Robust Data Sharing with Key-Value Stores

Cristina Băsescu*, Christian Cachin†, Ittay Eyal‡, Robert Haas†, Alessandro Sorniotti†,
Marko Vukolić§, and Ido Zachevsky‡
*Vrije Universiteit Amsterdam, Amsterdam, The Netherlands. cbu200@few.vu.nl
†IBM Research - Zurich, Rüschlikon, Switzerland. {cca,rha,aso}@zurich.ibm.com
‡Dep. of Electrical Engineering, The Technion – Israel Inst. of Technology, Haifa, Israel. {ittay,ido}@tx.technion.ac.il
§Eurécom, Sophia Antipolis, France. vukolic@eurecom.fr

***Abstract*—A key-value store (KVS) offers functions for storing and retrieving values associated with unique keys. KVSs have become the most popular way to access Internet-scale "cloud" storage systems. We present an efficient wait-free algorithm that emulates multi-reader multi-writer storage from a set of potentially faulty KVS replicas in an asynchronous environment. Our implementation serves an unbounded number of clients that use the storage concurrently. It tolerates crashes of a minority of the KVSs and crashes of any number of clients. Our algorithm minimizes the space overhead at the KVSs and comes in two variants providing regular and atomic semantics, respectively. Compared with prior solutions, it is inherently scalable and allows clients to write concurrently.**

Because of the limited interface of a KVS, textbook-style solutions for reliable storage either do not work or incur a prohibitively large storage overhead. Our algorithm maintains *two* copies of the stored value per KVS in the common case, and we show that this is indeed necessary. If there are concurrent write operations, the maximum space complexity of the algorithm grows in proportion to the point contention. A series of simulations explore the behavior of the algorithm, and benchmarks obtained with KVS cloud-storage providers demonstrate its practicality.

I. Introduction

A. Motivation

In the recent years, the *key-value store* (*KVS*) abstraction has become the most popular way to access Internet-scale "cloud" storage systems. Such systems provide storage and coordination services for online platforms [1], [2], [3], [4], ranging from web search to social networks, but they are also available directly as products with Amazon S3, Microsoft Azure Storage, Rackspace hosting, and many others.

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.

This work is motivated by the idea of enhancing the dependability of cloud services by connecting multiple clouds to an *intercloud* or a *cloud-of-clouds*. Although existing KVS services provide high availability and reliability using replication internally, a KVS service is managed by one provider; many common components (and thus failure modes) affect its operation. A problem with any such component may lead to service outage or even to data being lost, as witnessed during an Amazon S3 incident [5], Google's temporary loss of email data [6], and Amazon's recent service disruption [7]. As a remedy, a client may increase data reliability by replicating it among several storage providers (all offering a KVS interface), using the guarantees offered by *robust* distributed storage algorithms [8], [9]. Data replication across different clouds is a topic of active research [10], [11], [12], [13].

B. Problem

Our data replication scheme relies on multiple providers of raw storage, called *base objects* here, and emulates a single, more reliable shared storage abstraction, which we model as a *read/write register*. A register represents the most basic form of storage, from which a KVS service or more elaborate abstractions may be constructed. The emulated register tolerates asynchrony, concurrency, and faults among the clients and the base objects. For increased parallelism, the clients do not communicate with each other for coordination, and they may not even be aware of each other.

Many well-known robust distributed storage algorithms exist (for an overview see [14]). They all use versioning [15], whereby each stored value is associated with a logical timestamp. For instance, with the multi-writer variant of the register emulation by Attiya et al. [9], the base objects perform custom *computation* depending on the timestamp, in order to identify and to retain only the newest written value. Without this an *old-new overwrite* problem might occur when a slow write request with an old value and a small timestamp reaches a base object after the latter has already updated its state to a newer value with a higher timestamp. On the other hand, one might let each client use its own range of timestamps and retain all versions of a written value at the KVSs [16], [17], but this approach is overly expensive in the sense that it requires as many base objects as there are clients. If periodic garbage collection (GC) is introduced

to reduce the consumed storage space, one may face a *GC racing* problem, whereby a client attempts to retrieve a value associated with a key that has become obsolete and was removed.

C. Contribution

We provide a robust, asynchronous, and space-efficient emulation of a register over a set of KVSs, which may fail by crashing. Our formalization of a key-value store (KVS) object represents the common denominator among existing commercial KVSs, which renders our approach feasible in practice. Inspired by Internet-scale systems, the emulation is designed for an unbounded number of clients and supports multiple readers and writers (MRMW). The algorithm is *wait-free* [18] in the sense that all operations invoked by a correct client eventually complete. It is also *optimally resilient*, i.e., tolerates the failure of any minority of the KVSs and of any number of clients.

We give two variations of the emulation. Our basic algorithm emulates a register with *regular* semantics in the multi-writer model [19]. It does not require read operations to write to the KVSs. Precluding readers from writing is practically appealing, since the clients may belong to different domains and not all readers may have write privileges for the shared memory. But it also poses a challenge because of the GC racing problem. Our solution stores 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 an old-new overwrite and (2) under a *temporary* key, named according to the version; obsolete temporary keys are garbage-collected by write operations, which makes these keys vulnerable to the GC racing problem. The algorithm for reading accesses the values in the KVSs according to a specific order, which guarantees that every read terminates eventually despite concurrent write operations. In a sense, the eternal and temporary copies complement each other and, together, guarantee the desirable properties of our emulation outlined above.

We then present an extension that emulates an *atomic* register [20]. It uses the standard approach of having the readers write back the returned value [9]. This algorithm requires read operations to write, but this is necessary [20], [21].

Our emulations maintain only two copies of the stored value per KVS in the common case (i.e., failure-free executions without concurrent operations). We show that this is also 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. Hence, our emulations have optimal space complexity.

Even though it is well-known how to implement a shared, robust multi-writer register from simpler storage primitives such as unreliable single-writer registers [21], our algorithm is the first to achieve an emulation from KVSs with the minimum necessary space overhead.

Note that some of the available KVSs export proprietary versioning information. However, one cannot exploit this for a data replication algorithm before the format and semantics of those versions has been harmonized. Another KVS prototype allows to execute client operations [22], but this technique is far from commercial deployment. We believe that some KVSs may also support atomic "read-modify-write" operations at some future time, thereby eliminating the problem addressed here. But until these extensions are deployed widely and have been standardized, our algorithm represents the best possible solution for minimizing space overhead of data replication on KVSs.

Last but not least, we simulate the algorithm with practical network parameters for exploring its properties. The results demonstrate that in realistic cases, our algorithm seldom increases the duration of read operations beyond the optimal duration. Furthermore, the algorithm scales to many concurrent writers without incurring any slowdown. We have also implemented our approach and report on benchmarks obtained with cloud-storage providers; they confirm the practicality of the algorithm.

Roadmap: The rest of the paper is organized as follows. We discuss related work in Section II and introduce the system model in Section III. In Section IV, we provide two robust algorithms that use KVS objects to emulate a read/write register. Section V analyzes the correctness of the algorithms and Section VI establishes bounds on their space usage. In Section VII we describe simulations of specific properties of the algorithms, and in Section VIII we report on benchmarks obtained with an implementation. For lack of space, detailed proofs have been omitted here and can be found in the full version [23].

II. 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, that is, they rely on atomic computation steps at the base objects. These include the single-writer multi-reader (SWMR) atomic wait-free register implementation of Attiya et al. [9], its dynamic multi-writer counterparts by Lynch and Shvartsman [24], [25] and Englert and Shvartsman [26], wait-free simulations of Jayanti et al. [27], low-latency atomic wait-free implementations of Dutta et al. [28] and Georgiou et al. [29], and the consensus-free versions of Aguilera et al. [30]. These solutions are not directly applicable to our model where KVSs are used as base objects, due to the old-new overwrite problem.

Notable exceptions that are applicable in our KVS context are SWMR regular register emulation by Gafni and Lamport [16] and its Byzantine variant by Abraham et al. [17] that use registers as base objects. However, transforming

these SWMR emulations to support a large number of writers is inefficient: standard register transformations [21], [14] 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 [31] 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. [32] 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. [10] propose RACS, an approach that casts RAID techniques to the KVS context. RACS uses a model different from ours and basically relies on a proxy between the clients and the KVSs, which may become a bottleneck and single point-of-failure. In a variant that supports multiple proxies, the proxies communicate directly with each other for synchronizing their operations. Bessani et al. [13] propose a distributed storage system, called DepSky, which employs erasure coding and cryptographic tools to store data on KVS objects prone to Byzantine faults. However, the basic version of DepSky allows only a single writer and thereby circumvents the problems addressed here. An extension supports multiple writers through a locking mechanism that determines a unique writer using communication among the clients. In comparison, the multi-writer versions of RACS and DepSky both serialize write operations, whereas our algorithm allows concurrent write operations from multiple clients in a wait-free manner. Therefore, our solution scales easily to a large number of clients.

III. Model

A. Executions

The system is comprised of multiple *clients* and *(base) objects*. We model them as I/O automata [33], 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*[1] 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). This *completes* the operation. We consider an asynchronous system, i.e., there are no timing assumptions that relate invocations and responses. (Consult [33], [21] 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* 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 $o_1 \prec_\pi o_2$.

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 [34].

B. Register Specifications

Sequential Register: A *register* [20] 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.

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

Multi-Reader Multi-Writer Regular Register: The following semantics describe a *multi-reader multi-writer regular register* (*MRMW-regular*), adapted from [19]. 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 be 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* [20], also called linearizability [18]. 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.*

C. 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 $|live|$.

D. 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.

Algorithm 1: Key-value store object i

```
1  state
2      live ⊆ K × X, initially ∅
3  On invocation put_i(key, value)
4      live ← (live \ {⟨key, x⟩ | x ∈ X}) ∪ ⟨key, value⟩
5      return ACK
6  On invocation get_i(key)
7      if ∃x : ⟨key, x⟩ ∈ live then
8          return x
9      else
10         return FAIL
11 On invocation remove_i(key)
12     live ← live \ {⟨key, x⟩ | x ∈ X}
13     return ACK
14 On invocation list_i()
15     return {key | ∃x : ⟨key, x⟩ ∈ live}
```

Our goal is to have the clients *emulate* a MRMW-regular register and an atomic register using the KVS base objects [33]. 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).

IV. Algorithm

A. Pseudo Code Notation

Our algorithm is formulated using functions that execute the register operations. They perform computation steps, invoke operations on the base objects, and may *wait for* such operations to complete. To simplify the pseudo code, we imagine there are concurrent execution "threads" as follows. When a function **concurrently** executes a block, it performs the same steps and invokes the same operations once for each KVS base object in parallel. An algorithm proceeds past a **concurrently** statement as indicated by a termination property; in all our algorithms, this condition requires that the block completes for a majority of base objects.

In order to maintain a well-formed execution, the system implicitly keeps track of pending operations at the base objects. Relying on this state, every instruction to **concurrently** execute a code block explicitly waits for a base object to complete a pending operation, before its "thread" may invoke another operation. This convention avoids cluttering the pseudo code with state variables and complicated predicates that have the same effect.

B. MRMW-Regular Register

We present an algorithm for implementing a MRMW-regular register, where **read** operations do not 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 [9]. 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[2] $\langle seq, id \rangle \in \mathbb{N}_0 \times \mathcal{ID}$, 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 $(\mathbb{N}_0 \times \mathcal{ID}) \times \mathcal{V}$.[3]

At the heart of our algorithm lies the idea of using *temporary keys*, which are created and later removed at the KVSs, and an *eternal key*, denoted ETERNAL, which is never removed. Both represent a register value and its associated version. When a client writes a value to the emulated register, it determines the new version to be associated with the value, 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. The data stored under a temporary key directly represents the written value; data stored under the eternal key contains the register value and its version. The writer also performs garbage collection of values stored under obsolete temporary keys, which ensures the bound on space complexity.

1) Read: When a client reads from the emulated register through algorithm **regularRead** (Algorithm 3), 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 i, the reader invokes a function **getFromKVS**(i) (shown in Algorithm 2). It 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 attempts to retrieve the value under the key representing the largest version. If the key exists, the reader has found a suitable value. However, this step may fail due to the GC racing problem, that is, because a concurrent writer has removed the particular key between the times when the client issues the **list** and the **get** operations.

In this case, the reader retrieves the version/value pair stored under the eternal key. As the eternal key is stored first by a writer and never removed, it exists always after the first write to the register. If the retrieved 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 has occurred, and the reader starts another iteration of the loop.

This loop terminates after a bounded number of iterations: Note that an iteration is not successful only if a GC race and an old-new overwrite have both occurred. But a concurrent writer that may cause an old-new overwrite must have invoked its write operation *before* the reader issued the first **list** operation on some KVS. Thus, the number of loop iterations is bounded by the number of clients that concurrently execute a **write** operation in parallel to the **read** operation (i.e., the point contention of **write** operations).

Algorithm 2: Retrieve a legal version-value pair

```
1  function getFromKVS(i)
2      list ← list_i() \ ETERNAL
3      if list = ∅ then
4          return ⟨⟨0, ⊥⟩, ⊥⟩
5      ver_0 ← max(list)
6      while True do
7          val ← get_i(max(list))
8          if val ≠ FAIL then
9              return ⟨max(list), val⟩
10         ⟨ver, val⟩ ← get_i(ETERNAL)
11         if ver ≥ ver_0 then
12             return ⟨ver, val⟩
13         list ← list_i() \ ETERNAL
```

Algorithm 3: Client c MRMW-regular **read** operation

```
1  function regularRead_c()
2      results ← ∅
3      concurrently for each 1 ≤ i ≤ n, until a majority completes
4          if an op. is pending at KVS i then wait for a response
5          result ← getFromKVS(i)
6          results ← results ∪ {result}
7      return val such that ⟨ver, val⟩ ∈ results and ver' ≤ ver for any
       ⟨ver', val'⟩ ∈ results
```

2) Write: A client writes a value to the register using algorithm **regularWrite** (Algorithm 5). First, the client lists the temporary keys in each base object and determines the largest version found in a majority of them. It increments this version and obtains a new version to be associated with the written value.

Then the client stores the value and the new version in all KVSs using a function **putInKVS**, shown in Algorithm 4, which also performs garbage collection. It first lists the existing keys and removes obsolete temporary keys, i.e.,

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

[3]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.

all temporary keys excluding the one corresponding to the maximal version. Subsequently the function stores the value and the version under the eternal key. To store the value under a temporary key, the algorithm 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.

Algorithm 4: Store a value and a given version

```
1 function putInKVS(i, ver_w, val_w)
2     list ← list_i()
3     obsolete ← {v | v ∈ list ∧ v ≠ ETERNAL ∧ v < max(list)}
4     foreach ver ∈ obsolete do
5         remove_i(ver)
6     put_i(ETERNAL, ⟨ver_w, val_w⟩)
7     if ver_w > max(list) then
8         put_i(ver_w, val_w)
9         remove_i(max(list))
```

Algorithm 5: Client c MRMW-regular **write** operation

```
1  function regularWrite_c(val_w)
2      results ← {⟨0, ⊥⟩}
3      concurrently for each 1 ≤ i ≤ n, until a majority completes
4          if an op. is pending at KVS i then wait for a response
5          list ← list_i()
6          results ← results ∪ list
7      ⟨seq_max, id_max⟩ ← max(results)
8      ver_w ← ⟨seq_max + 1, c⟩
9      concurrently for each 1 ≤ i ≤ n, until a majority completes
10         if an op. is pending at KVS i then wait for a response
11         putInKVS(i, ver_w, val_w)
12     return ACK
```

C. Atomic Register

The atomic register emulation results from extending the algorithm for emulating the regular register. Atomicity is achieved by having a client write back its read value before returning it, similar to the write-back procedure of Attiya et al. [9].

The **write** operation is the same as before, implemented by function **regularWrite** (Algorithm 5). The **read** operation is implemented by function **atomicRead** (Algorithm 6). Its first phase is unchanged from before and obtains the value associated with the maximal version found among a majority of the KVSs. Its second phase duplicates the second phase of the **regularWrite** function, which stores the versioned value to a majority of the KVSs.

Algorithm 6: Client c atomic **read** operation

```
1  function atomicRead_c()
2      results ← ∅
3      concurrently for each 1 ≤ i ≤ n, until a majority completes
4          if an op. is pending at KVS i then wait for a response
5          result ← getFromKVS(i)
6          results ← results ∪ {result}
7      choose ⟨ver, val⟩ ∈ results such that ver' ≤ ver for any
       ⟨ver', val'⟩ ∈ results
8      concurrently for each 1 ≤ i ≤ n, until a majority completes
9          if an op. is pending at KVS i then wait for a response
10         putInKVS(i, ver, val)
11     return val
```

V. Correctness

In this section we sketch the arguments for correctness of the MRMW-regular register. The correctness of the atomic register follow analogously. Details and complete proofs are available in a technical report.

We say 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 an induced **put** operation stores a value under a temporary key corresponding to *ver* (Algorithm 5 line 11).

Safety: Consider any execution $\bar{\sigma}$ of the algorithm, the induced execution σ of the KVSs (in terms of KVS operations), and a real-time sequential permutation π of σ. Denote by π_i the sequence of actions from π that occur at some KVS replica i.

We first establish that for every KVS, the maximums of the versions returned by consecutive **list** operations cannot decrease, despite the fact that **write** operations also remove versions.

Lemma 1 (KVS version monotonicity). *Consider a KVS i, 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.*

The next step ensures that the versions of the emulated **read** and **write** operations respect the partial order of the operations in the execution. It holds because **read** and **write** operations always access a majority of the KVSs, and hence every two operations access at least one common KVS.

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 $v_1 < v_2$.*

We may now construct a sequential permutation $\bar{\pi}$ of an execution $\bar{\sigma}$ by ordering all **write** operations of $\bar{\sigma}$ according to their versions and then adding all **read** operations after their matching **write** operations; concurrent **read** operations are added after after their respective **write**s in the same order as in $\bar{\sigma}$. The safety of the MRMW-regular register follows.

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.*

Liveness: The **write** routine obviously completes in finite time. The critical element is the **read** operation, for which we include a detailed proof.

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

Proof: We argue that when a client c invokes **getFromKVS** for a correct KVS i, it returns in finite time. Algorithm 2 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 its operations are wait-free and the iteration terminates. Prior to starting the iteration, the client determines *list* from an operation $\mathbf{list}_i$. In line 8 the algorithm attempts to retrieve the value associated with key $v_c = \max(list)$ through an operation $\mathbf{get}_c(v_c)$. This returns FAIL and the client retrieves the eternal key with an operation $\mathbf{get}_c(\text{ETERNAL})$. We observe that $\mathbf{list}_c \prec_\sigma \mathbf{get}_c(v_c) \prec_\sigma \mathbf{get}_c(\text{ETERNAL})$.

Since $\mathbf{get}_c(v_c)$ fails, some client must have removed it from the KVS with a $\mathbf{remove}(v_c)$ operation. Applying Lemma 1 to version v_c now implies that prior to the invocation of $\mathbf{get}_c(v_c)$, 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 $\mathbf{put}_d(v_d)$. Combined with the previous observation, we conclude that $\mathbf{list}_c \prec_\sigma \mathbf{put}_d(v_d) \prec_\sigma \mathbf{get}_c(v_c) \prec_\sigma \mathbf{get}_c(\text{ETERNAL})$.

Furthermore, according to Algorithm 4, client d has stored a tuple containing $v_d > v_c$ under the eternal key prior to $\mathbf{put}_d(v_d)$ with an operation $\mathbf{put}_d(\text{ETERNAL})$. But the subsequent $\mathbf{get}_c(\text{ETERNAL})$ 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 $\mathbf{put}_e(\text{ETERNAL})$ precedes $\mathbf{get}_c(\text{ETERNAL})$ in σ and stores a version $v_e < v_c$. Note that $\mathbf{put}_e(\text{ETERNAL})$ 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 $\mathbf{list}_e$ its **list** operation whose response contains the largest version, as determined by e. Let $\mathbf{list}_c^0$ denote the initial list operation by c that determined ver_0 in Algorithm 2 (line 5). We conclude that $\mathbf{list}_e$ precedes $\mathbf{list}_c^0$ in σ. Summarizing the partial-order constraints on e, we have $\mathbf{list}_e \prec_\sigma \mathbf{list}_c^0 \prec_\sigma \mathbf{put}_e(\text{ETERNAL}) \prec_\sigma \mathbf{get}_c(\text{ETERNAL})$.

Thus, in one iteration of the loop by reader c, some client d concurrently writes to the register. An extra client e has invoked a **write** operation before $\mathbf{list}_c^0$ and irrevocably makes progress after d invokes a **write** operation. 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, and the lemma follows. ■

VI. Efficiency

We discuss the space complexity of the algorithms in this section. Note how the algorithm for writing performs garbage collection on a KVS *before* storing a temporary key in the KVS. This is actually necessary for bounding the space at the KVS, since the **putInKVS** function is called concurrently for all KVSs and may be aborted for some of them. If the algorithm would remove the obsolete temporary keys *after* storing the value, the function may be aborted just before garbage collection. In this way, many obsolete keys might be left around and permantly occupy space at the KVS.

We provide upper bounds on the space usage in Section VI-A and continue in Section VI-B with a lower bound. The time complexity of our emulations follows from analogous arguments.

A. 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 space complexity may increase by one for every concurrent write operation, i.e., by the point contention of writes, because every write operation may add an additional temporary key. The formal proof is found in the full version.

Theorem 5. *The space complexity of the MRMW-regular register emulation at any KVS is at most two plus the point contention of concurrent write operations.*

The same bound can be shown for the atomic register emulation, except here **read** operations may also increase the space complexity.

B. Minimal Space Complexity

Theorem 6. *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 also construct the key to be used in a **put** operation from values that it retrieved before. For instance, a client might store multiple values by simply using them as the key in put operations with empty values. This is allowed here and strengthens the lower bound. (Clearly, a practical KVS has a limit on the size of a key but the formal model does not.)

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_α^0 be the point in time after that; suppose the final **put** operation from c_x on KVS b is delayed until after T_α^0; then b executes this **put**; let T_α^1 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_α^1, client c_r invokes **read** and completes with responses from b and c; all operations from c_r to a 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_β^0 $(= T_\alpha^0)$ 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_β^0, client c_u invokes **write**(u) and completes; let T_β^1 be the time after that; all operations from c_u to KVS c are delayed until the end of the execution.

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

Subsequently, after T_γ^1, the final **put** operation from c_x to KVS b induced by operation **write**(x) is executed at KVS b; let T_γ^2 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_γ^2, 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_γ^2 are the same as their states in α at T_α^1, hence, c_r returns x as in α, which contradicts the specification of the register. ■

VII. Simulation

To assess the properties of the algorithm, we analyze it through simulations under realistic conditions in this section. In particular, we demonstrate the scalability properties of our approach and compare it with a single-writer replication approach. In Section VIII, we also assert the accuracy of the simulator by comparing its output with that of experiments run with an implementation of the algorithm, which accessed actual KVS cloud-storage providers over the Internet.

We have built a dedicated event-driven simulation framework in Python for this task. The simulator models our algorithm for clients (Algorithms 2, 3, 4, and 5) and for KVS replicas (Algorithm 1). In each simulation run, one or more clients perform **read** and **write** operations using our register emulation.

A. Simulation Setup

The simulated system contains a varying number of clients and three KVS replicas. The time for a client to execute a KVS operation consists of three parts: (1) the time for the invocation message to reach a KVS replica; (2) the time for a KVS to execute the operation, always assumed to be 0; and (3) the time for the response message to reach the client. Message delays (1) and (3) are influenced by two factors: first, the *network latency* of the client, which we model as a random variable with exponential distribution with a given mean; and, second, by the *size* of the transferred value and the available *network bandwidth*. We assume that metadata is always of negligible size and consider only the size of the stored values.

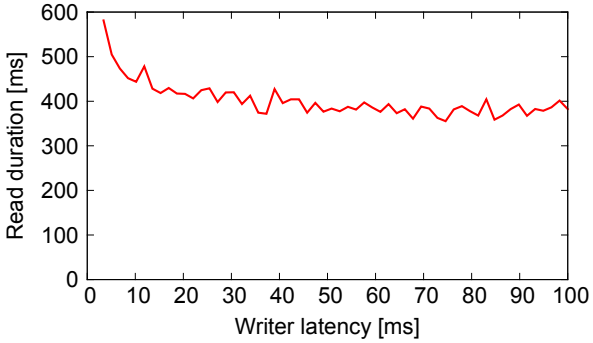

Figure 1. Simulation of the average duration of **read** operations shown with one concurrent writer accessing the KVS replicas at varying network latencies. The mean network latency of the reader is 100 ms; only when the writer has a much smaller latency does the **read** operations take longer than the expected minimum of 400 ms.

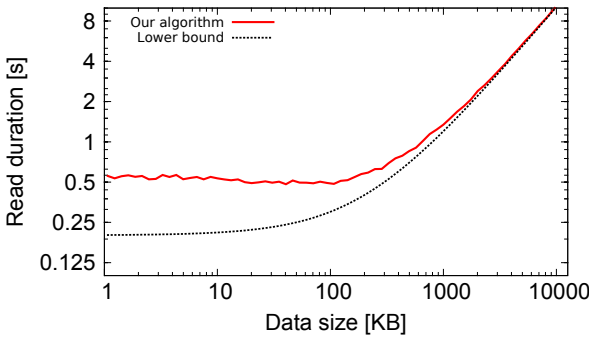

Figure 2. Simulation of the average duration of **read** operations as a function of the data size. For small values, the network latency dominates; for large value, the duration converges to the time for transferring the data.

As the base case for our explorations, we use a network latency with a mean of 100 ms. Unless stated differently, the network available to every client has 1 MBps bandwidth and the data size is small, namely 500 bytes.

The simulator drives the algorithm through **read** and **write** operations of the clients. Clients issue operations in a closed-loop manner: each client issues a new request only after it has received a response for the previous request. For measuring a statistic like the average duration of **read** and **write** operations, a run is simulated for some time, the number of completed operations is counted, and the average of the statistic per operation is output. The runs are sufficiently long to produce a reliable average.

B. Read Duration

Latency: A **read** operation takes at least two operations on the KVSs: an initial **list**, followed by at least one iteration of the loop in Algorithm 2. More iterations are needed only in the presence of concurrent **write** operations.

To observe this behavior, we run the simulation with a single writer and one reader. The two network latencies for the reader have a mean of 100 ms each. We vary the two network latencies of the writer from 2 ms to 100 ms in increments of 2 ms, to investigate a higher rate of **write** operations than **read** operations. Every average is computed from a simulation running for 40 s.

The average duration of the **read** operations is shown in Figure 1. As two network roundtrips are needed by every **read**, the minimum expected duration is 400 ms. We note that only when the writer's network latency is about 20 ms or less, will **read** operations take noticeably longer than their minimal duration. This corresponds to a writer that operates at least five times faster than the reader. However, an average **read** operation never exceeds 600 ms.

Data size: The second parameter that affects the **read** duration behavior is the data transfer time. We have already seen that for small values, **read** operations take longer than their minimal duration only in the presence of very fast **write** operations.

For this simulation, we let a fast writer with 1 ms mean network latency run concurrently to the reader. We vary the data size from 1 KB to 10 MB by multiplicative increments and simulate 16 data points for every 10-fold increase in size. We compare the average **read** duration of our algorithm to the theoretical lower bound, which is achieved by a non-robust algorithm that retrieves the value from one KVS.

The result is depicted in Figure 2. It shows that for small sizes, the network latency dominates the time for reading. Here, the read duration corresponds to the time needed for about three network roundtrips and matches the simulation of the reader's latency with much faster concurrent writes described previously. With larger sizes, the data transfer time becomes dominant, the **write** operations take longer, and the probability that the reader runs extra iterations of its loop decreases. For a data size of about 400 KB or more, our algorithm converges to the lower bound. This is because the value is transferred from the KVS only once, and the data transfer time dominates the operation duration.

C. Write Duration

This simulation addresses the scalability of **write** operations in the presence of multiple concurrent writers. We use a medium data size of 1 MB to illustrate the critical issue of **write** contention. With shorter values, the **put** operations finish quickly and we have not experienced much contention in preliminary simulations. For comparison we also simulate the performance of single-writer replication approaches, which have been considered in the related literature about data replication for cloud storage [10], [13]. These approaches provide the multi-writer capability by agreeing on a schedule with a single writer at any given time. In effect, this causes serial writes.

The network latencies for all writers are 100 ms; data size of 1 MB incurs a delay of 1 s because of the bandwidth

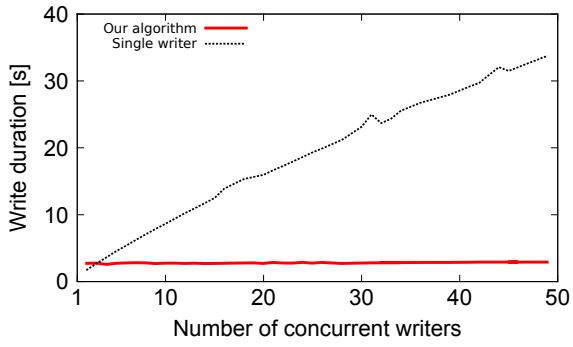

Figure 3. Simulation of the average duration of **write** operations as a function of the number of concurrent writers. The single-writer approach with serialized operations is shown for comparison.

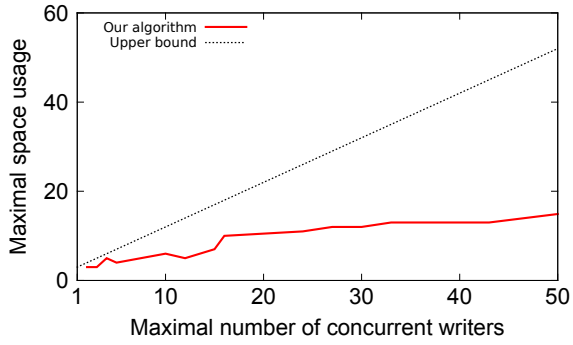

Figure 4. Simulation of the maximal space usage depending on the number of concurrent writers. The upper bound is the number of writers plus two according to Theorem 5.

constraint, which is imposed on the connection from every writer to the KVS replicas. Figure 3 shows the average duration of **write** operations invoked concurrently by a pool of clients, which grows from 1 to 50 clients. The averages are obtained by running the simulations for 30 s. The single-writer algorithm models **write** serialization through agreement, where we ignore the cost of reaching agreement.

For this simulation we use a batched garbage collection scheme, where a writing client invokes all **remove** operations concurrently. Although such a parallelization is impossible in our formal model, it is a practical optimization feasible with all KVS services we encountered.

The figure shows how the average duration of a **write** in our algorithm remains constant, even with many writers. In contrast, the time for writing in the single-writer approach obviously grows linearly with the number of concurrent writers.

D. Space Usage

To gain insight in the storage overhead, we measure the maximal space used at any KVS depending on the number of concurrently writing clients. The data size is 500 bytes, and the simulations are run for 50 s.

Figure 4 shows the *maximal* space usage at a KVS, where the number of concurrent writers increases from 1 to 50. Space usage is normalized to multiples of the data size. The upper bound from Theorem 5, given by the number of concurrent writers plus two, is included for comparison. The simulation shows that this bound is pessimistic and that the space used in practice is much smaller.

Further investigations show that the *average* space usage lies in the range of 2–5 in this simulation. This behavior can be explained by referring to the **write** algorithm. Concurrent writers indeed leave a large number of temporary keys behind, but the next writer removes all of them during garbage collection. As the time until removal is relatively short, the average space usage is small.

VIII. Implementation

A. Benchmarks

To evaluate the performances of **read** and **write** operations on cloud-storage KVSs in practice, we have implemented the algorithm in Java. The implementation uses the *jclouds* library (http://www.jclouds.org/), which supports more than a dozen practical KVS services.

Every client is initialized with a list of n accounts of KVS cloud-storage providers. The client library buffers operations on the KVSs as required by our model. Specifically, when a **read** or a **write** operation triggers a series of operations on the KVSs, these are appended to a dedicated FIFO queue for each one of the n KVSs; for each KVS, the implementation fetches the first operation from its queue and executes it as soon as the preceding one terminates.

The benchmark uses $n = 3$ KVS providers: Amazon S3, Microsoft Azure Storage, and Rackspace Cloudfiles. The client performs two **write** operations with the same key (so as to trigger the deletion of the first version) for 1000 different keys in closed-loop mode, followed by as many **read** operations with the keys written previously. We have instrumented the code to measure the completion time of the individual **list**, **put**, **get**, and **remove** operations as well as the duration of the **read** and **write** operations. The benchmark explores a data size ranging from 1 KiB to 10000 KiB in ten-fold increments.

Figures 5 and 6 show the results of the benchmark. Closer investigation of these times reveals that the duration of **read** operations is equal to the duration of the second-slowest **get** plus the duration of the second-slowest **list**. The reason is that the reader only waits for responses from a majority of the providers, and hence ignores the slowest response here. As for **write** operations, we observe that their duration equals twice the duration of the second-slowest **put** operation plus the duration of the second-slowest **list**. We also notice that **read** and **write** operations are faster than the slowest **get** and **put** operations: this can be seen in Figure 5, where Amazon S3 **get** operations are much

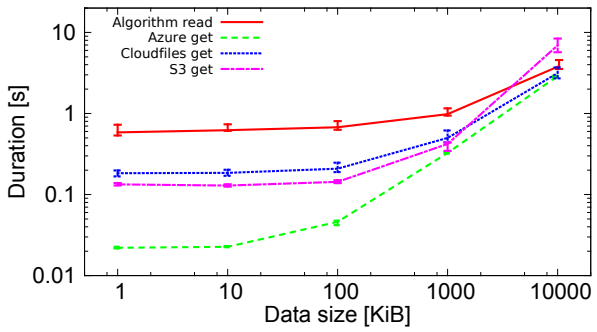

Figure 5. The median duration of **read** operations and **get** operations as the data size grows. The box plots also show the 30th and the 70th percentile.

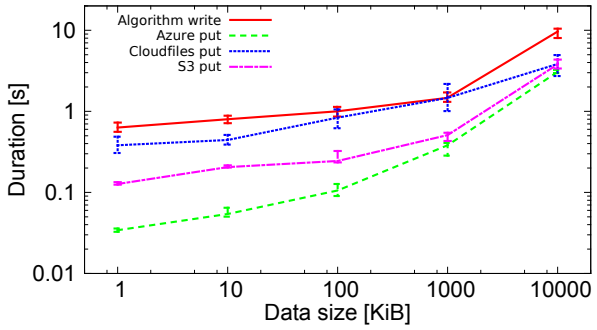

Figure 6. The median duration of **write** operations and **put** operations as the data size grows. The box plots also show the 30th and the 70th percentile.

slower than **read** operations for 10000 KiB data size, and in Figure 6, where Cloudfiles **put** operations are slightly slower than **write** operations for 1000 KiB input files.

B. Comparison of Simulation and Benchmarks

To compare the simulations with the behavior of the implemented system, we run an experiment with three KVS replicas and one client that performs 1000 **write** operations followed 1000 **read** operations. The data size is 2 MB. The same scenario is simulated with parameters set to values that were obtained from the experiment.

In particular, the simulation uses the same model as described before, with exponentially distributed network latencies for KVS operations. We measured the network latency of KVS operations excluding the time for data transfer. We assume that the invocation and response latencies of the simulated operations are symmetric and set their mean to half of the measured network latency. Furthermore, we determined the bandwidth of every KVS provider from the measurements of **put** and **get** operations.

For **get** and **put**, the mean network latency for the KVSs is set to 39.4 ms, 90.4 ms, and 81.2 ms, respectively. For

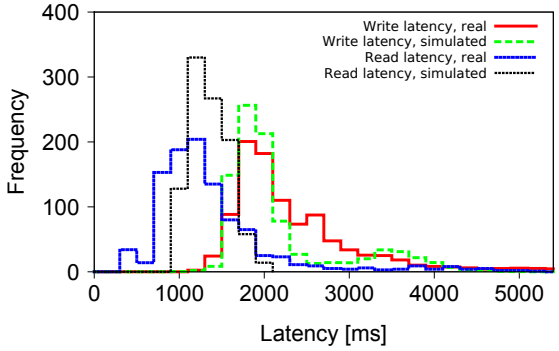

Figure 7. Comparison of the duration of **read** and **write** operations for the real system (solid lines) and the simulated system (dotted lines). The graph shows a histogram of the operation durations for 1000 **read** operations (centered at about 1200 ms) and 1000 **write** operations (centered at about 1800 ms).

list, the mean network latency is 36.5 ms, 181.1 ms, and 130.9 ms; and for **remove**, network latency is 18.5 ms, 100 ms, and 59.5 ms. The bandwidth limitations for the providers are 6.67 MBps, 2.33 MBps, and 1.5 MBps, respectively.

Figure 7 compares the durations of **read** and **write** operations in the experiment and the simulation. The graphs show a good match between the experimental system and the simulation. This reinforces the confidence in the simulation results.

IX. Conclusion

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

The algorithm is wait-free and robust against the crash failure of a minority of the KVSs and of any number of clients. The algorithm 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 garbage collection of obsolete values in parallel to wait-free client operations. Simulations and benchmarks with actual cloud-storage providers demonstrate that the algorithm works well under practical circumstances.

For ease of exposition, we have assumed atomic semantics of KVSs, but practical KVSs may only provide eventual consistency [35]. To address this question we have run extensive experiments and never observed non-atomic behavior; note that some cloud providers already provide atomic operations [36]. We plan to investigate this important issue in future work.

ACKNOWLEDGMENTS

We are grateful to Birgit Junker and to Sabrina Pérez for their contributions to the implementation of the algorithm.

This work has been supported in part by the European Commission through the ICT programme under contracts ICT-2007-216676 ECRYPT II and ICT-2009-257243 TClouds.

REFERENCES

[1] 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 *SOSP'07*, 2007.

[2] 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 *DSN-DCCS'10*, 2010.

[3] A. Lakshman and P. Malik, "Cassandra: A decentralized structured storage system," *SIGOPS Op. Sys. Review*, vol. 44, pp. 35–40, 2010.

[4] "Voldemort: A distributed database," http://project-voldemort.com/.

[5] "Amazon S3 availability event: July 20, 2008," http://status.aws.amazon.com/s3-20080720.html, retrieved Dec. 6, 2011.

[6] "Gmail back soon for everyone," http://gmailblog.blogspot.com/2011/02/gmail-back-soon-for-everyone.html, retrieved Dec. 6, 2011.

[7] "Amazon gets 'black eye' from cloud outage," http://www.computerworld.com/s/article/9216064/Amazon_gets_black_eye_from_cloud_outage, retrieved Dec. 6, 2011.

[8] D. K. Gifford, "Weighted voting for replicated data," in *SOSP'79*, 1979.

[9] H. Attiya, A. Bar-Noy, and D. Dolev, "Sharing memory robustly in message-passing systems," *J. ACM*, vol. 42, no. 1, pp. 124–142, 1995.

[10] H. Abu-Libdeh, L. Princehouse, and H. Weatherspoon, "RACS: a case for cloud storage diversity," in *SoCC'10*, 2010.

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

[12] J. K. Resch and J. S. Plank, "AONT-RS: Blending security and performance in dispersed storage systems," in *FAST'11*, 2011.

[13] A. Bessani, M. Correia, B. Quaresma, F. André, and P. Sousa, "Depsky: Dependable and secure storage in a cloud-of-clouds," in *EuroSys'11*, 2011.

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

[15] P. M. B. Vitányi and B. Awerbuch, "Atomic shared register access by asynchronous hardware (detailed abstract)," in *FOCS'86*, 1986.

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

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

[18] M. P. Herlihy and J. M. Wing, "Linearizability: a correctness condition for concurrent objects," *ACM Trans. on Prog. Lang. and Sys.*, vol. 12, pp. 463–492, July 1990.

[19] C. Shao, E. Pierce, and J. L. Welch, "Multi-writer consistency conditions for shared memory objects," in *DISC'03*, 2003, pp. 106–120.

[20] L. Lamport, "On interprocess communication," *Dist. Comp.*, vol. 1, no. 2, pp. 77–101, 1986.

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

[22] R. Geambasu, A. A. Levy, T. Kohno, A. Krishnamurthy, and H. M. Levy, "Comet: an active distributed key-value store," in *OSDI'10*, 2010.

[23] C. Băsescu, C. Cachin, I. Eyal, R. Haas, A. Sorniotti, M. Vukolić, and I. Zachevsky, "Robust data sharing with key-value stores," IBM Research, Research Report RZ 3802, 2011.

[24] N. A. Lynch and A. A. Shvartsman, "Robust emulation of shared memory using dynamic quorum-acknowledged broadcasts," in *FTCS'97*, 1997.

[25] S. Gilbert, N. Lynch, and A. Shvartsman, "Rambo: A reconfigurable atomic memory service for dynamic networks," *Dist. Comp.*, vol. 23, pp. 225–272, 2010.

[26] B. Englert and A. A. Shvartsman, "Graceful quorum reconfiguration in a robust emulation of shared memory," in *ICDCS'00*, 2000.

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

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

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

[30] M. K. Aguilera, I. Keidar, D. Malkhi, and A. Shraer, "Dynamic atomic storage without consensus," *J. ACM*, vol. 58, pp. 7:1–7:32, April 2011.

[31] G. Chockler and D. Malkhi, "Active disk Paxos with infinitely many processes," *Dist. Comp.*, vol. 18, pp. 73–84, 2005.

[32] Y. Ye, L. Xiao, I.-L. Yen, and F. Bastani, "Secure, dependable, and high performance cloud storage," in *SRDS'10*, 2010.

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

[34] M. Herlihy, "Wait-free synchronization," *ACM Trans. Prog. Lang. and Sys.*, vol. 13, no. 1, pp. 124–149, 1991.

[35] W. Vogels, "Eventually consistent," *Comm. ACM*, vol. 52, no. 1, pp. 40–44, 2009.

[36] B. Calder, J. Wang, A. Ogus, N. Nilakantan, A. Skjolsvold, S. McKelvie *et al.*, "Windows Azure Storage: A highly available cloud storage service with strong consistency," in *SOSP'11*, 2011.