Implementing a Robust Multi-Cloud Store: Capabilities and Limitations

Position Paper

Gregory Chockler
Royal Holloway, University of London
Gregory.Chockler@rhul.ac.uk

Dan Dobre
NEC Labs Europe
dan.dobre@neclab.eu

Alexander Shraer
Google, Inc.
shralex@google.com

Abstract

Cloud-based storage services have established themselves as a paradigm of choice for supporting bulk storage needs of modern networked services and applications. Although individual storage service providers can be trusted to do their best to reliably store the user data, exclusive reliance on any single provider or storage service leaves the users inherently at risk of being locked out of their data due to outages, connectivity problems, and unforeseen alterations of the service contracts. An emerging multi-cloud storage paradigm addresses these concerns by replicating data across multiple cloud storage services, potentially operated by distinct providers. In this position paper, we report on our recent study seeking to shed light on fundamental capabilities and limitations of building robust multi-cloud storage services out of a collection of fault-prone cloud stores. We outline the initial results we have obtained so far, and propose directions for future exploration.

1 Introduction

A rapidly growing number of Internet companies offer Storage-As-A-Service to their customers. These include big corporations such as Amazon, Google, Microsoft, Apple, EMC, HP, IBM, AT&T, as well as numerous smaller providers such as Dropbox, Box, Rackspace, Nirvanix and many others. The popularity of cloud storage stems from its flexible deployment, convenient pay-per-use model, and little (if any) administrative overhead. This is especially attractive for smaller businesses, who cannot afford the costs of deploying and administering enterprise-scale storage infrastructure on their premises, and would rather outsource this task to an external entity.

Although cloud storage providers make tremendous investments into ensuring reliability and security of the service they offer, most of them have suffered from well-publicized outages where the integrity and/or availability of data have been compromised for prolonged periods of time [27, 12, 23]. In addition, even in the absence of outages, the customers can still lose access to their data due to connectivity problems, or unexpected alterations in the service contract. In fact, the problem of *data lock-in* [6] in which the customers become critically dependent on a specific cloud provider for all their data storage needs has long been considered a major roadblock to a wider adoption of cloud storage for sensitive data such as banking, medical, or critical infrastructure domains.

To address these concerns, *multi-cloud* storage systems whereupon the data is replicated across multiple cloud storage services (potentially operated by distinct providers) have recently become a hot topic in the systems community [29, 1, 38, 10, 9]. Note that since in this setup, the individual stores can be hosted by different cloud providers, the service implementation becomes the sole responsibility of a *user-side* proxy (such as a client library, or a middle tier) whose goal is to mediate between the users and the individual cloud stores so as to ensure data availability in the face of asynchrony, concurrency, and failures of both individual services and users.

Cloud Store	API	Comments
Yahoo! PNUTS [15]	Test-and-Set-Write(required version)	Write if and only if the present version is equal to required version
Amazon SimpleDB [34]	PutAttributes	Update iff specified attribute/value match the existing one
Amazon DynamoDB [17]	PutItem	Replace an existing item if it has certain attribute values
MongoDB [28]	Update if current	
Google Cloud Storage [35]	Conditional Write based on monotonically increasing versions	
Windows Azure Storage [36]	Update/Delete with if-match	
Riak [31]	Update/Delete with if-match	
Yahoo! Zookeeper [21]	setData(String path, byte[] data, int version)	
IBM Spinnaker [30]	TAS, TAS-multi-put	Inserts values into specified columns if a list of "column=timestamp" tests succeed

Table 1: Conditional Update Primitives Exposed by the Existing Cloud Data Stores

Although a significant progress has so far been made in building practical multi-cloud storage systems [1, 10, 9], as of today, little is known about their fundamental capabilities and limitations. The primary challenge lies in a wide variety of the storage interfaces and consistency semantics offered by different cloud providers to their external users. For example, whereas Amazon S3 [32] supports a simple read/write interface, other storage services also expose a selection of more advanced transactional primitives, such as conditional writes (see Table 1).

In this position paper, we report on our recent study [13] seeking to shed light on fundamental capabilities and limitations of building robust multi-cloud storage services out of a collection of fault-prone cloud stores. We outline the initial results we have obtained so far (see Section 3), and propose directions for future exploration (see Section 4).

2 System Model

For the sake of formal analysis, we model a multi-cloud storage system as an asynchronous fault-prone shared memory system of Jayanti et al. [22]. Specifically, we abstract individual cloud stores as fault-prone shared objects (to which we refer as *base* objects), cloud users as processes accessing these objects, and a reliable multi-cloud storage service as a *fault-tolerant object emulation* consisting of the process algorithms interacting with the base objects.

3 Initial Results

In this section, we outline the results of our recent study [13] that explored the space and time complexity of building multi-cloud storage services abstracted as fault-tolerant object emulations as a function of the following parameters:

1. emulated object *type*,
2. *the number of supported processes*, and
3. the *safety properties* offered by the individual base objects being used.

3.1 Space Requirements for Supporting Multiple Clients

Our first result establishes a lower bound on the space required to emulate a reliable wait-free multi-writer/single-reader register supporting *safe* consistency [25, 24, 33], which is one of the most basic guarantees one could expect from a storage service. We assume underlying storage services supporting read/write and list primitives[1], which we model as multi-writer/multi-reader (MWMR) *atomic snapshot* objects [2, 4, 5]. In [13], we prove the following

Theorem 3.1 *Let A be an implementation of a wait-free multi-writer/1-reader safe register out of a collection of n base wait-free atomic multi-writer/multi-reader snapshot objects [2, 4, 5] each of which storing vectors consisting of $m > 0$ entries. Suppose that at most $t > 0$ base objects can crash, and $n > t$. Then, there exists a failure-free execution α of A consisting of a sequence $W_1, W_2, \ldots, W_k$ of write operations W_i invoked by processes $P_1, \ldots, P_n$ respectively such that*

1. *W_i is invoked after W_j returns for all $i > j$,*
2. *$P_i \neq P_j$ for all $i \neq j$,*
3. *$k = \lfloor (nm - t - 1)/t \rfloor$, and*
4. *There does not exist an extension α' of α in which a new process $P \notin \{P_1, \ldots, P_k\}$ invokes a write operation after W_k returns.*

That is, irrespective of the number of failures being tolerated by the emulation, the *maximum* number of distinct processes (i.e., k) that could *ever* write the emulated register (either concurrently or not) is bounded by a quantity, which is *linear* in the number of entries supported by the underlying snapshot objects.

In other words, the space overhead associated with storing each individual data item (such as e.g., a single key/value) is proportional to the maximum number of clients that could ever update this item, and in particular, cannot be optimized under the assumption of bounded maximum concurrency (i.e., point contention [3, 8]). In practice, this means that architects of the multi-cloud storage services should avoid using speculative techniques that stipulate bounded peak load (such as e.g., memory overcommit [37, 11, 20]), but rather focus on limiting the *total* number of writers (e.g., through deploying proxies, or access control mechanisms) as the means of optimizing the space usage.

In particular, our result explains the space overheads incurred by practical implementations of reliable multi-cloud data stores recently published by Basescu et. al [9] and Ye et al. [38]. Their algorithms incur worst-case space complexity proportional to the number of writers (regardless of concurrency), which matches our lower bound.

An important theoretical consequence of our space bound is that it shows that shared memory algorithms based on reliable multi-writer registers are subject to a linear (in the number of writers) space blow-up when transformed to a fault-prone shared memory. This means that failures cause multi-writer registers to "lose" their ability to support multiple writers using constant space, rendering them equivalent to single-writer registers. Note that since our proof assumes base objects supporting multi-writer atomic snapshot, which is in a sense, the strongest possible read/write memory abstraction, our result also carries over to other multi-writer object emulations in this model, such as e.g., consensus, and multi-writer snapshots.

3.2 Space-Efficient Emulations Using Conditional Writes

We next turn to emulating reliable registers over storage services supporting transactional update primitives. First, it is well known that a constant number of *read-modify-write* objects is indeed sufficient to reliably emulate a multi-writer atomic register [7, 18]. This implies that strengthening the underlying object semantics is indeed essential to avoid linear dependency on the number of writers implied by the lower bound in Section 3.1.

However, the read-modify-write objects employed by the existing implementations are too specialized to be exposed by the commodity cloud storage interfaces. For example, the write operation implementation of the ABD protocol [7, 18] relies on a read-modify-write primitive which performs the following two steps as a single atomic

[1] The list primitive is supported by Amazon S3

operation: (1) test the timestamp associated with the value supplied in the write operation against the one associated with the stored value, and (2) update the stored value with the supplied one only if the supplied timestamp is strictly greater than the stored one.

Instead, the cloud storage providers typically expose *general purpose* read-modify-write primitives, such as those summarized in Table 1. These primitives are variants of *conditional writes*, and therefore, essentially equivalent to *compare-and-swap (CAS)*. We therefore, adopt a shared memory system with fault-prone *CAS objects* as our model of cloud storage services supporting conditional writes, and study reliable object emulations in this environment. In particular, in [13], we show that there exist reliable *constant* space implementations of the following two objects:

- **multi-writer atomic register**, which requires the underlying clouds to only support a *single CAS* object per stored value, is *adaptive to point contention* (i.e., the maximum number of clients executing concurrently with the operation), and tolerates up to a minority of base object failures, and
- **Ranked Register (RR)** [14] using a single fault-prone *CAS* object. A collection of such Ranked Registers can be used to construct a reliable Ranked Register, from which agreement is built [14]. Our construction thus obtains a multi-cloud state machine replication service capable of supporting infinitely many clients using constant space.

4 Open Questions

We believe that our work opens several interesting new avenues for future research. Below, we enumerate several open questions naturally arising from, or extending our initial results above:

- The step complexity of our atomic register implementation is quadratic in point contention. Is this optimal for emulating atomic registers? Interestingly, if this question can be answered in the affirmative, and since an atomic object is known to be implementable in constant space from generic read-modify-write primitives (ABD [7, 18]), this would imply that there is a time complexity separation between *CAS* and generic read-modify-write primitive, which have been previously thought to be equivalent in all other respects (such as e.g., the power to implement consensus [19]).
- If the step complexity of our atomic register implementation is optimal, is the same true for weaker consistency variants (such as regular, safe, or timeline)?
- Is it possible to improve the running time of the atomic register implementation under contention by leveraging timeliness assumption (e.g., loosely synchronised clocks), or randomization?
- Are there any benefits with respect to space or time complexity for mixing replicas exposing read/write primitives with those exposing conditional write primitives?
- Our space bound is shown for safe registers which, despite being weaker than atomic and regular registers, still require that a read returns the value of a most recently completed write (if it does not overlap the write). It remains open whether a similar bound also applies to weaker forms of consistency, such as e.g., sequential [26], timeline [15], or causal consistency.
- The proof of our space bound in [13] crucially depends on a capability to terminate each high-level write operation without waiting for the responses from the base object writes that were invoked in the course of the prior write invocations but have been left "hung" (due to a possibility of the object failure). Note that this capability is no longer available if each WRITE is guaranteed to terminate, or in other words, all writers are correct. Since the writer reliability can be enforced in many practical settings, it will be interesting to see whether a constant memory algorithm can be constructed under the assumption of reliable writers, or the space bound can be further strengthened to also apply in this case.
- We conjecture that there is a simple algorithm implementing a $k * m$-writer/multi-reader atomic register out of $n > (k+1)t$ atomic snapshot objects of length m. This algorithm will use each "row" of n snapshot slots to support k writers using an algorithm similar to ABD. Thus, the number of supported writers can be bounded from below by $(\lceil (n-2t)/t \rceil)m$.

- We further conjecture that the above bound is tight since it is possible to choose $(\lceil (n-2t)/t \rceil)$ subsets S_i out of $n > t$ snapshot objects each of which consisting of t objects so that any implementation will have a run where each high-level write invocation invocation terminates while leaving base object write invocations pending on all objects in one of the sets S_i.
- Our initial results assume a benign failure model for both base objects and clients. An interesting research direction will be to strengthen the failure model to include Byzantine failures, and study the consistency and complexity tradeoffs involved in implementing a reliable multi-cloud store in this environment.
- Finally, it will be interesting to explore performance tradeoffs involved in implementing a reliable multi-cloud key-value store out of commodity data stores supporting conditional write primitives of various types. Possible questions to study in this context include the performance impact of storage interfaces supported by different vendors, and the implications of the write contention as found in the commonly used cloud store benchmarks, such as YCSB [16]. A natural next step will be to leverage the reliable multi-cloud read/write primitives for implementing multi-object transactions.

References

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

[2] Yehuda Afek, Hagit Attiya, Danny Dolev, Eli Gafni, Michael Merritt, and Nir Shavit. Atomic snapshots of shared memory. *J. ACM*, 40(4):873–890, 1993.

[3] Yehuda Afek, Hagit Attiya, Arie Fouren, Gideon Stupp, and Dan Touitou. Long-lived renaming made adaptive. In *Proceedings of the eighteenth annual ACM symposium on Principles of distributed computing*, PODC '99, pages 91–103, New York, NY, USA, 1999. ACM.

[4] James H. Anderson. Composite registers. *Distributed Computing*, 6(3):141–154, 1993.

[5] James H. Anderson. Multi-writer composite registers. *Distributed Computing*, 7(4):175–195, 1994.

[6] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. A view of cloud computing. *Commun. ACM*, 53(4):50–58, April 2010.

[7] Hagit Attiya, Amotz Bar-Noy, and Danny Dolev. Sharing memory robustly in message-passing systems. *J. ACM*, 42(1):124–142, January 1995.

[8] Hagit Attiya and Arie Fouren. Algorithms adapting to point contention. *J. ACM*, 50(4):444–468, July 2003.

[9] Cristina Basescu, Christian Cachin, Ittay Eyal, Robert Haas, Alessandro Sorniotti, Marko Vukolic, and Ido Zachevsky. Robust data sharing with key-value stores. In *DSN*, pages 1–12, 2012.

[10] Alysson Bessani, Miguel Correia, Bruno Quaresma, Fernando André, and Paulo Sousa. Depsky: Dependable and secure storage in a cloud-of-clouds. In *European Conference on Computer Systems (EuroSys)*, 2011.

[11] David Breitgand and Amir Epstein. Improving consolidation of virtual machines with risk-aware bandwidth oversubscription in compute clouds. In *INFOCOM*, pages 2861–2865, 2012.

[12] Rory Cellan-Jones. The Sidekick Cloud Disaster. `http://www.bbc.co.uk/blogs/technology/2009/10/the_sidekick_cloud_disaster.html`, 2009.

[13] Gregory Chockler, Dan Dobre, and Alex Shraer. Consistency and complexity tradeoffs for highly-available multi-cloud store. `http://people.csail.mit.edu/grishac/mcstore.pdf`, 2013.

[14] Gregory Chockler and Dahlia Malkhi. Active disk paxos with infinitely many processes. *Distrib. Comput.*, 18(1):73–84, July 2005.

[15] Brian F. Cooper, Raghu Ramakrishnan, Utkarsh Srivastava, Adam Silberstein, Philip Bohannon, Hans-Arno Jacobsen, Nick Puz, Daniel Weaver, and Ramana Yerneni. Pnuts: Yahoo!'s hosted data serving platform. *Proc. VLDB Endow.*, 1(2):1277–1288, August 2008.

[16] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM symposium on Cloud computing*, SoCC '10, pages 143–154, New York, NY, USA, 2010. ACM.

[17] Amazon DynamoDB. `http://aws.amazon.com/dynamodb/`.

[18] Seth Gilbert, Nancy A. Lynch, and Alexander A. Shvartsman. Rambo: a robust, reconfigurable atomic memory service for dynamic networks. *Distributed Computing*, 23(4):225–272, 2010.

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

[20] Michael R. Hines, Abel Gordon, Márcio Silva, Dilma Da Silva, Kyung Dong Ryu, and Muli Ben-Yehuda. Applications know best: Performance-driven memory overcommit with ginkgo. In *CloudCom*, pages 130–137, 2011.

[21] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. Zookeeper: wait-free coordination for internet-scale systems. In *Proceedings of the 2010 USENIX annual technical conference (ATC)*, Berkeley, CA, USA, 2010.

[22] P. Jayanti, T. Chandra, , and S. Toueg. Fault-tolerant wait-free shared objects. *Journal of the ACM*, 45(3):451–500, 1998.

[23] Michael Krigsman. MediaMax / The Linkup: When the cloud fails. `http://blogs.zdnet.com/projectfailures/?p=999`, 2008.

[24] Leslie Lamport. On interprocess communication. part i: Basic formalism. *Distributed Computing*, 1(2):77–85, 1986.

[25] Leslie Lamport. On interprocess communication. part ii: Algorithms. *Distributed Computing*, 1(2):86–101, 1986.

[26] Leslie Lamport. How to make a correct multiprocess program execute correctly on a multiprocessor. *IEEE Trans. Computers*, 46(7):779–782, 1997.

[27] Richard MacManus. More Amazon S3 Downtime: How Much is Too Much? `http://readwrite.com/2008/07/20/more_amazon_s3_downtime`, 2008.

[28] mongoDB. `http://www.mongodb.org/`.

[29] TClouds Project. Privacy and resilience for Internet-scale critical infrastructures. `http://www.tclouds-project.eu`.

[30] Jun Rao, Eugene J. Shekita, and Sandeep Tata. Using paxos to build a scalable, consistent, and highly available datastore. *PVLDB*, 4(4):243–254, 2011.

[31] Riak. `http://basho.com/riak`.

[32] Amazon Simple Storage Service (Amazon S3). `http://aws.amazon.com/s3/`.

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

[34] Amazon SimpleDB. http://aws.amazon.com/simpledb/.

[35] Google Cloud Storage. https://cloud.google.com/products/cloud-storage.

[36] Microsoft Azure Storage. http://www.windowsazure.com/en-us/manage/services/storage.

[37] Dan Williams, Hani Jamjoom, Yew-Huey Liu, and Hakim Weatherspoon. Overdriver: handling memory overload in an oversubscribed cloud. In *Proceedings of the 7th ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, VEE '11, pages 205–216, New York, NY, USA, 2011. ACM.

[38] Yunqi Ye, Liangliang Xiao, I-Ling Yen, and Farokh Bastani. Secure, dependable, and high performance cloud storage. In *Proceedings of the 29th Symposium on Reliable Distributed Systems (SRDS)*, pages 194–203, 2010.