporary key that corresponds to v_1. Operation o_2 is either a **write** or a **read** operation. In both cases, it first lists the versions in a majority of the KVSs and determines the maximal version among the responses. Let this maximal version be v_2. Because at least one KVS lies in the intersection of the two sets accessed by o_1 and by o_2, we conclude that $v_2 \geq v_1$. If o_2 is a **read** operation, it reads version v_2, and if o_2 is a **write** operation, it writes a version strictly larger than v_2. Therefore, according to the construction of $\bar{\pi}$, we obtain $o_1 \prec_{\bar{\pi}} o_2$ as required. □

Theorem 8 (Atomic liveness). *Every **read** and **write** operation of the atomic register emulation in Algorithms 6 and 5 completes in finite time.*

Proof sketch. The only difference between the regular and the atomic register emulations lies in the write-back step at the end of the **atomicRead** function. It is easy to see that storing the temporary key corresponding to the same version again may only effect the algorithm and its analysis in a minor way. In particular, the argument for showing Lemma 4 must be extended to account for concurrent **read** operations, which may also store values to the KVSs now. Similar to a concurrent **write** operation, an atomic **read** operation may delay a reader by one iteration in its loop. But again, there are only a finite number of clients writing concurrently. A **read** operation therefore completes after a finite number of steps. □

6 Efficiency

Our algorithms emulate a MRMW-regular and atomic registers from KVS base objects. The standard emulations of such registers use base objects with atomic read-modify-write semantics, which may receive versioned values and always retain the value with the largest version. Since a KVS has simpler semantics, our emulations store more than one value in each KVS. We discuss the space complexity of the algorithms in this section. We start by providing upper bounds in Section 6.1 and continue in Section 6.2 with a lower bound. The time complexity of our emulations follows from analogous arguments.

6.1 Maximal Space Complexity

It is obvious from Algorithm 5 that when a **write** operation runs in isolation (i.e., without any concurrent operations) and completes the **putInKVS** function on a set $\mathcal{C}$ of more than $n/2$ correct KVSs, then every KVS in $\mathcal{C}$ stores only the eternal key and one temporary key. Every such KVS has space complexity two.

When there are concurrent operations, the following theorem states an upper bound on the space complexity at a KVS.

Theorem 9. *In any execution $\bar{\sigma}$ of the MRMW-regular register emulation, the space complexity at any KVS object is at most linear in the number of* **write** *operations that are concurrent to each other in $\bar{\sigma}$.*

Proof. More precisely, we show that the maximal number of associated keys in a KVS object is two plus the number of concurrent **write** operations. The theorem is proven by considering the operations $o_1, o_2, \ldots$ of a legal real-time sequential permutation π of σ, the KVS execution induced by $\bar{\sigma}$.

If at some operation o_t the number of keys that is written to KVS i but not removed is x, then at some operation prior to o_t, at least x register operations were concurrently run. We prove by induction on t. Initially the claim holds since there are no keys put and no clients run. Assume it holds until o_{t-1} and prove for o_t. If operation o_t is not a **put**, then the number of put keys is the same as at o_{t-1} and the claim holds by the induction assumption.

If operation o_t is **put**$_i$, invoked by some client c, then it is performed by this client's **write**$_c$ that first removed all but one temporary keys in its GC routine (Algorithm 4 lines 4– 9). These **remove** operations precede the **put** in $\bar{\sigma}$, and therefore also its real-time sequential permutation π. All (except maybe one) versions that were written by **write**s that completed before **write**$_c$ are therefore removed before operation o_t. The temporary keys in the system at o_{t-1} are ones that were written by operations concurrent with **write**$_c$. The **put**$_c$ operation therefore increases their number by one, so the number of keys is at most the number of concurrent **write** operations, as required. $\square$

A similar theorem holds for the atomic register emulation, except here **read** operations may also increase the space complexity. The proof is similar to that of the regular register, and is omitted for brevity.

Theorem 10. *For any execution $\bar{\sigma}$, the maximal storage occupied by the atomic algorithm on a KVS i is at most linear in the concurrent number of operations.*

6.2 Minimal Space Complexity

We show that every emulation of even a *safe* [27] register, which is weaker than a regular register, from KVS base objects incurs space complexity two at the KVS objects.

Theorem 11. *In every emulation of a safe MRMW-register from KVS base objects, there exists some KVS with space complexity two.*

Proof. Toward a contradiction, suppose that every KVS stores only one key at any time.

Note that a client in an algorithm may access a KVS in an arbitrary way through the KVS interface. For modeling the limit on the number of stored values at a KVS, we assume that every **put** operation removes all previously stored keys and retains only the one stored by **put**. A client might still "compress" the content of a KVS by listing all keys, retrieving all stored values, and storing a representation of those values under one single key. In every emulation algorithm for the write operation, the client executes w.l.o.g. a "final" **put** operation on a KVS (if there is no such **put**, we add one at the end).

Note a client might construct the key and value to be used in a **put** operation from other values that it retrieved before. (Clearly, a practical KVS has a limit on the size of a key but the formal model does not.) For instance, a client might store multiple values by simply using them as the key in multiple put operations with empty values. This is allowed here and strengthens the lower bound.

Since operations are executed asynchronously and can be delayed, a client may invoke an operation at some time, at some later time the object (KVS) executes the operation atomically, and again at some later time the client receives the response.

In every execution of an operation with more than $n/2$ correct KVSs it is possible that all operations of some client invoked on less than $n/2$ KVSs are delayed until after one or more client operations complete.

Consider now an execution with three KVSs, denoted a, b, and c. Consider three executions α, β, and γ that involve three clients c_u, c_x, and c_r.

Execution α**.** Client c_x invokes **write**(x) and completes; let T^0_α be the point in time after that; suppose the final **put** operation from c_x on KVS b is delayed until after T^0_α; then b executes this **put**; let T^1_α be the time after that; suppose the corresponding response from b to c_x is delayed until the end of the execution.

Subsequently, after T^1_α, client c_r invokes **read** and completes; all operations from c_r to c are delayed until the end of the execution. Operation **read** returns x according to the register specification.

Execution β**.** Client c_x invokes **write**(x) and completes, exactly as in α; let T^0_β $(= T^0_\alpha)$ be the time after that; suppose the final **put** operation from c_x on KVS b is delayed until the end of the execution.

Subsequently, after T^0_β, client c_u invokes **write**(u) and completes; let T^1_β be the time after that; all operations from c_u to KVS c are delayed until the end of the execution.

Subsequently, after T^1_β, client c_r invokes **read** and completes; all operations from c_r to a are delayed until the end of the execution. Operation **read** by c_r returns u according to the register specification.

Execution γ**.** Client c_x invokes **write**(x) and completes, exactly as in β; let T^0_γ $(= T^0_\beta)$ be the time after that; suppose the final **put** operation from c_x to KVS b is delayed until some later point in time.

Subsequently, after T^0_γ, client c_u invokes **write**(u) and completes, exactly as in β; let T^1_γ $(= T^1_\beta)$ be the time after that; all operations from c_u to KVS c are delayed until the end of the execution.

Subsequently, after T^1_γ, the final **put** operation from c_x to KVS b induced by operation **write**(x) is executed at KVS b; let T^1_γ be the time after that; suppose the corresponding response from KVS b to c_x is delayed until the end of the execution.

Subsequently, after T^1_γ, client c_r invokes **read** and completes; all operations from c_r to KVS a are delayed until the end of the execution. The **read** by c_r returns u by specification. But the states of KVSs b and c at T^1_γ are the same as their states in α at T^0_α, hence, c_r returns x as in α, a contradiction. □

7 Conclusion

This paper investigates how to build robust storage abstractions from unreliable key-value store (KVS) objects, as they are commonly provided by many distributed "cloud" storage systems over the Internet. We provide an emulation of a multi-writer multi-reader regular register over a set of atomic KVSs; it supports an unbounded number of clients, who need not know each other and never interact among themselves directly.

Our algorithm is wait-free and robust against the crash failure of a minority of the KVSs and of any number of clients. The algorithm employs versioning and stores versioned values under two types of keys — an eternal key that is never removed, and temporary keys that are dynamically added and removed. This novel mechanism allows for efficient garbage collection of obsolete values, which ensures that the number of values stored in each KVS remains small and bounded. At the same time, the garbage collection allows every read operation to complete in a wait-free manner. Moreover, readers never need to store any values at a KVS — a property often desired due to security concerns. We also extend our emulation from a regular register to an atomic register.

In future work, we plan to explore the design space for robust storage algorithms when eventually consistent key-value stores [38] are used as base objects.

References

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

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

[3] M. K. Aguilera, I. Keidar, D. Malkhi, and A. Shraer. Dynamic atomic storage without consensus. In *Principles of Distributed Computing (PODC)*, pages 17–25, 2009.

[4] Amazon S3 availability event: July 20, 2008. `http://status.aws.amazon.com/s3-20080720.html`, retrieved April 26, 2011.

[5] Amazon gets 'black eye' from cloud outage. `http://www.computerworld.com/s/article/9216064/Amazon_gets_black_eye_from_cloud_outage`, retrieved April 26, 2011.

[6] Amazon Simple Storage Service. `http://aws.amazon.com/s3/`, retrieved April 26, 2011.

[7] E. Anderson, X. Li, A. Merchant, M. A. Shah, K. Smathers, J. Tucek, M. Uysal, and J. J. Wylie. Efficient eventual consistency in Pahoehoe, an erasure-coded key-blob archive. In *Proc. 40th International Conference on Dependable Systems and Networks (DSN-DCCS)*, 2010.

[8] H. Attiya, A. Bar-Noy, and D. Dolev. Sharing memory robustly in message-passing systems. *J. ACM*, 42(1):124–142, 1995.

[9] H. Attiya and J. Welch. *Distributed Computing: Fundamentals, Simulations and Advanced Topics*. John Wiley & Sons, 2004.

[10] A. Bessani, M. Correia, B. Quaresma, F. André, and P. Sousa. Depsky: Dependable and secure storage in a cloud-of-clouds. In *European Conference on Computer Systems (EuroSys)*, 2011.

[11] C. Cachin, R. Guerraoui, and L. Rodrigues. *Introduction to Reliable and Secure Distributed Programming (Second Edition)*. Springer, 2011.

[12] C. Cachin, R. Haas, and M. Vukolić. Dependable services in the intercloud: Storage primer. Research Report RZ 3783, IBM Research, Oct. 2010.

[13] G. Chockler and D. Malkhi. Active disk Paxos with infinitely many processes. *Distributed Computing*, 18:73–84, 2005. 10.1007/s00446-005-0123-x.

[14] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. In *Symposium on Operating System Principles (SOSP)*, pages 205–220, 2007.

[15] P. Dutta, R. Guerraoui, R. R. Levy, and M. Vukolic. Fast access to distributed atomic memory. *SIAM J. Comput.*, 39(8):3752–3783, 2010.

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

[17] Eucalyptus. `http://eucalyptus.com/`, retrieved April 26, 2011.

[18] R. Fan and N. A. Lynch. Efficient replication of large data objects. In *Distributed Computing (DISC)*, pages 75–91, 2003.

[19] E. Gafni and L. Lamport. Disk Paxos. *Distributed Computing*, 16(1):1–20, 2003.

[20] C. Georgiou, N. C. Nicolaou, and A. A. Shvartsman. Fault-tolerant semifast implementations of atomic read/write registers. *J. Parallel Distrib. Comput.*, 69(1):62–79, 2009.

[21] D. K. Gifford. Weighted voting for replicated data. In *Symposium on Operating System Principles (SOSP)*, pages 150–162, 1979.

[22] Gmail back soon for everyone. `http://gmailblog.blogspot.com/2011/02/gmail-back-soon-for-everyone.html`, retrieved April 26, 2011.

[23] M. Herlihy. Wait-free synchronization. *ACM Trans. Program. Lang. Syst.*, 13(1):124–149, 1991.

[24] M. P. Herlihy and J. M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12:463–492, July 1990.

[25] P. Jayanti, T. D. Chandra, and S. Toueg. Fault-tolerant wait-free shared objects. *J. ACM*, 45:451–500, May 1998.

[26] A. Lakshman and P. Malik. Cassandra: A decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44:35–40, 2010.

[27] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.

[28] L. Lamport. On interprocess communication. *Distributed Computing*, 1(2):77–101, 1986.

[29] N. A. Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1996.

[30] N. A. Lynch and A. A. Shvartsman. Robust emulation of shared memory using dynamic quorum-acknowledged broadcasts. In *Proceedings of the 27th Annual International Symposium on Fault-Tolerant Computing*, pages 272–281, 1997.

[31] N. A. Lynch and A. A. Shvartsman. Rambo: A reconfigurable atomic memory service for dynamic networks. In *Distributed Computing (DISC)*, pages 173–190, London, UK, 2002. Springer-Verlag.

[32] J. Maccormick, C. A. Thekkath, M. Jager, K. Roomp, L. Zhou, and R. Peterson. Niobe: A practical replication protocol. *ACM Transactions on Storage*, 3:1:1–1:43, Feb. 2008.

[33] Mezeo: Cloud storage platform. `http://www.mezeo.com/`, retrieved April 26, 2011.

[34] Rackspace hosting. `http://www.rackspacecloud.com/cloud_hosting_products/files/`, retrieved April 26, 2011.

[35] G. Roxana, A. A. Levy, T. Kohno, A. Krishnamurthy, and H. M. Levy. Comet: an active distributed key-value store. In *Proc. 9th Symp. Operating Systems Design and Implementation (OSDI)*, 2010.

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

[37] P. M. B. Vitányi and B. Awerbuch. Atomic shared register access by asynchronous hardware (detailed abstract). In *Proceedings of the 27th Annual Symposium on Foundations of Computer Science*, pages 233–243, 1986.

[38] W. Vogels. Eventually consistent. *Commun. ACM*, 52(1):40–44, 2009.

[39] Voldemort: A distributed database. `http://project-voldemort.com/`, retrieved April 26, 2011.

[40] Windows Azure Storage. `http://www.microsoft.com/windowsazure/storage/`, retrieved April 26, 2011.

[41] Y. Ye, L. Xiao, I.-L. Yen, and F. Bastani. Secure, dependable, and high performance cloud storage. In *Proc. 29th Symposium on Reliable Distributed Systems (SRDS)*, pages 194–203, 2010.