Brief Announcement: Robust Data Sharing with Key-Value Stores

Cristina Basescu
Vrije Universiteit Amsterdam
1081 HV Amsterdam
Netherlands
cristina.basescu@gmail.com

Christian Cachin
IBM Research - Zurich
CH-8803 Rüschlikon
Switzerland
cca@zurich.ibm.com

Ittay Eyal
Technion — Israel Institute of Technology
Haifa 32000, Israel
ittay@tx.technion.ac.il

Robert Haas
IBM Research - Zurich
CH-8803 Rüschlikon
Switzerland
rha@zurich.ibm.com

Marko Vukolić
Eurécom
F-06904 Sophia Antipolis
France
vukolic@eurecom.fr

ABSTRACT

A key-value store (KVS) offers functions for storing and retrieving values associated with unique keys. KVSs have become widely used as shared storage solutions for Internet-scale distributed applications.

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

Categories and Subject Descriptors

C.2.4 [**Distributed Systems**]: Distributed Applications

General Terms

Algorithms, Theory

Keywords

Distributed Storage, Cloud Storage

1. INTRODUCTION

Recent years have seen an explosion of Internet-scale applications, ranging from web search to social networks. These applications are typically implemented with many machines running in multiple data centers. In order to coordinate their operation, these machines access some shared storage.

In this context, a prominent storage model is the *key-value store* (*KVS*). A KVS offers a range of simple functions for manipulation of unstructured data objects (called *values*), each one identified by a unique *key*. KVSs are used as storage services directly [2, 5] or indirectly, as non-relational (NoSQL) databases [15, 4]. While different services and systems offer various extensions to the KVS interface, the common denominator of existing KVS services implements an associative array: A client may *store* a value by associating the value with a key, *retrieve* a value associated with a key, *list* the keys that are currently associated, and *remove* a value associated with a key.

Storage services provide reliability using replication and tolerate the failure of individual data replicas. However, when all data replicas are managed by the same entity, there are naturally common system components, and therefore failure modes common to all replicas. A failure of these components may lead to data becoming not available or even being lost, as recently witnessed during an Amazon S3 outage [1] and Google's temporary loss of email data [3]. Therefore, a client can increase data reliability by replicating it among several storage services using the guarantees offered by *robust* distributed storage algorithms [13, 7]. Such an algorithm uses multiple storage providers, called *base objects* here, and emulates a single, more reliable shared storage abstraction, which we model as a *read/write register*. The register tolerates asynchrony, concurrency, and faults among the clients and the base objects.

Many well-known robust distributed storage algorithms exist [7, 10, 9]. Perhaps surprisingly, none of them direcly exploits key-value stores as base objects. The problem arises because existing solutions are either (1) unsuitable for KVSs since they rely on storage nodes that perform custom *computation*, which a KVS cannot do, or (2) prohibitively expensive, in the sense that they require as many base objects as there are clients [12, 6].

In the following, we describe the challenges behind running robust storage algorithms over a set of KVS base objects.

2. CHALLENGES

Many existing robust register emulations are based on versioning, in the sense that they associate each stored value

PODC'11, June 6–8, 2011, San Jose, California, USA.
ACM 978-1-4503-0719-2/11/06.

with a *version* (also called a timestamp) that increases over time. Consider the classical multi-writer emulation of a fault-tolerant register [13, 7]. A writer determines first the largest version from some majority of the base objects, derives a larger version, and then stores the new value together with the larger version at a majority of base objects. The base object then performs *computation* and actually stores the new value only if it comes with a larger version than the one it stores locally. However, a KVS does not offer such an operation.

Similar to existing emulations, we want the robust storage solution to be *wait-free* [14], such that every correct client may proceed independently of the speed or failure of other clients (or more precisely, every operation invoked by a correct client eventually completes).

If a classical algorithm is cast blindly into the KVS context without adjustment, all values are stored with the same key. This may cause a larger version and an associated, recently written value to be overwritten by a smaller version and an outdated value. We call this the *old-new overwrite* problem.

Another equally naïve solution is to store each version under a separate key; such a KVS accumulates all versions that have ever been stored and takes up unbounded space. As remedy for this, one could remove small versions from a KVS after a value with a larger version has been stored. But this might, in turn, jeopardize wait-freedom. Consider a read operation that lists the existing keys and then retrieves the value with the largest version. If this version is removed between the time when the KVS executes the list operation and the time when the client retrieves it from the KVS, the read operation will fail. We refer to this as the *garbage-collection race* problem.

3. CONTRIBUTION

We provide two robust, asynchronous, and efficient emulations of a register over a set of fault-prone KVS replicas. Both emulations are designed for an unbounded number of clients, which may all read from and write to the register (i.e., the emulations implement a multi-writer multi-reader register). This makes the algorithms appropriate for Internet-scale systems. Both emulations are *wait-free* and *optimally resilient*. The latter property means that the algorithm tolerates crash-stop failures of any minority of the KVS replicas and of any number of clients.

The two emulations differ in their consistency semantics. The first one emulates a multi-writer *regular* register [17][1] and it does not require read operations to write to KVSs (that is, to change the state of a KVS by storing a value). Precluding readers from storing values is practically appealing, since the clients may belong to different domains and not all of them should be permitted to write to the shared memory. But this poses a problem because of the garbage-collection race problem described previously. Our solution instructs a write operation to store the same value *twice*, under different keys: Once under an *eternal* key, which is never removed by garbage collection but vulnerable to an old-new overwrite, and a second time under a *temporary* key, named according to the version. Outdated temporary keys are garbage-collected periodically, for instance by write operations, which exposes them to garbage-collection races. Taken together, however, the eternal and temporary copies complement each other and guarantee a wait-free emulation with regular semantics.

The second algorithm emulates an *atomic* or *linearizable* register [16], where all read and write operations appear to execute at a single point in time between their invocation and response. This emulation requires read operations to store values at the underlying KVS replicas, but this cannot be avoided [16, 11]. We derive our atomic emulation from the regular emulation, by instructing the readers to write back the value they are about to return [7]. Details appear in the full version [8].

[1]Roughly, a regular read may return a value written by the latest write that precedes it or one of the concurrently written values [16, 17].

4. REFERENCES

[1] Amazon S3 availability event: July 20, 2008. `http://status.aws.amazon.com/s3-20080720.html`.

[2] Amazon Simple Storage Service. `http://aws.amazon.com/s3/`.

[3] Gmail back soon for everyone. `http://gmailblog.blogspot.com/2011/02/gmail-back-soon-for-everyone.html`.

[4] Project Voldemort: A distributed database. `http://project-voldemort.com/`.

[5] Windows Azure Storage. `http://www.microsoft.com/windowsazure/storage/`.

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

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

[8] C. Basescu, C. Cachin, I. Eyal, R. Haas, and M. Vukolić. Robust data sharing with key-value stores. Research Report RZ 3802, IBM Research, 2011.

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

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

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

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

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

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

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

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

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