Consistency Anchor Formalization and Correctness Proofs

Alysson Bessani and Miguel Correia

DI–FCUL–TR–2014–02

DOI:10455/6899

(http://hdl.handle.net/10455/6899)

May 2014

Consistency Anchor Formalization and Correctness Proofs

Alysson Bessani[1] and Miguel Correia[2]
[1]Faculdade de Ciências/LaSIGE, [2]Instituto Superior Técnico/INESC-ID
Universidade de Lisboa, Portugal

Abstract

The consistency anchor technique allows one to increase the consistency provided by eventually consistent cloud storage services like Amazon S3. This technique has been used in the SCFS (Shared Cloud File System) cloud-backed file system for solving read-write conflicts and to provide strong consistency guarantees (i.e., equivalent to an atomic register) despite the weak consistency provided by the underlying cloud storage services. Here we present a formalization of such technique and prove its correctness.

1 Introduction

A key innovation of SCFS [1, 4] is the ability to provide strongly consistent storage over the eventually-consistent services offered by clouds [14]. Given the recent interest in strengthening eventual consistency in other areas, such as in geo-replication [12], we formalize the general technique here, decoupled from the file system design.

The technique is called consistency anchor, and considers two storage systems, one with limited capacity for maintaining metadata and another to save the data itself. We call the metadata store a *consistency anchor* (CA) and require it to enforce some strong consistency guarantee S, while the *storage service* (SS) may only offer eventual consistency. The objective is to provide a *composite storage* (CS) system that satisfies S, even if the data is kept in SS. The idea is illustrated in Figure 1.

This technical memo is organized in the following way. Section 2 presents our system model and underlying assumptions. Sections 3 and 4 describe the basic algorithm and its correctness proofs, respectively. Finally, Section 5 provides some extensions to address malicious writers and garbage collection.

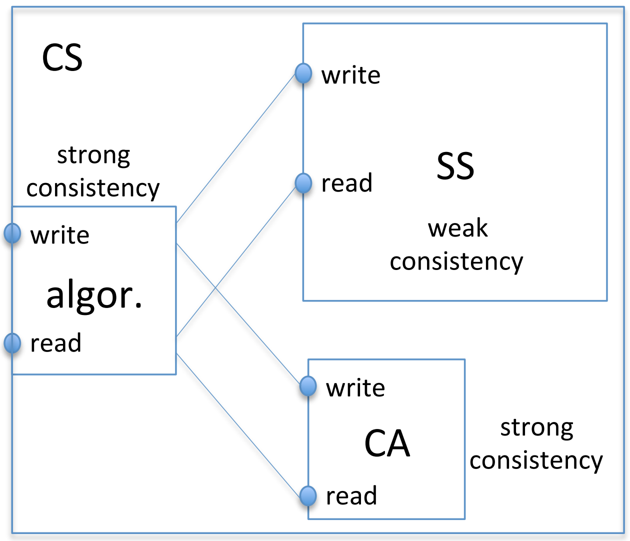

Figure 1: Composite service (CS) as a composition of a consistency anchor (CA) and a storage service (SS).

2 System Model and Assumptions

We assume an asynchronous system, with no time bounds on communication and processing, in which a unbounded number of crash-prone processes communicate through shared memory read/write registers.

For better matching the real world system, we group these registers in two storage services offering different guarantees:

- *Consistency Anchor* (CA): a strongly consistent data store with capacity to maintain m data items. Each of these data items behave like an atomic register [2, 10].
- *Storage Service* (SS): a weakly consistent data store with capacity to maintain an unbounded number of data items. Each of these data items satisfy only eventual consistency [14].

Our aim here is to use these two services to build a third one, the *Composite Service* (CS), that will provide the same consistency guarantee of the consistency anchor. The interface of all these services[1] is similar and contains only two opera-

[1]In SCFS [4], the CA is implemented by a coordination service [5, 6, 9], which provides strong consistency, while the SS is implemented by cloud storage services (e.g., S3), which normally satisfies only eventual consistency.

tions:

- write(id, v): writes a value v in the data unit id (i.e., equivalent to executing a $write(v)$ on a register id);
- read(id): reads the value stored in the data unit id (i.e., equivalent to executing a $read()$ on a register id).

We assume these operations satisfy wait-freedom [7] in both services (i.e., they never block). Moreover, the initial value of a data unit before its first write is assumed to be $null$.

Finally, we assume the existence of a collision-resistant hash function H.

2.1 Strong vs Weak Consistency

We can associate computational steps by calling and returning from these operations. An operation *precedes* another if it returns before the other operation is called. Two operations are *concurrent* if neither of them precedes the other.

Using the notion of precedence and concurrency, we can precisely define what we mean by strong and weak consistency in this paper.

Definition 1 (Strong Consistency) *Let $\mathcal{W}$ be the last write executed for a data unit that precedes at least one read $\mathcal{R}$ executed by any processes on this data unit. A storage service is said to be* strongly consistent *if the value written in $\mathcal{W}$ will be read in $\mathcal{R}$.*

Definition 2 (Weak Consistency) *Let $\mathcal{W}$ be the last write executed for a data unit that precedes an infinite number of non-concurrent reads $\mathcal{R}_1$, $\mathcal{R}_2$, ... executed by different processes on this data unit. A storage service is said to be* weakly consistent *if $\exists i \geq 1$ such that the value written in $\mathcal{W}$ is read in all $\mathcal{R}_j, j \geq i$.*

These definitions are very weak and aim to capture only the intuitive notions that in a strongly consistent data store a written value is immediately available to be read after its write completes and in a weakly consistent data store the written value will be eventually available for read. The semantics of safe, regular and atomic registers [10], as well as linearizability [8], satisfy strong consistency while models like eventual consistency [14] and eventual linearizability [13] satisfy weak consistency.

```
CS.write($id, v$):
w1: $h \leftarrow \mathsf{H}(v)$
w2: SS.write($id|h, v$)
w3: CA.write($id, h$)
```

```
CS.read($id$):
r1: $h \leftarrow$ CA.read($id$)
r2: if $h = null$ return $null$
r3: do
r4:     $v \leftarrow$ SS.read($id|h$)
r5: while $v = null$
r6: return $v$
```

Figure 2: Algorithm for increasing the consistency of the storage service (SS) using a consistency anchor (CA).

3 Algorithm

The basic idea of our construction CS is to first write the data in SS and then update the CA with some information about the current version (e.g., a hash) of the data item. In this way, when such version information is read, we know for sure that the associated data is already written in SS. In this way, the read procedure consists in reading the last written version information from the strongly consistent CA and keep trying to read the associated data, which was written and thus will be available eventually.

The algorithm for increasing the consistency guarantees is presented in Figure 2, and the insight is to anchor the consistency of the resulting storage service on the consistency offered by the CA. For writing, the client starts by calculating a collision-resistant hash of the data object (step w1), and then saves the data in the SS together with its identifier id concatenated with the hash (step w2). Finally, the data's identifier and hash are stored in the CA (step w3). One should notice that this mode of operation creates a new version of the data object in every write. Therefore, a garbage collection mechanism is needed to reclaim the storage space of no longer needed versions.

For reading, the client has to obtain the current hash of the data from CA (step r1), and then needs to keep on fetching the data object from the SS until a copy is available (steps r3-r5). The loop is necessary due to the weak consistency of the SS – after a write completes, the new hash can be immediately acquired from the CA, but the data is only eventually available in the SS.

4 Correctness Proofs

In this section we first prove the main property of our consistency anchor scheme: that the composite storage will provide strong consistency (Definition 1) as long as the consistency anchor satisfies strong consistency. Later on, we prove that the specific case for SCFS, considering that the consistency anchor satisfies the semantics of an atomic register.

In all proofs in this section we consider a single data item id for CS. This is not a problem whatsoever since read and write operations affect a single data unit (i.e., the data units are completely separated objects).

4.1 General Proof

Theorem 1 (General Safety) *If CA satisfies strong consistency and SS satisfies weak consistency, then CS satisfies strong consistency.*

Proof: According to the definition of strong consistency (Definition 1), we have to show that a read operation $\mathcal{R}$ on CS returns the value written on the last write operation $\mathcal{W}$ completed on CS. According to the algorithm of Figure 2, a CS.write comprises a write on SS and on CA, i.e., $\mathcal{W} \rightarrow \mathcal{W}^{SS}, \mathcal{W}^{CA}$. On the other hand, a CS.read comprises a read in CA and one or more reads in SS, i.e., $\mathcal{R} \rightarrow \mathcal{R}^{CA}, \mathcal{R}_1^{SS} \ldots \mathcal{R}_n^{SS}$.

Let the value written in $\mathcal{W}$ for id be v. It means that $\mathcal{W}^{CA}$ writes $\mathsf{H}(v)$ for id and $\mathcal{W}^{SS}$ writes v for $id|\mathsf{H}(v)$. Furthermore, let the CS.read $\mathcal{R}$ return $v' \neq null$ for a data unit id, meaning that $\mathcal{R}^{SS}$ returned v' for a (internal) data unit k s.t. the id k was the value read in $\mathcal{R}^{CA}$ for a data unit id.

Since (1) $\mathcal{W}$ precedes $\mathcal{R}$, this implies that $\mathcal{W}^{CA}$ precedes $\mathcal{R}^{CA}$ and (2) CA satisfies strong consistency, we have that the value read in $\mathcal{R}^{CA}$ is $k = \mathsf{H}(v)$ (the value written in $\mathcal{W}^{CA}$). Consequently, $\mathcal{R}^{SS}$ will try to read the value associated with id and $k = \mathsf{H}(v)$, and the only written value for this data unit is $v' = v$. ■

Theorem 2 (General Liveness) *If both CA and SS satisfies wait freedom, then CS also satisfies wait freedom.*

Proof: A CS.write operation is clearly wait-free since both SS.write and CA.write are wait-free. A CS.read, on the other hand, requires that both CA.read and SS.read operations to finish, and also that the condition on the while of r5 to be false. This condition will be false only if SS.read($id|h$) returns some written value.

By the algorithm of Figure 2, we have that if CA.read(id) returns h, then CA. write(id,h) was invoked, which implies that $\mathcal{W}^{SS}$ =SS.write($id|h$,v) such that $h =$

$\mathsf{H}(v)$ completed. Consequently, $\mathcal{W}^{SS}$ precedes any SS.read($id|h$) executed in the while loop of steps r3-r5.

Given that no other write besides $\mathcal{W}^{SS}$ for $id|h$ is executed in SS, and that it satisfies our definition of weak consistency (Definition 2), eventually one of the reads for $id|h$ will read v s.t. $h = \mathsf{H}(v)$, and CS.read will complete. ■

4.2 SCFS' Consistency Anchor Proof

The SCFS implementation of the consistency anchor considers the use of DepSpace [5] or Zookeeper [9] as a coordination service and implements the composite storage read and write operations. DepSpace – which is the system that fully satisfies the requirements of SCFS and that was used to evaluate it [4] – satisfies the properties of an atomic register. Furthermore, the storage service used in SCFS (DepSky [3] or Amazon S3) provides only eventual consistency, which satisfies our definition of weakly consistent storage service. In this section we prove that in this practical setting, the composite storage also satisfies the semantics of an atomic register (Definition 3).

Definition 3 (Atomic register [2]) *An atomic register providing read and write operations must satisfy the following two properties:*

1. *Every read operation returns either the value written by the most recent preceding write operation (the initial value if there is no such write) or a value written by a write concurrent with this read operation;*
2. *If a read operation $\mathcal{R}_1$ reads a value from a write operation $\mathcal{W}_1$, and a read operation $\mathcal{R}_2$ reads a value from a write operation $\mathcal{W}_2$, with $\mathcal{R}_1$ preceding $\mathcal{R}_2$, then $\mathcal{W}_2$ does not precede $\mathcal{W}_1$.*

We prove the two properties of Definition 3 for SCFS in two separate lemmata.

Lemma 1 (SCFS Atomicity 1) *If all CA data unit satisfies the semantics of an atomic register (Definition 3) and SS satisfies weak consistency, every CS.read operation returns either the value written by the most recent preceding CS.write operation (the initial value if there is no such write) or a value written by a CS.write concurrent with this CS.read operation.*

Proof: Let $\mathcal{R}$ be a read operation returning v, $\mathcal{W}_l$ be last write (of v_l) preceding $\mathcal{R}$, and $\mathcal{W}_c$ a possible write (of v_c) concurrent to $\mathcal{R}$.

According to the algorithm of Figure 2, a CS.write comprises a write on SS and on CA, i.e., $\mathcal{W}_l \rightarrow \mathcal{W}_l^{SS}, \mathcal{W}_l^{CA}$ and $\mathcal{W}_c \rightarrow \mathcal{W}_c^{SS}, \mathcal{W}_c^{CA}$. On the other hand, a

CS.read comprises a read in CA and one or more reads in SS, i.e., $\mathcal{R} \rightarrow \mathcal{R}^{CA}, \mathcal{R}_1^{SS}$ $...\mathcal{R}_n^{SS}$.

Theorem 1 already proves the case in which $\mathcal{W}_c$ does not exists. So it remains to prove that if CS.read $\mathcal{R}$ returns $v \neq v_l \neq null$, then there exists a concurrent write $\mathcal{W}_c$ and $v = v_c$.

If $\mathcal{R}$ returns $v \neq null$ for a data unit id, meaning that $\mathcal{R}^{SS}$ returned v for a (internal) data unit $id|\mathsf{H}(v)$ s.t. this id was the value read in $\mathcal{R}^{CA}$ for a data unit id. Considering that CA satisfies Definition 3 and assuming $v_c \neq v_l$ and that there are no other concurrent writes besides $\mathcal{W}_c$, $\mathcal{R}^{CA}$ can only return $\mathsf{H}(v)$, with $v \neq v_l$ if $v = v_c$. ■

Lemma 2 (SCFS Atomicity 2) *If all CA data unit satisfies the semantics of an atomic register (Definition 3) and SS satisfies weak consistency, then if a CS.read operation $\mathcal{R}_1$ reads a value from a CS.write operation $\mathcal{W}_1$, and a CS.read operation $\mathcal{R}_2$ reads a value from a CS.write operation $\mathcal{W}_2$, with $\mathcal{R}_1$ preceding $\mathcal{R}_2$, then $\mathcal{W}_2$ does not precede $\mathcal{W}_1$.*

Proof: Let v_1 be the value written in $\mathcal{W}_1$ and read $\mathcal{R}_1$ and v_2 be the value written in $\mathcal{W}_2$ and read $\mathcal{R}_2$.

Let's assume for the sake of contradiction that $\mathcal{R}_1$ precedes $\mathcal{R}_2$ and $\mathcal{W}_2$ precedes $\mathcal{W}_1$. If $\mathcal{R}_1$ precedes $\mathcal{R}_2$ then $\mathcal{R}_1^{CA}$ (reading $\mathsf{H}(v_1)$) precedes $\mathcal{R}_2^{CA}$ (reading $\mathsf{H}(v_2)$). If $\mathcal{W}_2$ precedes $\mathcal{W}_1$ then $\mathcal{W}_2^{CA}$ (writing $\mathsf{H}(v_2)$) precedes $\mathcal{W}_1^{CA}$ (writing $\mathsf{H}(v_1)$). This violates the Property 2 of the atomic register (Definition 3), which is satisfied by the CA, and consequently $\mathcal{W}_2$ can not precede $\mathcal{W}_1$. ■

Lemmata 1 and 2 together prove that if CA satisfies the semantics of an atomic register, then CS provides this same consistency guarantee.

5 Extensions

In this section we present two extensions to the protocol of Figure 2 for implementing garbage collection and tolerating Byzantine processes.

5.1 Garbage Collection

Our algorithm creates a new data unit in SS for each write in the composite storage, requiring thus an amount of storage space proportional to the number of writes executed in the system. In practice this creates the necessity for running a garbage collection procedure to delete old versions of each data unit.

A simple protocol for implementing this procedure would be to list all objects in SS and remove the ones that are not registered in the CA. A problem with this

```
CS.write($id, v$):
w1: $h \leftarrow \mathsf{H}(v)$
w2: $ts \leftarrow \mathcal{T}$
w3: SS.write($id|ts|h, v$)
w4: CA.write($id, ts|h$)
```

```
CS.read($id$):
r1: $ts|h \leftarrow$ CA.read($id$)
r2: if $h = null$ return $null$
r3: $t \leftarrow \mathcal{T}$
r4: do
r5:     $v \leftarrow$ SS.read($id|h$)
r6: while $v = null$ and $\mathcal{T} - t < \Delta$
r7: return $(h = \mathsf{H}(v))?v : null$
```

Figure 3: Algorithm for increasing the consistency of the storage service (SS) using a consistency anchor (CA) with Byzantine writers and considering the existence of a garbage collection to clean old versions in SS.

procedure is that it might cause the deletion of a value being written, i.e., a value that was written in the SS (w2) but still not in the CA (w3). This problem can be solved assuming approximately synchronized clocks between processes and adding the write timestamp ts on the name of the data unit being written in the CA (using $id|ts|h$ instead of $id|h$). This timestamp can be obtained by reading the local clock $\mathcal{T}$ in the writer. Figure 3 presents the write and read algorithm with this modification.

In this way, when a list operation is executed for garbage collecting old versions, it is possible to order the versions according to this timestamp and preserve the most recent(s), deleting only the ones created long time ago.

5.2 Dealing with Byzantine Writers

The consistency anchor algorithm tolerates crash faults on writers and readers. In case a writer crashes, the worst possible behavior is that the value will be written to SS but the version information will not be updated in CA. The only bad consequence of this is the additional storage space used in SS, which will be proportional to the number of faults (in the worst case).

However, the algorithm will not work if we consider the possibility of malicious (Byzantine) writers. Such dishonest writers can break the protocol through the execution of *invalid writes* that will brake the safety or liveness of reads. More specifically, a malicious writer can affect the protocol in two ways. First, it can execute CA.write(id, h) without executing SS.write($id|h, v$). Second, it can execute CA.write(id, h) and SS.write($id|h, v$) with $h \neq \mathsf{H}(v)$. In the first case all reads for id preceding this write will be stuck in the while loop forever, while in the second the returned value will not be the one of the last complete write.[2]

To deal with these problems we need additional assumptions and some modifications on the protocol. First, we need to assume that a written value will be returned by a read after at most Δ time units. This bound can be obtained experimentally with a confidence as high as needed or be based on internal parameters of SS configuration. Furthermore, we need to change the semantics of CS to return *null* in a read when the last executed write was invalid.

The modified CS.read algorithm is presented in Figure 3. In this version, the reader marks the instant it started reading SS (r3 - $\mathcal{T}$ represents the current local time in the reader) and the while loop procedure exits either after reading a valid read value or Δ time units (r6), thus preventing missing writes to SS from blocking the operation. Furthermore, the new algorithm considers that the value written in SS might not match the hash written in CA. In this case, the read operation also returns *null* (r7).

Acknowledgements

We would like to thank Nuno Neves for the comments to improve this paper.

References

[1] The SCFS webpage. `http://code.google.com/p/depsky/wiki/SCFS`.

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

[3] 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.

[4] 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 2014 USENIX Annual Technical Conference*, 2014.

[2]Notice that it is difficult to define what is a complete write since an invalid write may or may not be considered as the last complete write (see [11] for a discussion on this).

[5] A. N. Bessani, E. P. Alchieri, M. Correia, and J. S. Fraga. DepSpace: a Byzantine fault-tolerant coordination service. In *Proc. of the 3rd ACM European Systems Conference*, pages 163–176, Apr. 2008.

[6] M. Burrows. The Chubby Lock Service. In *Proceedings of 7th Symposium on Operating Systems Design and Implementation*, Nov. 2006.

[7] M. Herlihy. Wait-free synchronization. *ACM Transactions on Programing Languages and Systems*, 13(1):124–149, Jan. 1991.

[8] M. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programing Languages and Systems*, 12(3):463–492, July 1990.

[9] P. Hunt, M. Konar, F. Junqueira, and B. Reed. Zookeeper: Wait-free Coordination for Internet-scale Services. In *Proc. of the USENIX Annual Technical Conference*, pages 145–158, June 2010.

[10] L. Lamport. On interprocess communication (part II). *Distributed Computing*, 1(1):203–213, Jan. 1986.

[11] B. Liskov and R. Rodrigues. Tolerating Byzantine faulty clients in a quorum system. In *Proc. of the 26th IEEE International Conference on Distributed Computing Systems*, June 2006.

[12] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen. Don't settle for eventual: Scalable causal consistency for wide-area storage with COPS. In *Proc. of the 23rd ACM Symposium on Operating Systems Principles*, Cascais, Portugal, Oct. 2011.

[13] M. Serafini, D. Dobre, M. Majuntke, P. Bokor, and N. Suri. Eventually linearizable shared objects. In *Proc. of the 29th ACM Symposium on Principles of Distributed Computing*, 2010.

[14] W. Vogels. Eventually consistent. *Communications of the ACM*, 52(1):40–44, 2009.