Exploring Key-Value Stores in Multi-Writer Byzantine-Resilient Register Emulations

Tiago Oliveira, Ricardo Mendes and Alysson Bessani

DI–FCUL–TR–2016–02

DOI:10451/25098

(http://hdl.handle.net/10451/25098)

November 2016

Publication without review in the repository of the Department of Informatics
of the University of Lisbon, Faculty of Sciences
(http://repositorio.ul.pt/handle/10451/12254).

Exploring Key-Value Stores in Multi-Writer Byzantine-Resilient Register Emulations*

Tiago Oliveira, Ricardo Mendes, and Alysson Bessani
LaSIGE, Faculdade de Ciências, Universidade de Lisboa – Portugal

Abstract

Resilient register emulation is a fundamental technique to implement dependable storage and distributed systems. In data-centric models, where servers are modeled as fail-prone base objects, classical solutions achieve resilience by using fault-tolerant quorums of read-write registers or read-modify-write objects. Recently, this model has attracted renewed interest due to the popularity of cloud storage providers (e.g., Amazon S3, Google Storage, Microsoft Azure Storage), that can be modeled as key-value stores (KVSs) and combined for providing secure and dependable multi-cloud storage services. In this paper we present three novel wait-free multi-writer multi-reader regular register emulations on top of Byzantine-prone KVSs. We implemented and evaluated these constructions using five existing cloud storage services and show that their performance matches or surpasses existing data-centric register emulations.

Keywords: Byzantine fault tolerance, register emulation, multi-writer, key-value store, data-centric algorithms

1 Introduction

Resilient register emulations on top of message passing systems are a cornerstone of fault-tolerant storage services. These emulations consider the provision of shared objects supporting read and write operations executed by a set of clients. In the traditional approach, these objects are implemented in a set of fail-prone servers (or replicas) that run some specific code for the emulation [9, 14, 18, 21, 22, 28–30, 34].

A less explored approach, dubbed *data-centric*, does not rely on servers that can run arbitrary code, but on passive replicas modeled as base objects that provide a constrained interface. These base objects can be as simple as a network-attached disk, a remote addressable memory, or a queue, or as complex as a transactional database, or a full-fledged cloud storage service. By combining these fail-prone base objects, one can build fault-tolerant services for storage, consensus, mutual exclusion, etc, using only *client-side code*, leading to arguably simpler and more manageable solutions.

The data-centric model has been discussed since the 90s [24], but the area gained visibility and practical appeal only with the emergence of network-attached disks technology [20]. In particular, several theoretical papers tried to establish lower bounds and impossibility results for implementing resilient read-write registers and consensus objects considering different types of fail-prone base objects (read-write registers [8, 19] vs. read-modify-write objects [16, 17]) under both crash and Byzantine fault models [7]. More recently, there has been a renewed interest in data-centric algorithms for the cloud-of-clouds model [11, 33]. In this model, the base objects are cloud services (e.g., Amazon S3, Microsoft Azure Blob Storage) that offer interfaces similar to read-write registers or *key-value stores* (KVSs). This approach ensures that the stored data is available even if a subset of cloud providers is unavailable or corrupts their copy of the data (events that do occur in practice [27]).

To the best of our knowledge, there are only two existing works for register emulation in the cloud-of-clouds model: DepSky [11], which tolerates Byzantine faults (e.g., data corruption or cloud misbehavior) on the providers but *supports only a single-writer per data object*, and Basescu et al. [10], which genuinely supports multiple writers, but *tolerates only crash faults and does not support erasure codes.*

In this paper we present new register emulations on top of cloud storage services that support multiple concurrent writers (avoiding the need for expensive mutual exclusion algorithms [11]), tolerate Byzantine

*This is an extended version of a paper with the same title that appeared on *Proceedings of the 20th International Conference On Principles Of DIstributed Systems - OPODIS'16*, in December 2016. The main difference here are the proofs that appear on the appendix.

failures in base objects (minimizing the trust assumptions on cloud providers), and integrate erasure codes (decreasing the storage requirements significantly). In particular, we present three new multi-writer multi-reader (MWMR) regular register constructions:

1. an optimally-resilient register using full replication;
2. a register construction requiring more base objects, but achieving better storage-efficiency through the use of erasure codes;
3. another optimally-resilient register emulation that also supports erasure codes, but requires additional communication steps for writing.

These constructions are wait-free (operations terminate independently of other clients), uniform (they work with any number of clients), and can be adapted to provide atomic (instead of regular) semantics.

We achieve these results by exploring an often overlooked operation offered by KVSs – *list* – which returns the set of stored keys. The basic idea is that by embedding data integrity and authenticity proofs in the key associated with a written value, it is possible to use the list operation in multiple KVSs to detect concurrent writers and establish the current value of a register. Although KVSs are equivalent to registers in terms of synchronization power [13], the existence of the list operation in the interface of the former is crucial for our algorithms.

Besides the reduction on the storage requirements, an additional benefit of supporting erasure codes when untrusted cloud providers are considered is that they can be substituted by a secret sharing primitive (e.g., [25]) or any privacy-aware encoding (e.g., [14, 32]), ensuring confidentiality of the stored data.

The three constructions we propose are described, proved correct, implemented and evaluated using real clouds (Amazon S3 [1], Microsoft Azure Storage [15], Rackspace Cloud Files [5], Google Storage [3] and Softlayer Cloud Storage [6]). Our experimental results show that these novel constructions provide advantages both in terms of latency and storage costs.

2 Related Work

Existing fault-tolerant register emulations can be divided in two main groups depending on the nature of the fail-prone "storage blocks" that keep the stored data. The first group comprises the works that rely on servers capable of running part of the protocols [9, 18, 22, 28, 34], i.e., constructions that have both a client-side and a server-side of the protocol. Typically, in this kind of environment it is easier to provide robust solutions as servers can execute specific steps of the protocol atomically, independently of the number of clients accessing it.

In the second group we have the *data-centric* protocols [7, 8, 17, 19, 24]. This approach considers a set of clients interacting with a set of passive servers with a constrained interface, modeled as shared memory objects (called base objects). The first work in this area was due to Jayanti, Chandra and Toueg [24], where the model was defined in terms of fail-prone shared memory objects. This work presented, among other wait-free emulations [23], a Byzantine fault-tolerant single-writer single-reader (SWSR) safe-register construction using $5f + 1$ base objects to tolerate f faults. Further works tried to establish lower bounds and impossibility results for emulating registers tolerating different kinds of faults considering different types of base objects. For example, Aguilera and Gafni [8] and Gafni and Lamport [19] used regular and/or atomic registers to implement crash-fault-tolerant MW and SW registers,[1] respectively, while Chockler and Malkhi [17] used read-modify-write objects to transform the SW register of Gafni and Lamport [19] in a ranked register, a fundamental abstraction for implementing consensus. Abraham et al. [7] provided a Byzantine fault-tolerant SW register, which was latter used as a basis to implement consensus. The main limitation of these algorithms is that, although they are asymptotically efficient [8], the number of communication steps is still very large, and the required base objects are sometimes stronger than KVSs [17] or implement weak termination conditions [7].

More recently, there has been a renewed interest in data-centric algorithms for the cloud-of-clouds model [10, 11]. Here the base objects are cloud services offering interfaces similar to key-value stores. These solutions ensure that the stored data is available even if a subset of cloud providers is unavailable or corrupts their copy of the data. DepSky [11] provided a regular SW register construction that tolerates Byzantine faults by less than a third of the base objects, ensuring also the confidentiality of the stored data by using a secret sharing scheme [25]. However, to support multiple writers an expensive lock

[1] From now on we avoid characterizing the constructions about the number of readers, as all constructions discussed in the rest of the paper support multiple readers (MR).

Table 1: Data-centric resilient register emulations. * Can be extended to achieve atomic semantics.

Work	Fault model	Technique	Base Objects	Resilience	Semantics
Jayanti et al. [24]	Byzantine	replication	atomic registers	$5f+1$	SW safe
Gafni and Lamport [19]	crash	replication	atomic registers	$2f+1$	SW regular
Chockler and Malkhi [17]	crash	replication	rmw registers	$2f+1$	MW ranked
Abraham et al. [7]	Byzantine	replication	regular registers	$3f+1$	SW regular
	Byzantine	replication	regular registers	$3f+1$	SW safe
Aguilera and Gafni [8]	crash	replication	atomic registers	$2f+1$	MW atomic
Bessani et al. [11]	Byzantine	replication	regular registers	$3f+1$	SW regular
	Byzantine	erasure code	regular registers	$3f+1$	SW regular
Basescu et al. [10]	crash	replication	atomic KVSs	$2f+1$	MW regular*
This paper	Byzantine	replication	atomic KVSs	$3f+1$	MW regular*
	Byzantine	erasure code	atomic KVSs	$4f+1$	MW regular*
	Byzantine	erasure code	atomic KVSs	$3f+1$	MW regular*

protocol must be executed to coordinate concurrent accesses. Another work in this line [10] provided a regular MW register that replicates the data by a majority of KVSs. Its main purpose was to reduce the necessary storage requirements. To achieve that, writers remove obsolete data synchronously, creating the need to store each version in two keys: a temporary key, that could be removed, and an eternal key, common for all writers and versions, that is never erased. In the best case, the algorithm requires a storage space of $2 \times S \times n$, where S is the size of the data and n is the number of KVSs.

Using registers or KVSs as base objects in the data-centric model makes it more challenging to implement dependable register emulations, as general replicas have more synchronization power than such objects [13]. The three new register constructions presented in this paper advance the state of the art by supporting multiple writers and erasure-coded data in the data-centric Byzantine model, using a rather weak base object – a KVS. Two of these constructions have optimal resilience, as they require $3f+1$ base objects to tolerate f Byzantine faults in an asynchronous system (with confirmable writes) [30]. Table 1 summarizes the discussed data-centric constructions.

3 System Model

3.1 Register Emulation

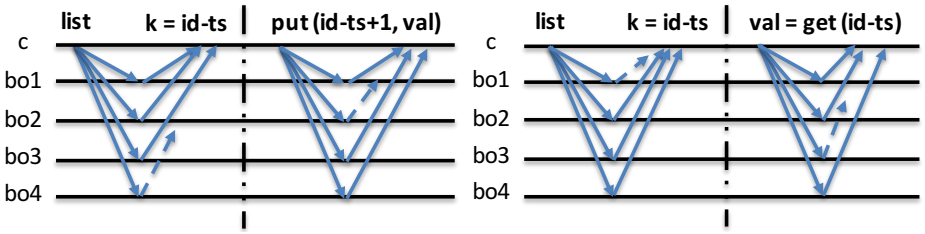
We consider an *asynchronous* system composed of a finite set of clients and n cloud storage providers that provide a KVS interface. We refer to clients as *processes* and to cloud storage providers as *base objects*. Each process has a unique identifier from an infinite set named *IDs*, while the base objects are numbered from 0 to $n-1$.

We aim to provide *MW-register* abstractions on top of n base objects. Concretely, a register abstraction offers an interface specification composed of two *operations*: **write(v)** and **read()**. The sequential specification of a register requires that a read operation returns the last value written, or $\perp$ if no write has ever been executed. Processes interacting with registers can be either *writers* or *readers*.

A process operation starts by an *invoke* action on the register, and ends with a *response*. An operation *completes* when the process receives the response. An operation o_1 *precedes* another operation o_2 (and o_2 *follows* o_1) if it completes before the invocation of o_2. Operations with no precedence relation, are called *concurrent*.

Unless stated otherwise, the register implementations should be *wait-free* [23], i.e., the operation invocations should complete in a finite number of internal steps. Moreover, we provide *uniform* implementations, i.e., implementations that do not rely on the number of processes, allowing processes to not know each other initially.

We provide two register abstraction semantics, *regular* and *atomic*, which differ mainly in the way they deal with concurrent accesses [26]. A regular register guarantees only that different read operations agree on the order of preceding write operations. Any read operation overlapping a write operation may return the value being written or the preceding value. An atomic register employs a stronger consistency notion than regular semantics. It stipulates that it should be possible to place each operation at a singular point (linearization point) between its invocation and response. This mean that after a read operation completes, a following read must return at least the version returned in the preceding read, even in the presence of concurrent writes.



(a) WRITE (b) READ

Figure 1: General structure of our MW-regular register emulations.

3.2 Threat Model

Up to f out-of n base objects can be subject to NR-arbitrary failures [24], which are also known as Byzantine failures. The behavior of such objects can be unrestricted: they may not respond to an invocation, and if they do, the content of the response may be arbitrary. Unless stated otherwise, readers may also be subject to Byzantine failures. Writers can only fail by crashing, because even if the protocol tolerates Byzantine writers, they may always store arbitrary values or overwrite data on the register. Processes and base objects are said to be *correct* if they do not fail.

For cryptography, we assume that each writer has a private key K_r to sign some of the information stored on the base objects. These signatures can be verified by any process in the system through the corresponding public key K_u. Moreover, we also assume the existence of a collision-resistant cryptographic hash function to ensure integrity. There might be multiple writer keys as long as readers can access their public counterparts.

3.3 Key-Value Store Specification

Current cloud storage service providers offer a key-value store (KVS) interface, which act as a passive server where it is impossible to run any code, forcing the implementations to be *data-centric*. Specifically, KVSs allow customers to interact with associative arrays, i.e, with a collection of $\langle key, value \rangle$ pairs, where any *key* can have only one value associated at a time and there can not be equal keys. Moreover, the size of stored values are expected to be much larger than the size of the associated keys. We assume the presence of four operations: (1) **put**(k, v), (2) **get**(k), (3) **list**(), and (4) **remove**(k). The first operation associates a key k with the value v, returning *ack* if successful and *ERROR* otherwise; the second retrieves the value associated with a key k, or *ERROR* if the key does not exist; the third returns an array with all the keys in the collection, or [] if there are no keys in the collection; and the last operation disassociates a key k from its value, releasing storage space and the key itself, returning an *ack* if successful and *ERROR* otherwise. Finally, we assume that individual KVS's operations are atomic and wait-free.

4 Multi-Writer Constructions

In this section we describe the three MW-regular register implementations. Before discussing the algorithms in detail (§4.3 to §4.6), we present an overview of the general structure of the protocols (§4.1) and describe the main techniques employed in their construction (§4.2). The correctness proofs of the protocols are presented in the appendix.

4.1 Overview

Our three MW-regular protocols differ mainly in the storage technique employed (replication or erasure code), the number of base objects required ($3f + 1$ or $4f + 1$), and the number of sequential base object accesses (two or three steps). Excluding these differences, the general structure of all protocols is similar to the one illustrated in Figure 1.

In the write operation, the client first lists a quorum of base objects (KVSs) in order to find the key encoding the most recent version written in the system, and then puts the value being written associated with a unique key encoding a new (incremented) version in a quorum. The read operation requires finding

the most recent version of the object (as in the first phase of the write operation), and then retrieving the value associated with that key.

Notice that our approach considers that each written value requires a new key-value pair in the KVSs. However, it is impossible to implement wait-free data-centric MW-regular register emulations without using at least one "data element" per written version if the base objects do not provide conditional update primitives (similar to Compare-and-Swap) [10, 16]. Therefore, any practical implementation of these algorithms must consider some form of garbage collection, as discussed in §5.2.

4.2 Protocols Mechanisms

Our algorithms use a set of mechanisms that are crucial for achieving Byzantine fault tolerance, MW semantics and storage efficiency. To simplify the exposition of the algorithms in the following sections (§4.4 to §4.6), we first describe such mechanisms.

4.2.1 Byzantine Quorum Systems

Our protocols employ dissemination and masking Byzantine quorum systems to tolerate up to f Byzantine faults [28]. *Dissemination quorum systems* consider quorums of $q = \lceil \frac{n+f+1}{2} \rceil$ base objects, requiring thus a total of $n > 3f$ base objects in the system. This ensures each two quorums intersect in at least $f + 1$ objects (one correct). *Masking quorum systems* require quorums of size $q = \lceil \frac{n+2f+1}{2} \rceil$ and a total of $n > 4f$ base objects, ensuring thus quorum intersections with at least $2f + 1$ base objects (a majority of correct ones).

4.2.2 Multi-Writer Semantics

We use the **list** operation of KVSs to design MW uniform implementations. This operation is very important as it allows us to discover new versions written by unknown clients. With this, the key idea of our protocols is making each writer to write in its own abstract register in a similar way to what is done in traditional transformation of SW to MW registers [26]. We achieve this by putting the client unique id on each key alongside with a timestamp ts, resulting in the pair $\langle ts, id \rangle$, which represents a *version*. This approach ensures that clients writing new versions of the data never overwrite versions of each other.

4.2.3 Object integrity and authenticity

We call the pair $\langle$*data key*, *data value*$\rangle$ an *object*. In our algorithms, the *data key*[2] is represented by a tuple $\langle ts, id, h \rangle_s$, where $\langle ts, id \rangle$ is the version, h is a cryptographic hash of the *data value* associated with this key, and s is a signature of $\langle ts, id, h \rangle$ (there is a slight difference in the protocol of §4.6, as will be discussed later). Having all this information on the data key allows us to validate the integrity and authenticity of the version (obtained through the **list** operation) before reading the data associated with it. Furthermore, if some version has a valid signature we call it *valid*. A data value is said to be *valid* if its hash matches the hash present in a valid key (this can only be verified after reading the value associated with the key). Consequently, an object is valid if both the version and the value are valid.

4.2.4 Erasure codes

Two of our protocols employ erasure codes [32] to decrease the storage overhead associated with full replication. This technique generates n different coded blocks, one for each base object, from which any $m < q$ base objects blocks can reconstruct the data. Concretely, in our protocols we use $m = f + 1$.

Notice that this formulation of coded storage can also be used to ensure confidentiality of the stored data, by combining the erasure code with a secret sharing scheme [25], in the same way it was done in DepSky [11].

4.3 Pseudo Code Notation and Auxiliary Functions

We use the '+' operator to represent the concatenation of strings and the '.' operator to access data key fields. We represent the parallelization of base object calls with the tag **concurrently**. Moreover, we assume the existence of a set of functions: (1) *H(v)* generates the cryptographic hash of v; (2) *encode(*v, n, m*)* encodes v into n blocks from which any m are sufficient to recover it; (3) *decode(*bks, n, m, h*)* recovers

[2]For the remaining of this paper we may refer to this only as *key*.

Algorithm 1: Auxiliary functions.

```
1  Function listQuorum() begin
2      L[0..n − 1] ←⊥;
3      concurrently for 0 ≤ i ≤ n − 1 do
4          L[i] ← list_i;
5      wait until |{i : L[i] ≠⊥}| ≥ q;
6      return L;
7  Function writeQuorum(data_key, value) begin
8      ACK[0..n − 1] ←⊥;
9      concurrently for 0 ≤ i ≤ n − 1 do
10         ACK[i] ← put(data_key, value[i])_i;
11     wait until |{i : ACK[i] = true}| ≥ q;
12 Function maxValidVersion(L) begin
13     return ⟨vr, h⟩_s ∈ ⋃_{i=0}^{n−1} L[i] : verify(s, K_u) ∧ ∄ ⟨vr′, h′⟩_{s′} ∈ ⋃_{i=0}^{n−1} L[i] : vr′ > vr ∧ verify(s′, K_u)) ;
```

a value v by decoding any subset of m out-of n blocks from the array bks if $H(v) = h$, returning $\perp$ otherwise; (4) *sign(info*, K_r*)* signs *info* with the private key K_r, returning the resulting signature s; (5) *verify(*s, K_u*)* verifies the authenticity of signature s using a public key K_u.

Besides these cryptographic and coding functions, our algorithms employ three auxiliary functions, described in Algorithm 1. The first function, *listQuorum* (Lines 1-6), is used to (concurrently) list the keys available in a quorum of KVSs. It returns an array L with the result of the **list** operation in at least q KVSs.

The *writeQuorum*(*data_key*, *value*) function (Lines 7-11) is used for clients to write data in a quorum of KVSs. The key *data_key* is equal in all base objects, but the value *value*$[i]$ may be different in each base object, to accommodate erasure-coded storage. When at least q successful **put** operations are performed, the loop is interrupted.

The last function, *maxValidVersion*(L) finds the maximum version number correctly signed on an array L containing up to n KVS' **list** results (possibly returned from *listQuorum* function), returning 0 (zero) if no valid version is found.

4.4 Two-Step Full Replication Construction

Our first Byzantine fault-tolerant MW-regular register construction employs full replication, storing thus the entire value written in each base object. The algorithm is optimally resilient as it employs a dissemination quorum system [28]. Algorithm 2 presents the write and read procedures for the construction.

Processes perform write operations using the procedure **FR-write** (Lines 1–7). The protocol starts by listing a quorum of base objects (Line 2). Then, it finds the maximum version available with a valid signature in the result using the function *maxValidVersion*(L) (Line 3), and creates the new data key by concatenating a new unique version, and the hash of the value to be written together with the signature of these fields (Lines 4–5). Lastly, it uses the *writeQuorum* function to write the data to the base objects (Lines 7).

The read operation is represented in the **FR-read** procedure (Lines 8–22). As in the write operation, it starts by listing a quorum of base objects. Then the reader enters in a loop until it reads a valid value (Line 10–21). First, it gets the maximum valid version listed (Line 11), and then it triggers n parallel threads to read that version from different KVSs. Next, it waits either for a valid value, which is immediately returned, or for a quorum of q responses (Line 19). The only way the loop terminates due to the second condition is if it is trying to read a version being written concurrently with the current operation, i.e., a version that is not yet available in a quorum. This is possible if the first q base objects to respond do not have the maximum version available yet. When this happens, the version is removed from the result of the **list** operation (Line 20), and another iteration of the outer loop is executed to fetch a smaller version. Notice that a version that belongs to a complete write can always be retrieved from the inner loop due to the existence of at least one correct base object in the intersection between Byzantine quorums.

Without concurrency, the protocol requires one round of **list** and one of **put** for writing, and one round of **list** and one of **get** for reading. In fact, it is impossible to implement a MW register with fewer object calls since for writing and reading we always need to use at least one round of **put** and **get** operations, respectively, and to find the maximum version available we can only use **list** or **get** to

Algorithm 2: Regular Byzantine Full Replication (FR) MW register ($n > 3f$) for client c.

```
1  Procedure FR-write(value) begin
2      L ← listQuorum();
3      max ← maxValidVersion(L);
4      new_key ← ⟨max.ts + 1, c, H(value)⟩;
5      data_key ← new_key + sign(new_key, K_r);
6      v[0..n − 1] ← value;
7      writeQuorum(data_key, v);
8  Procedure FR-read() begin
9      L ← listQuorum();
10     repeat
11         data_key ← maxValidVersion(L);
12         d[0..n − 1] ←⊥;
13         concurrently for 0 ≤ i ≤ n − 1 do
14             value_i ← get(data_key)_i;
15             if H(value_i) = data_key.hash then
16                 d[i] ← value_i;
17             else
18                 d[i] ← ERROR;
19         wait until (∃i : d[i] ≠⊥ ∧d[i] ≠ ERROR) ∨ (|{i : d[i] ≠⊥}| ≥ q);
20         ∀i ∈ {0, n − 1} : L[i] ← L[i] \ {data_key};
21     until ∃i : d[i] ≠⊥ ∧d[i] ≠ ERROR;
22     return d[i];
```

retrieve that information from the base objects.

4.5 Two-Step Erasure Code Construction

Differently from the protocol described in the previous section, which employs full replication with a storage requirement of $q \times S$ wherein S is the size of the object, in our second Byzantine fault-tolerant MW-regular register emulation we use storage-optimal erasure codes. Since the erasure code we use [32] generates n coded blocks, each with $\frac{1}{f+1}$ of the size of the data, the storage requirement is reduced to $q \times \frac{S}{f+1}$.

The main consequence of storing different blocks in different base objects for the same version, is that the number of base objects accessed in dissemination quorum systems is not enough to construct a wait-free Byzantine fault-tolerant MW-regular register. This happens because the intersection between dissemination quorums contains only $f+1$ base objects, meaning that when reading the version associated with the last complete write operation, the quorum accessed may contain only 1 valid response (f can be faulty). This is fine for full replication as a single updated and correct value is enough to complete a read operation. However, it may lead to a violation of the regular semantics when erasure codes are employed since we now need at least $f + 1$ encoded blocks to reconstruct the last written value.

To overcome this issue, we use Byzantine masking quorum systems [28], where the quorums intersect in at least $2f + 1$ base objects. Despite the increase in the number of base objects ($n > 4f$), the storage requirement is still significantly reduced when compared with the previous protocol. As an example, for $f = 1$, this protocol has a storage overhead of 100% (a quorum of four objects with coded blocks of half of the original data size) while in the previous protocol the overhead is 200% (a quorum of three objects with a full copy of the data on each of them).

Algorithm 3 presents this protocol. The **EC-write** procedure is similar to the write procedure of Algorithm 2. The only difference is the use of erasure codes to store the data. Instead of full replicating the data, it uses the *writeQuorum* function to spread the generated erasure-coded blocks through the base objects in such a way that each one of them will store a different block (Lines 6–7). Notice that the hash on the data key is generated over the full copy of data and not over each of the coded blocks.

The read procedure **EC-read** is also similar to the read protocol described in §4.4, but with two important differences. First, we remove from L the versions we consider impossible to read (Lines 10–11), i.e., versions that appear in less than $f+1$ responses. Second, instead of waiting for one valid response in the inner loop, we wait until we can reconstruct the data or for a quorum of responses. Again, the only way the loop terminates through the second condition is if we are trying to read a concurrent version. For reconstructing the original data, every time a new response arrives we try to decode the blocks and verify the integrity of the obtained data (Line 18). Notice that the integrity is verified inside the *decode* function. A version associated with a complete write can always be successfully decoded because any

Algorithm 3: Regular Byzantine Erasure-Coded (EC) MW register ($n > 4f$) for client c.

```
1  Procedure EC-write(value) begin
2      L ← listQuorum();
3      max ← maxValidVersion(L);
4      new_key ← ⟨max.ts + 1, c, H(value)⟩;
5      data_key ← new_key + sign(new_key, K_r);
6      v[0..n − 1] ← encode(value, n, f + 1);
7      writeQuorum(data_key, v);
8  Procedure EC-read() begin
9      L ← listQuorum();
10     foreach ver ∈ L : #_L(ver) < f + 1 do
11         ∀i ∈ {0, n − 1} : L[i] ← L[i] \ {ver};
12     repeat
13         data_key ← maxValidVersion(L);
14         data ←⊥;
15         concurrently for 0 ≤ i ≤ n − 1 do
16             d[i] ← get(data_key)_i;
17             if data =⊥ then
18                 data ← decode(d, n, f + 1, data_key.hash);
19         wait until data ≠⊥ ∨ |{i : d[i] ≠⊥}| ≥ q;
20         ∀i ∈ {0, n − 1} : L[i] ← L[i] \ {data_key};
21     until data ≠⊥ ∧ data ≠ ERROR;
22     return data;
```

Algorithm 4: Regular Byzantine Erasure-Coded (EC) MW register ($n > 4f$) for client c.

```
1  Procedure 3S-write(value) begin
2      L ← listQuorum();
3      max ← maxValidVersion(L);
4      data_key ← ⟨max.ts + 1, c⟩;
5      proof_info ←"PoW"+⟨max.ts + 1, c, H(value)⟩;
6      proof_key ← proof_info + sign(proof_info, K_r) ;
7      v[0..n − 1] ← encode(value, n, f + 1);
8      writeQuorum(data_key, v);
9      v[0..n − 1] ← ∅;
10     writeQuorum(proof_key, v);
11 Procedure 3S-read() begin
12     L ← listQuorum();
13     proof_key ← maxValidVersion(L);
14     data_key ← ⟨proof_key.ts, proof_key.id⟩;
15     data ←⊥;
16     concurrently for 0 ≤ i ≤ n − 1 do
17         d[i] ← get(data_key)_i;
18         if data =⊥ then
19             data ← decode(d, n, f + 1, data_key.hash);
20     wait until data ≠⊥;
21     return data;
```

accessed quorum will provide at least $f + 1$ valid blocks for decoding this version's value. As soon as the integrity is verified, the outer loop stops and the value is returned (Lines 21–22).

4.6 Three-Step Erasure Code Construction

Our last construction implements a Byzantine-resilient MW-regular register using erasure codes and dissemination quorums, being thus both storage-efficient and optimally-resilient. We achieve this by storing in each base object two objects per version instead of one. The first one, the *data object*, is used to store the encoded data blocks. The second one, the *proof object*, is an object with a zero-byte value used to prove that a given data object is already available in a quorum of base objects (similar to what is done in previous works [11, 18]). The key of the data object is composed only by the version, i.e., the tuple $\langle ts, id \rangle$. In turn, the key of the proof object is composed by the string $\langle \text{“PoW”}, ts, id, h \rangle_s$, in which h is the hash of the full copy of data and s is a signature of $\langle \text{“PoW”}, ts, id, h \rangle$.

Algorithm 4 presents the protocol. The write procedure, called **3S-write**, starts by listing the proof

objects from a quorum of base objects (Line 2). Then, it finds the maximum valid version between the proof objects. For simplicity, this algorithm uses the same function $maxValidVersion(L)$ as the previous protocols, but here we are only interested in proof objects. Next, it creates the new data key and the new proof key to be written (Lines 4–6). Then it writes the data object in a quorum (ensuring that different base objects will store different coded blocks) and, after that, it writes the proof object (Lines 7-10). This sequence of actions ensures that when a valid proof object is found in at least one base object, the corresponding data object is already available in a quorum of base objects.

The **3S-read** procedure is used for reading. The idea is to list proof objects from a quorum, find the maximum valid version among them, and read the data object associated with that proof object. Notice that to read the data we do not need to wait for a quorum of responses as it is enough to have $m = f + 1$ valid blocks to decode the value (Lines 18–19). This holds because, differently from the two previous algorithms, here we are sure that the data values with a version matching the maximum version found in valid proof objects is already stored in a quorum of base objects.

As explained before, this protocol works with only $3f + 1$ base objects. This is done without adding any extra call to the base objects in the read operation, which still needs only two rounds of accesses, one for **list** and one for **get**. However, for writing, one additional round of **put** is needed (to replicate the proof object). This trade-off is actually profitable in a cloud-of-clouds environment since the monetary costs of storing erasure-coded blocks in extra clouds is much larger than sending zero-byte objects to the clouds we use.

5 Protocols Extensions

This section presents a discussion of how the protocols presented in this paper can be modified to offer atomic semantics [26], and what are the possible solutions to garbage collect obsolete data versions.

5.1 Atomicity

There are many known techniques to transform regular registers in atomic ones. Most of them require servers running part of the protocol [14, 30], which is impossible to implement with our base objects. Fortunately, the simplest transformation can be used in data-centric algorithms. This technique consists in forcing readers to *write-back* the data they read to ensure this data will be available in a quorum when the read completes [10, 21, 29].

Our three read constructions could implement this technique by invoking *writeQuorum* to write the read value before returning it. However, writing back read values in our first two protocols may carry out performance issues as the stored data size might be non-negligible. In turn, employing the same write-back technique in our last protocol (Algorithm 4) does not have such overhead, as a reader would only need to write-back the small proof object (see §4.6). Hence, the performance effect of using this technique in the read procedure is independent of the size of the data being read.

A final concern about using write-backs to achieve atomicity is that we would have to assume that readers may only fail by crash, otherwise they may write bogus values in the base objects. In the regular constructions this is not required as we do not need to give write permissions to readers.

5.2 Garbage Collection

Existing solutions. Register emulations that employ versioning must use a garbage collection protocol to remove obsolete versions, otherwise an unbounded amount of storage is required. DepSky [11] provides a garbage collection protocol that is triggered periodically to remove older versions from the system. Although practical in many applications (e.g., cloud-backed file system [12]), this solution is vulnerable to the *garbage collection racing problem* [10,34]. This problem happens when a client is reading a version that had became obsolete due to a concurrent write, and removed by a concurrent execution of the garbage collection protocol, making it impossible for a reader to obtain the value associated with it.

To the best of our knowledge, there are only two works that solve this problem. The solution of [34] makes readers announce the version they are going to read, preventing the garbage collector from deleting it. Unfortunately, this solution cannot be directly applied in the data-centric model since it requires servers capable of running parts of the algorithm. Another solution was proposed in [10]. In this protocol each writer stores the value in a *temporary* key, which can be garbage collected by other writers, and also in an *eternal* key, that is never deleted. This approach allows readers to obtain the value from the eternal key when the temporary key is erased by concurrent writers. A solution like this can be applied

to our first protocol (see §4.4), which employs full replication. Yet, it does not work with erasure-coded data. This happens because the eternal key is overwritten whenever a write operation occurs, and since several writers can operate simultaneously, the eternal key in different base objects may end up with blocks belonging to different versions. Therefore, it might lead to the impossibility of getting $f+1$ blocks of the same version to reconstruct the original value.

Adapting the solutions to our protocols. All existing solutions for garbage collection can be adapted to the protocols discussed in §4. The approach of deleting obsolete versions asynchronously by a thread running in background can be naturally integrated to our protocols. This thread can be triggered by the clients at the end of the write operations, making each client responsible for removing its obsolete data.

Since we do not rely on server-side code for our protocols, devising a solution where readers announce the version they are about to read (by writing an object with that information to a quorum of base objects) would require substantial changes in our system model. More specifically, to ensure wait-freedom for read operations, only objects with versions lower than the ones announced can be garbage collected. This solution may not tolerate the crash of the readers – if a reader crashes without removing its announcement, larger versions than the one it announced will never be removed. It is possible to add an expiration time to the announcement to avoid this. Yet, this would still require changes in the system model to add synchrony assumptions for the expiration time to (eventually) hold, and not consider Byzantine readers (that could block garbage collection by announcing the intention to read all versions).

Using the eternal key approach together with erasure codes significantly increases the storage requirements of our algorithms. The idea is to make each writer not only to store the coded blocks into temporary keys, but also to replicate full copies of the original data in eternal keys. This approach may lead to a decrease in the write performance (related with an extra write of a full copy of the data per base object) and an increase of $n \times S$ in each protocol storage requirements.

Discussion. The three proposed solutions explore different points in the design space of data-centric storage protocols. In the first approach, we do not really solve the garbage collection racing problem. The second solution requires a stronger system model and additional base object accesses in the read operation. The third solution increases the storage requirements and reduces the write performance as writers have to write not only the coded blocks, but also full copies of the data.

We argue that most applications would prefer to have better performance and a reduced storage complexity, at the cost of eventually repeating failed reads. Therefore, we chose to support the asynchronous garbage collection triggered periodically (for example hourly, daily or even when a given number of versions has been written), as done in DepSky [11].

6 Evaluation

This section presents an evaluation of our three new protocols, comparing them with two previous constructions targeting the cloud-of-clouds model [10, 11].

6.1 Setup and Methodology

The evaluation was done using a machine in Lisbon and a set of real cloud services. This machine is a Dell Power Edge R410 equipped with two Intel Xeon E5520 (quad-core, HT, 2.27Ghz), and 32GB of RAM. This machine was running an Ubuntu Server Precise Pangolin operative system (12.04 LTS, 64-bits, kernel 3.5.0-23-generic), and Java 1.8.0_67 (64-bits).

Furthermore, we compare our protocols with the MW-regular register of [10], which we call ICS, and the SW-regular register of DepSky (the DepSky-CA algorithm) [11]. The protocols proposed in this paper were implemented in Java using the APIs provided by real storage clouds. We used the DepSky implementation available online [2]. However, since there is no available implementation of ICS, we implemented it using the same framework we used for our protocols. All the code used in our experiments is available on the web [4].

All experiments consider $f = 1$ and the presented results are an average of 1000 executions of the same operation, employing garbage collection after every 100 measurements. The storage clouds used were Amazon S3 [1], Google Storage [3], Microsoft Azure Storage [15], Rackspace Cloud Files [5], and Softlayer Cloud Storage [6]. ICS was configured to use the first three of them ($n = 3$); the Two-Step Full

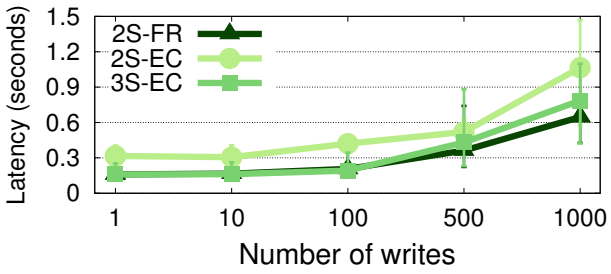
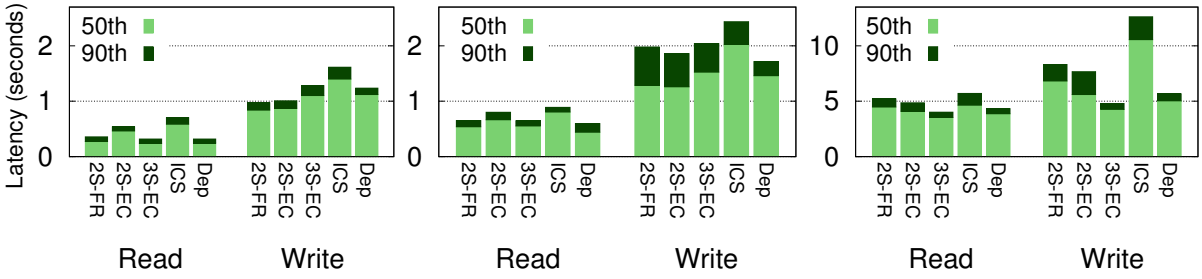


Figure 2: Average latency and std. deviation of *listQuorum* for different number of stored keys.



(a) 64kB (b) 1MB (c) 16MB

Figure 3: Median and 90-percentile latencies for read and write operations of register emulations.

Replication (2S-FR), Three-Step Erasure Codes (3S-EC) and DepSky protocols used the first four clouds mentioned ($n = 4$); and the Two-Step Erasure Codes (2S-EC) protocol used all of them ($n = 5$).

6.2 List Quorum Performance

One of the main differences between our protocols and the other MW-regular register of the literature designed for KVSs, namely ICS [10], is that in our algorithms the garbage collection is decoupled from the write operations. Since in ICS the garbage collection is included in the write procedure, the **list** operation invoked in its base objects always return a small number of keys. However, as in our protocols the garbage collection is executed in background, it is important to understand how the presence of obsolete keys (not garbage collected) in the KVSs affects the latency of listing the available keys. Notice this issue does not affect DepSky as it does not use the **list** operation [11].

Figure 2 shows the latency of executing the *listQuorum* function with different numbers of keys stored in the KVSs, for our three protocols (which consider different quorum sizes). As can be seen, 2S-EC presents the worst performance, indicating that listing bigger quorums is more costly. We can also observe that the performance degradation of the **list** operation when there are less than 100 obsolete versions is very small (specially for 2S-FR and 3S-EC). However, the latency is roughly 2× and 4× worse when listing 500 and 1000 versions, respectively. This suggests that triggering the garbage collection once every 100 write operations will avoid any significant performance degradation.

6.3 Read and Write Latency

Figure 3 shows the write and read latency of our protocols, ICS [10] and DepSky [11], considering different sizes of the stored data.

The results show that, when reading 64kB and 1MB, 2S-FR and 3S-EC present almost the same performance, while 2S-EC is slightly slower, due to the use of larger quorums. This means that reading only one data value with a full copy of the data is as fast as reading $f + 1$ blocks with half of the size of the original data. This is not the case for 16MB data. The results show it is faster to read $f + 1$ data blocks of 8MB in parallel from different clouds (2S-EC and 3S-EC) than reading a 16MB object from one cloud (2S-FR).

For writing 64kB objects 3S-EC is slower than 2S-FR and 2S-EC. This happens due to the latency of the third step of the protocol (write of the proof object). When writing 1MB objects, our protocols present roughly the same latency, being the 3S-EC protocol a little bit slower (also due to the write of the proof object). However, when clients write 16MB data objects, the additional latency associated with

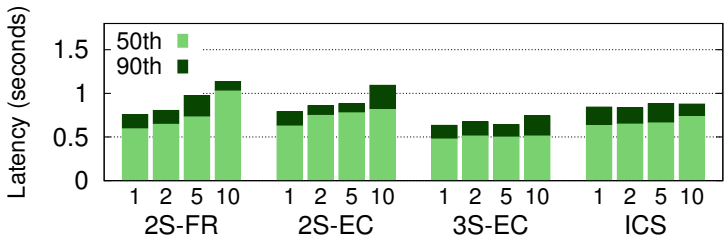


Figure 4: Median and 90-percentile read latencies in presence of contending writers.

this third step is negligible. Overall, these results can be explained by the fact that the proof object has zero bytes. Thus, 3S-EC protocol presents the best performance due to its use of dissemination quorums and erasure codes. For this data size, the 2S-FR protocol presents the worst performance of our protocols as it stores a full copy of the data in all clouds.

The key takeaway here is that our protocols present a performance comparable with DepSky [11] (Dep), *which does not support multiple writers*, and a performance up to 2× better than the *crash fault-tolerant* MW register presented in [10] (ICS). On the other hand, ICS presents the worst latency among the evaluated protocols. One of the main reasons for this to happen is because it does not use erasure codes. Furthermore, for reading, this protocol always waits for a majority of data responses, which makes it slower than, for example, the 2S-FR that only waits for one valid **get** response. In turn, for writing, ICS writes the full copy of the data twice on each KVS to deal with the garbage collection racing problem, removing also obsolete versions.

6.4 Read Under Write Contention

Figure 4 depicts the read latency of 1 MB objects in presence of multiple contending writers. This experiment does not consider DepSky as it only offers SW semantics.

The results show that both 2S-FR and 2S-EC read latencies are affected by the number of contending writers. This happens for two reasons: (1) under concurrent writes, these protocols typically try to read incomplete versions from the KVSs before finding a complete one (i.e., the loop on read protocols is executed more than once); (2) since we are not garbage collecting obsolete versions, more writers send more versions to the clouds, negatively influencing the *listQuorum* function latency. Since 3S-EC is not affected by the first factor, its read operation performs slightly better with contending writers.

ICS's read presents a constant performance with the increase of contending writers, however, 2S-FR and 2S-EC present competitive results and 3S-EC presents results always better than it, even without garbage collecting obsolete versions.

7 Conclusion

This paper considers the study of fundamental storage abstractions resilient to Byzantine faults in the data-centric model, with applications to cloud-of-clouds storage. In this context, we presented three new register emulations: (1) one that uses dissemination quorums and replicates full copies of the data across the clouds, (2) another that uses masking quorums and reduces the space complexity through the use of erasure codes, and (3) a third one that increases the number of accesses made to the clouds to use dissemination quorums together with erasure codes.

Our evaluation shows that the new protocols have similar or better performance and storage requirements than existing emulations that either support a single writer [10] or tolerate only crashes [11].

Acknowledgements. This work was supported by FCT through projects LaSIGE (UID/CEC/00408/2013) and IRCoC (PTDC/EEI-SCR/6970/2014), and by EU through the H2020 SUPERCLOUD project (643964).

References

[1] Amazon S3. `http://aws.amazon.com/s3/`.

[2] DepSky webpage. `http://cloud-of-clouds.github.io/depsky/`.

[3] Google storage. `https://developers.google.com/storage/`.

[4] MWMR-registers webpage. `https://github.com/cloud-of-clouds/mwmr-registers/`.

[5] Rackspace cloud files. `http://www.rackspace.co.uk/cloud/files`.

[6] Softlayer Cloud Storage. `http://www.softlayer.com/Cloud-storage/`.

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

[8] M. Aguilera, B. Englert, and E. Gafni. On using network attached disks as shared memory. In *Proc. of the PODC*, 2003.

[9] H. Attiya, A. Bar-Noy, and D. Dolev. Sharing memory robustly in message-passing systems. *Journal of the ACM*, 42(1), 1995.

[10] C. Basescu et al. Robust data sharing with key-value stores. In *Proc. of the DSN*, 2012.

[11] A. Bessani, M. Correia, B. Quaresma, F. Andre, and P. Sousa. DepSky: Dependable and secure storage in cloud-of-clouds. *ACM Transactions on Storage*, 9(4), 2013.

[12] A. Bessani, R. Mendes, T. Oliveira, N. Neves, M. Correia, M. Pasin, and P. Verissimo. SCFS: a shared cloud-backed file system. In *Proc. of the USENIX ATC*, 2014.

[13] C. Cachin, B. Junker, and A. Sorniotti. On limitations of using cloud storage for data replication. In *Proc. of the WRAITS*, 2012.

[14] C. Cachin and S. Tessaro. Optimal resilience for erasure-coded Byzantine distributed storage. In *Proc. of the DSN*, 2006.

[15] B. Calder et al. Windows Azure storage: a highly available cloud storage service with strong consistency. In *Proc. of the SOSP*, 2011.

[16] G. Chockler, D. Dobre, A. Shraer, and A. Spiegelman. Space bounds for reliable multi-writer data store: Inherent cost of read/write primitives. In *Proc. of the PODC*, 2016.

[17] G. Chockler and D. Malkhi. Active disk paxos with infinitely many processes. *Distributed Computing*, 18(1), 2005.

[18] D. Dobre, G. O. Karame, W. Li, M. Majuntke, N. Suri, and M. Vukolic. Powerstore: Proofs of writing for efficient and robust storage. In *Proc. of the CCS*, 2013.

[19] E. Gafni and L. Lamport. Disk paxos. *Distributed Computing*, 16(1), 2003.

[20] G. Gibson et al. A cost-effective, high-bandwidth storage architecture. In *Proc. of the ASPLOS*, 1998.

[21] G. Goodson, J. Wylie, G. Ganger, and M. Reiter. Efficient Byzantine-tolerant erasure-coded storage. In *Proc. of the DSN*, 2004.

[22] J. Hendricks, G. R. Ganger, and M. K. Reiter. Low-overhead Byzantine fault-tolerant storage. In *Proc. of the SOSP*, 2007.

[23] M. Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1), 1991.

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

[25] Hugo Krawczyk. Secret sharing made short. In *Proc. of the CRYPTO*, 1993.

[26] L. Lamport. On interprocess communication (part II). *Distributed Computing*, 1(1), 1986.

[27] R. Los, D. Shacklenford, and B. Sullivan. The notorious nine: Cloud Computing Top Threats in 2013. Technical report, Cloud Security Alliance (CSA), February 2013.

[28] D. Malkhi and M. Reiter. Byzantine quorum systems. *Distributed Computing*, 11(4), 1998.

[29] D. Malkhi and M.K. Reiter. Secure and scalable replication in Phalanx. In *Proc. of the SRDS*, 1998.

[30] J.P. Martin, L. Alvisi, and M. Dahlin. Minimal Byzantine storage. In *Proc. of the DISC*, 2002.

[31] T. Oliveira, R. Mendes, and A. Bessani. Exploring key-value stores in multi-writer Byzantine-resilient register emulations. Technical Report DI-FCUL-2016-02, ULisboa, 2016.

[32] M. Rabin. Efficient dispersal of information for security, load balancing, and fault tolerance. *Journal of the ACM*, 36(2), 1989.

[33] P. Verissimo, A. Bessani, and M. Pasin. The TClouds architecture: Open and resilient cloud-of-clouds computing. In *Proc. of the DCDV*, 2012.

[34] Y. Ye, L. Xiao, I-L. Yen, and F. Bastani. Secure, dependable, and high performance cloud storage. In *Proc. of the SRDS*, 2010.

A Appendix: Correctness

This section presents the correctness proofs of the protocols of Algorithms 2, 3 and 4. We start by proving that the auxiliary functions used by the protocols (presented in Algorithm 1) are wait-free.

Lemma 1. *Every correct process completes the execution of listQuorum and writeQuorum in finite time.*

Proof. Both algorithms are used by all protocols. This means that they use both dissemination and masking quorums. Since they send n requests, one for each base object, and at most f base objects are allowed to be faulty, a quorum of responses will always be received as $q \leq n - f$. Consequently, both algorithms return in finite time. □

A.1 Two-Step Algorithms Proof

In the following we prove the correctness of Algorithms 2 and 3, as they follow the same rationale, although employing different Byzantine quorum systems. For these proofs, we denote by L the output of the *listQuorum* executed in the beginning of **FR-read** and **EC-read** procedures. Moreover we define m as the number of required responses to obtain the requested value. Notice that $m = 1$ for full replication and $m = f + 1$ for erasure-coded data. The next lemmas state that L contains a version that respects MW-regular semantics.

Lemma 2. *A value associated with a complete write operation is always found in L, and can be retrieved from the base objects.*

Proof. Since we do not consider malicious writers, we know that clients only write valid objects. Furthermore, we know that when listing or reading a dissemination (resp. masking) Byzantine quorum of base objects we will also access $f + 1$ (resp. $2f + 1$) objects where the last complete write was executed, as by definition quorums intersect by this amount of objects. Using this fact, the lemma can then be reduced to prove that such intersection will contain m correct base objects, which will provide the last version written. This is indeed the case as $m = 1$ in full replication (intersection of $f + 1$ – at least one correct) and $m = f + 1$ with erasure coded data (intersection of $2f + 1$ – at least $f + 1$ correct). □

Lemma 3. *The maximum version found on L corresponds to the last complete write operation, or to a concurrent one.*

Proof. L only contains valid versions that were already stored by a writer, otherwise it would be impossible to find them. Since we do not consider malicious writers, we know they will follow the protocol, for example, by incrementing the maximum ts found and has not lied about his id when creating the pair $\langle ts + 1, c\rangle$. Therefore, each writer always writes a version larger than the maximum version found, respecting thus partial order.

According to Lemma 2, a version whose write is complete is always found in L. Consequently, we can claim that, without concurrency, the maximum version in L belongs to the last complete write operation. Furthermore, if there are any concurrent write operation being executed, it may appear as the maximum version found in L, as its version is surely greater then the version of the last complete write. □

These lemmas allow us to prove that both protocols (Algorithms 2 and 3) respect the specification of a multi-writer multi-reader regular register and are wait-free.

Theorem 1. *A FR-read (resp. EC-read) operation running concurrently with zero or more FR-write (resp. EC-write) operations will return the value associated with the last complete write or one of the values being written.*

Proof. Both read procedures start by calling *listQuorum*. By Lemma 3 we know that the maximum version found in L belongs to the last complete write operation or to a concurrent one. Independently of the case, the procedures try to read it. If the version belongs to a concurrent write it may not be retrieved. In this case the algorithms exclude it and fetch the new maximum valid version listed (see loop in Lines 10–21 and 12–21, in Algorithms 2 and 3, respectively). However, according to Lemma 2, if no concurrent version can be read, we know that the value associated with the last complete version is always retrieved. This proves that both protocols respect regular semantics. □

Theorem 2. *The FR-write, EC-write, FR-read and EC-read procedures satisfy wait-freedom.*

Proof. The **FR-write** and **EC-write** procedures, besides executing local computation steps, call the functions of Algorithm 1. Since these functions are wait-free (Lemma 1), the write protocols are also wait-free.

Both read operations start by calling the *listQuorum*, which is wait-free (Lemma 1). After that, the algorithms enter in a loop that only terminates after finding a valid value to return. By Lemma 2 we know that a value associated with a complete version is always found in L, and that this version can be retrieved. Then the number of iterations of this loop is bounded by the number of writes being executed concurrently that can be seen in L, but whose the value cannot be retrieved. Since after failing a read we try a smaller version, the algorithms will eventually try to fetch the value written in the last complete write. Consequently, the read procedures terminate in finite time. □

A.2 Three-Step Algorithm Proof

We now sketch the correctness proof of Algorithm 4. The complete proofs are very similar to the ones presented before for the Algorithms 2 and 3. The main difference here is the existence of the proof object used to prove that the data object associated with it is already stored in a dissemination quorum. The following lemmas state the properties of this object.

Lemma 4. *The value associated with every valid proof object in L can be retrieved from at least $f + 1$ base objects.*

Proof. Each valid proof object found in L was previously written by a correct writer. In turn, since we do not consider malicious writers, each writer only replicates the proof object after storing its associated data object in at least $q = \lceil \frac{n+f+1}{2} \rceil$ base objects. Therefore, we know that the data object associated with a valid proof object in L is available in at least $q - f \geq f + 1$ base objects. This means that there will be enough data objects to reconstruct the original value. □

Lemma 5. *The maximum valid version found among the proof objects observed in L corresponds to the last complete write, or to a concurrent one.*

Proof. This can be proved following the same rationale of Lemma 3. Writers are considered correct and therefore they calculate new versions correctly, i.e., they find the maximum valid version on a quorum of proof objects and increment it ensuring that each new version has a greater *version* number. Furthermore, an **3S-write** operation is considered complete only after it writes the proof object to a dissemination quorum. Since we know that the intersection of any two dissemination quorums contains at least $f + 1$ base objects, the proof object associated with the last complete write can always be found. This proves that, without concurrency, the maximum version found corresponds to the last complete write operation. If some concurrent writes are being executed, they may be seen as the maximum version because they surely have a greater version than the last complete write. □

Using these lemmas we are now able to prove that Algorithm 4 respects multi-writer multi-reader regular register semantics and wait-freedom.

Theorem 3. *A 3S-read operation running concurrently with zero or more 3S-write operations will return the value associated with the last complete write or one of the values being written.*

Proof. According to Lemma 4, the value associated with each valid proof object in L can always be read from the base objects. The protocol reads the value associated with the maximum valid version found in L, and we know by Lemma 5 that this version is associated either with the last complete write or to one of the values being written. Therefore, the value returned by **3S-read** belongs to the last complete write or to a concurrent one. □

Theorem 4. *The 3S-read and 3S-write procedures satisfy wait freedom.*

Proof. The write procedure invokes *listQuorum* once, to obtain L, and *writeQuorum* twice, one to write the data blocks and another to write the proof objects. According to Lemma 1, these two operations terminate in finite time, and thus **3S-write** always terminates as well.

The read procedure also satisfies wait freedom due to Lemma 4: a value associated with a valid proof object is always available for read. □