On Limitations of Using Cloud Storage for Data Replication

Christian Cachin
IBM Research - Zurich
Säumerstr. 4
CH-8803 Rüschlikon, Switzerland
cca@zurich.ibm.com

Birgit Junker[1]
Open Systems AG
Räffelstrasse 29
CH-8045 Zurich, Switzerland
bju@open.ch

Alessandro Sorniotti
IBM Research - Zurich
Säumerstr. 4
CH-8803 Rüschlikon, Switzerland
aso@zurich.ibm.com

***Abstract*—Cloud storage services often provide a *key-value store (KVS)* functionality, an object-based interface for accessing a collection of unstructured data items or *blobs*. Every blob is associated with a key that serves as identifier to access the blob. In the simplest form, a key-value store provides only methods for writing and reading an entire blob, for removing blobs, and for listing all defined keys. On the other hand, many existing schemes for replicating data with the goal of enhancing resilience (e.g., based on quorum systems) associate logical *timestamps* with the stored values, in order to distinguish multiple versions of the same data item.**

This paper uses the consensus number of a shared storage abstraction as a measure for its power to facilitate the implementation of data replication. It is demonstrated that a KVS is a very simple primitive, not different from read/write registers in this sense, and that a replica capable of the typical operations on timestamped data is fundamentally more powerful than a KVS. Hence, data replication schemes over storage providers with a KVS interface are inherently more difficult to realize than replication schemes over providers with richer interfaces.

***Keywords*-Data storage; resilience; replication; quorum systems; wait-freedom; consensus number.**

I. Introduction

One way of enhancing the dependability of a cloud service consists of connecting multiple clouds to an *intercloud* or a *cloud-of-clouds* and running the service on multiple replicas in different clouds. This approach tolerates service outages and security incidents affecting individual clouds. Although existing cloud platforms provide high availability and reliability using internal replication, some common failure modes remain and only intercloud replication tolerates those. The platform of a cloud provider is usually treated as a single security domain and has little internal diversification. Hence, only an intercloud provides multiple administrative domains; this is especially important for building intrusion-tolerant systems [1], [2].

Today cloud storage is often provided via the abstraction of a *key-value store (KVS)*, an object-based interface for accessing a collection of unstructured *blobs*. Every blob is associated with a key that serves as identifier to access the blob. The common denominator of the KVS services available today contains only methods for writing and reading an entire blob, for removing blobs, and for listing all defined keys. This simplistic interface poses a problem for replication schemes that maintain versioned data on multiple KVS replicas. In particular, the existing solutions for reliable storage either do not work or incur a prohibitively large storage overhead. Băsescu et al. [3] address this problem in detail and present a replication algorithm that maintains two copies of the stored value per KVS in the common case. They also show that storing two copies is necessary, in order to achieve wait-free client operations.

Data replication in the intercloud has recently received a lot of attention [4], [5], [6], [7]. One class of such systems assume specialized interfaces on the storage replicas, which are capable of limited processing and command execution (like active disks [8] or the storage servers of HAIL [4] and Cleversafe [6]). Replicas can therefore carry out limited computation, such as comparing timestamps and conditionally storing data. This feature is required by many replicated storage algorithms based on quorum systems, starting with some of the first schemes [9], [10], [11]. A second class of cloud-data replication schemes, in particular RACS [5] and DepSky [7], uses a KVS provider; they compensate for the relative simplicity of the KVS interface by adding extra components for synchronization among multiple clients.

In this paper, we identify an inherent difference between the storage abstractions used by the two classes of replication systems mentioned before. We examine the power of the popular KVS model from the perspective of the designer of a failure- and intrusion-tolerant replication scheme. A replication method enables multiple clients to operate on a storage abstraction emulated from a pool of potentially faulty storage providers, such as cloud-based KVSs; implicitly, the richness, complexity, and performance of the emulated service depends on the power of the underlying primitives. We analyze the *consensus number* storage abstraction as a measure for its capability to provide wait-free synchronization among the set of clients according to Herlihy's fundamental notion [12], [13].

Surprisingly, we find that the typical processing steps expected from a replica in traditional replicated storage schemes give it universal power — these replicas have

[1]Work done at IBM Research - Zurich.

infinite consensus number and are as powerful as the consensus abstraction for implementing other concurrent data structures. A key-value store, on the other hand, has the least amount of synchronization power available in any shared object that has been studied — the consensus number of a KVS is one and falls into the same class as a simple read/write register.

Our result illustrates why data replication for typical cloud storage systems requires additional mechanisms for letting multiple clients access the system concurrently.

This paper continues by introducing the model and background in the next section. Subsequently Section III addresses the consensus number of a replica, which can process timestamped data, and Section IV analyzes the KVS abstraction. Section V discusses implications of the result.

II. Model

This section introduces the formal model. A comprehensive introduction to wait-free synchronization algorithms for shared-memory systems is provided by Herlihy and Shavit [13]. Much of the relevant background work was developed in the context of algorithms for multi-processors, where a set of multiple "processors" concurrently access resources in a "shared memory." With the advent of cloud computing, this model has expanded its scope: it applies analogously to multiple *clients* accessing shared resources in the *cloud*. We focus on one shared object and assume correct clients, in contrast to the intended application domain, where clients access a set of potentially faulty objects.

A. Linearizability

In the system model considered here, an unbounded number of *clients* access the *operations* provided by a shared *object* O. Since the system is asynchronous and the execution of an operation is not instantaneous, each operation is represented by two events, denoting the *invocation* and the *response*. The sequence of invocations and responses of O occurring in an execution σ are called a *history*. An invocation and a response *match* if they are both events occurring at the same client, concern the same object, and the response occurs after the invocation and before any other invocation concerning the same object. An operation whose invocation appears in a history is *complete* if the history also contains a matching response. The subsequence of σ containing only the complete operations of σ is denoted by *complete*(σ). If an operation is not complete in a history it is called *pending*. An *extension* of σ is any history that can be obtained from σ by appending responses for any subset of the pending requests in σ.

In a sequence of events σ an operation o *precedes* another operation o' if o completes before o' is invoked. This is denoted by $o <_\sigma o'$. If neither of two operations precedes the other, they are *concurrent*. A sequence of events that does not contain any concurrent events is called *sequential*. We assume that every client invokes operations on one object in a *well-formed way*, that is, the client never invokes an operation on an object when an operation by that client on the same object is pending.

A history π consisting only of events in a history σ is said to *preserve the real-time order* of σ if for any two operations o and o' in π, the condition $o <_\sigma o'$ implies that $o <_\pi o'$.

A *client subhistory* of σ for a client c is the subsequence of σ that contains only those events that occur at c; it is denoted by $\sigma|_c$. For an object O, the *object subhistory* $\sigma|_O$ is defined analogously. Two histories σ and σ' are *equivalent* if for every client c it holds that $\sigma|_c = \sigma'|_c$.

The *sequential specification* defines a shared object by describing its behavior in sequential executions. A history σ is *legal* if each of its object subhistories is legal with respect to the sequential specification.

An important class of objects appear to execute operations "atomically," as captured by the notion of *linearizability* formalized by Herlihy and Wing [14]. We consider only linearizable semantics in this work.

Definition 1 (Linearizability). A history of events σ is *linearizable* if it has an extension σ' and there exists a legal sequential history π such that:

1) *complete*(σ') is equivalent to π; and
2) π preserves the real-time order of σ.

B. Wait-freedom

In a system where several clients execute operations on shared object(s), none of the clients should be prevented from making progress due to operations of other clients. A system achieving this property is called *wait-free*. This is an important aspect of resilient Internet services. The notion was made formal by Herlihy [12] but probably appears first in the work of Lamport [15]. We consider only wait-free systems in the remainder of this work.

Definition 2 (Wait-Freedom). Consider a system where several clients access a shared object. The system is *wait-free* if in all executions, every client gets a response to an operation invocation within a finite number of steps, that is, independent of failures and of actions of the other clients.

C. Consensus number

The *consensus problem* requires multiple clients to agree on a common value from a set of proposed values. A *consensus object* abstracts a service that provides a wait-free implementation of a distributed protocol that solves the consensus problem [12].

Definition 3 (Consensus Object). A *consensus object* is a shared object with one operation *decide*(v); it takes a value v, called a *proposal*, as input parameter and returns a *decision value*. Every client calls *decide* with its own proposal v at most once. The returned decision value d satisfies:

1) *(Validity)* The value d is the proposal of some client; and
2) *(Consistency)* All clients return the same decision value d.

A consensus object permits any number of clients. For simplicity we consider only *binary consensus* in this work, where the proposals are either zero or one.

Next we introduce the concept of consensus numbers, which is an important measure for the synchronization power of a shared object in a wait-free system.

Definition 4 (Consensus Number [12]). The *consensus number* of a shared object is the maximum number of clients for which this object can solve the consensus problem. If no maximum exists, the consensus number is said to be infinite.

The consensus number provides a measure for classifying shared objects with respect to their power to synchronize multiple concurrent clients. By definition, a consensus object has infinite consensus number; consensus has been called *universal* for this reason. One of the simplest objects considered in the literature is a read/write register; it has consensus number one [13]. The following result, first shown by Herlihy [12], organizes all shared objects in a hierarchy based on their consensus numbers.

Proposition 1 ([12]). *If an object X has consensus number n and another object Y has consensus number $m < n$, then X cannot be implemented in a wait-free way from Y in a system with more than m clients.*

In other words, a given shared object is strictly more powerful than any other shared object that has a smaller consensus number.

III. The consensus number of a replica

Many protocols that implement robust shared memory in distributed systems use the notion of logical timestamps [16] for identifying different versions of a stored value over time. They usually maintain the stored value in the form of a pair, consisting of a timestamp ts and the actual value v.

We now introduce a *replica object*, which is inherent in a large number of distributed implementations of shared memory; it corresponds, for example, to the processors used by Attiya et al. [10] and to the active disks of Chockler and Malkhi [8]. Our replica object provides functionality to conditionally store a timestamp/value pair, which is required from the storage primitive in many robust shared storage implementations. It serves any number of clients.

More precisely, a replica object R stores a timestamp/value pair internally and offers two operations, called *condwrite* and *read*, as shown in Algorithm 1. Operation *condwrite*(ts, v) takes a timestamp/value pair as input and returns a constant symbol; it only stores the value v in the replica object if the timestamp ts is bigger than the internally stored timestamp. Operation *read* takes no input and returns a value; it simply accesses the internally stored value and returns it.

Algorithm 1 Sequential spec. of the replica object R

```
state
    $(R.ts, R.v)$, initially $(0, \perp)$;

operation condwrite$(ts, v)$
    if $ts > R.ts$ then
        $(R.ts, R.v) \leftarrow (ts, v)$;
    return ACK;

operation read$()$
    return $R.v$;
```

From the point of view of synchronization, replica objects can be used to implement a consensus object for any number of clients. Hence, a replica is universal and can implement any synchronization primitive.

Theorem 2. *The consensus number of a replica object is infinite.*

Proof: We show how to implement a consensus object C from a replica object R. The emulation is shown in Algorithm 2 and works as follows. At the start, the timestamp/value pair at R is initialized to $(0, \perp)$. When a client invokes *decide*(v) of C with a proposal v, then the emulation tries to write the pair $(1, v)$ to R. Subsequently it reads the value stored by R and returns it as the decision value.

Algorithm 2 Implementation of a consensus object C using a replica R

```
operation decide$(v)$
    $R$.condwrite$(1, v)$;
    $d \leftarrow R$.read$()$;
    return $d$;
```

The intuition behind this emulation is that only the first invocation of *condwrite* executed by R will succeed in storing a value, say v_1, at R; it does not matter which client executes it. Every subsequent conditional write is simply ignored because R already stores timestamp 1. Once the first client has decided v_1, every other client that invokes *decide*(v) also obtains v_1.

Object C is wait-free because the implementation contains no loops, the underlying replica R is wait-free, and every operation immediately returns. Furthermore, C satisfies the *validity* property of a consensus object, because decided value is read from R and because the proposal of at least one client is written to R before it is read. Hence, the decided value must be an input from a client. Moreover, C also implements *consistency* because only the first ever invocation of *condwrite* may change the value the is returned from R by *read*. Hence, C is a consensus object according to Definition 3.

Note that there is no upper bound on the number of clients that can write to or read from R; therefore, this implementation solves consensus for any number of clients. According to Definition 4, together with Proposition 1, this implies that the consensus number of a replica object is infinite. ■

IV. THE CONSENSUS NUMBER OF A KEY-VALUE STORE

A key-value store (KVS) represents an object-based storage service, which has become popular in the context of cloud storage. Pioneered by Amazon S3 [17], it now represents a de-facto standard for many commercial cloud storage services (e.g., Windows Azure [18], Rackspace [19], and many others [20]).

This section illustrates the fundamental power of a KVS for wait-free synchronization. We show how to implement a KVS from a so-called snapshot object in a wait-free manner. Snapshot objects represent a prominent abstraction of shared storage with many applications. Since a snapshot object can be implemented from register objects, we can show that the KVS object has consensus number one.

A. Key-value store objects

A *key-value store* object, abbreviated *KVS*, is an associative array that allows storage and retrieval of *values* in a set V associated with *keys* in a set K. The size of the stored values is typically much larger than the length of a key, so the values in V cannot be translated to elements of K and be stored as keys.

A KVS supports four operations, formally specified in Algorithm 3 [3]. The operations support (1) *storing* a value *val* associated with a key *key* (denoted *put*(*key*, *val*)), (2) *retrieving* a value *val* associated with a key (*val* ← *get*(*key*)), which may also return FAIL if *key* does not exist, (3) *listing* the keys that are currently associated (l ← *list*()), and (4) *removing* a value associated with a key (*remove*(*key*)).

Algorithm 3 Sequential spec. of a key-value store object

```
state
    $live \subseteq \mathcal{K} \times \mathcal{V}$, initially $\emptyset$;

operation $put(key, val)$
    $live \leftarrow (live \setminus \{(key, v) \mid v \in \mathcal{V}\}) \cup (key, val)$;
    return ACK;

operation $get(key)$
    if $\exists v \in \mathcal{V}$ such that $(key, v) \in live$ then
        return $v$;
    else
        return FAIL;

operation $list()$
    return $\{key \mid \exists v : (key, v) \in live\}$;

operation $remove(key)$
    $live \leftarrow live \setminus \{(key, v) \mid v \in \mathcal{V}\}$;
    return ACK;
```

B. Snapshot objects

A *snapshot object* [21], abbreviated *SO*, is a shared object that stores n values in a system of n clients, one value per client. In a single-writer snapshot object, as considered here, every value may only be written by the corresponding client and all clients may read all values.

More precisely, an atomic snapshot object SO maintains a vector D of n values from a domain $\mathcal{V}$ and provides two operations, denoted *update* and *scan*. When a client with an index $i \in \{1, \ldots, n\}$ invokes $update(i, v)$ for a value $v \in \mathcal{V}$, then SO atomically sets $D[i] \leftarrow v$ and responds with an acknowledgment. No client may invoke *update* with the index of another client. Operation *scan*() with no parameters may be invoked by any client and returns the vector D.

The sequential specification of an atomic snapshot object requires that for each D returned by *scan*, entry $D[i]$ for $i = 1, \ldots, n$ equals the value d given in the most recent preceding $update(i, d)$ operation by the client with index i; if there is no such preceding *update*, then $D[i]$ is equal to $\bot$.

Interestingly, one can implement an atomic snapshot object for any number of clients only from atomic read/write registers [21]. Because registers have consensus number one, atomic snapshot objects have consensus number one.

C. From snapshot objects to KVS objects

This section gives a constructive proof of the following theorem, by exhibiting a wait-free implementation of a KVS object from atomic snapshot objects.

Theorem 3. *The consensus number of a key-value store object is one.*

Proof: We implement a KVS object K with one atomic snapshot object SO. Recall that clients interact with every object in a well-formed manner.

The idea behind the emulation is to maintain in $SO[i]$ a list of all *put* and *remove* operations executed on the KVS by the client with index i. Every such operation is represented by a tuple $(i, ts, key, val) \in \{1, \ldots, n\} \times \mathbb{N} \times \mathcal{K} \times \mathcal{V}$, where ts represents a logical timestamp that is incremented by every client when it executes an operation.

The history of operations of the client with index i, as stored in $D[i]$, consists of a concatenated list of such tuples:

$$D[i] \;=\; (i, ts, key, val) \,\|\, \cdots \,\|\, (i, ts', key', val').$$

For two tuples $\tau = (i, ts, key, val)$ and $\tau' = (j, ts', key', val')$ we define a *tuple order* relation and say that τ is *bigger* than τ', denoted $\tau > \tau'$, whenever $ts > ts'$ or $ts = ts' \wedge i > j$.

We next describe the implementation; a formal statement appears in Algorithm 4.

Algorithm 4 Implementation of a KVS object K form a snapshot object SO.

```
operation put(key, val) by client with index $i$
    $D \leftarrow SO.scan()$;
    $t \leftarrow 0$;
    for $j \in \{1, \ldots, n\}$ and $(j, ts, k, v) \in D[j]$ do
        if $k = key$ and $t < ts$ then
            $t \leftarrow ts$;
    $t \leftarrow t + 1$;
    $d \leftarrow D[i] \,\|\, (i, t, key, val)$;
    $SO.update(i, d)$;
    return;

operation get(key) by client with index $i$
    $D \leftarrow SO.scan()$;
    $(t, val) \leftarrow (0, \perp)$;
    for $j \in \{1, \ldots, n\}$ and $(j, ts, k, v) \in D[j]$ do
        if $k = key$ and $t < ts$ then
            $(t, val) \leftarrow (ts, v)$;
    return val;

operation remove(key) by client with index $i$
    put(key, $\perp$);                      // invoke the operation on itself
    return;

operation list()
    $D \leftarrow SO.scan()$;
    let $L$ be the set of distinct keys from $D$, i.e.,
        $L \leftarrow \{k | (j, ts, k, v) \in \bigcup_{i=1}^{n} D[i]\}$;
    $K \leftarrow \emptyset$;
    for $key \in L$ do
        let $(j, ts, k, v)$ be the biggest tuple in $\bigcup_{i=1}^{n} D[i]$
            with $k = key$ according to tuple order;
        if $v \neq \perp$ then
            $K \leftarrow K \cup \{key\}$;
    return $K$;
```

The operation *put*(key, val) by a client with index i first scans SO and retrieves all histories. Then it determines the maximal tuple (j, ts, k, v) from all histories according to tuple order. It increments the timestamp, sets $t \leftarrow ts + 1$, appends the tuple (i, t, key, val) to the list in $D[i]$, and uses the result to update its entry in SO. Recall that only the client with index i may invoke *update*$(i, \cdot)$.

Operation *get*(key) just scans the snapshot object and searches in the returned histories for the largest tuple according to tuple order whose key equals *key*; denote this by (i, ts, key, val). If such a tuple is found, the operation returns *val*; otherwise, it returns $\perp$.

To execute *remove*(key), the implementation stores the special character $\perp$ under *key* using the *put* operation already implemented.

Finally, the *list*$()$ operation scans the snapshot object, examines every tuple from every history, and retains, for every key, the maximal tuple according to tuple order. These tuples are collected in a set K. The list to return is then obtained by extracting the keys of those tuples from K in which the value is not $\perp$, i.e., those that have not been removed.

Note that the size of $D[i]$ could be reduced without affecting the emulation: operations that have been superseded by other operations (for the same key but with a larger timestamp) can be eliminated to save space.

We now show that this implementation produces linearizable histories that satisfy the specification of a KVS according to Algorithm 3. Given a history σ, we construct a legal sequential history π that satisfies the properties of linearizability and argue that it is legal.

- The empty history is legal.
- Any history without concurrent operations is legal. This follows because the timestamps are strictly monotonically increasing during all *put* and *remove* operations. As the *get* operation returns the value with the highest timestamp in tuple order, *get* returns the most recently written value. The same argument shows that the output of the *list* operation is legal.
- Two concurrent *get* and/or *list* operations with no concurrent *put* or *remove* operations can be ordered in any way, since they do not affect each other.
- Two concurrent *put* and/or *remove* operations with the same key are scheduled according to the order on the tuples that represent them. Suppose clients p and r are concurrently invoking operations $\omega_p = put$ and $\omega_r = remove$ with the same key. The operations are scheduled such that $\omega_p >_\pi \omega_r$ if and only if the tuple representing ω_p is bigger than the tuple representing ω_r. Note that $p \neq r$ because all clients execute operations in a well-formed way. Thus, one of the two tuples is strictly bigger than the other in tuple order. All subsequent *get* and *list* operations also return the result of the operation (ω_p or ω_r) that is scheduled last. Another subsequent put and remove operation with the same key will increment the timestamp, hence, the results of ω_p or ω_r are never returned afterwards.
- Consider now a *put* or *remove* operation that is concurrent to *get* or *list* operation. We observe that the *update* on SO near the end of the *put* and *remove* operations and the *scan* on SO at the beginning of the *get* and *list* operations are linearizable because SO is atomic.
  We schedule a *put* or *remove* operation ω in π before a concurrent *get* or *list* operation ρ whenever the *update* in ω precedes the *scan* in ρ. If $\omega <_\pi \rho$, then ω incremented the timestamp and ρ returns the value written by ω because it has the maximal timestamp, as required. Otherwise, if $\rho <_\pi \omega$, then ρ appears in π before ω. This is also legal since the *scan* operation of ρ has not been affected by the *update* operation in ω according to the construction of π. Hence, the value returned by ρ is legal.
- Finally, consider multiple concurrent *put* and/or *remove* operations. As shown before, executions with one *put* or *remove* operation concurrent to one further operation

can be linearized.

Assume that $k-1$ *put* and *remove* operations have been linearized and consider the k-th *put* or *remove* operation ω. It is scheduled:

  - after all *get* and *list* operations whose embedded *scan* precedes the *update* operation in ω;
  - before all *get* and *list* operations whose embedded *scan* is scheduled after the *update* operation in ω;
  - before all *put* and *remove* operations by clients with a higher index; and
  - after all *put* and *remove* operations by clients with a smaller index.

  It is straightforward to verify that π constructed like this is sequential and legal.

- History π preserves the real-time order of σ because no sequential operations are reordered.

This completes the construction of our wait-free implementation of a KVS object from a single-writer atomic snapshot object. Since the snapshot object has consensus number one, Proposition 1 implies that the consensus number of a KVS object is also one. ■

V. Conclusion

This paper shows that the consensus number of a typical storage replica in timestamp-based replication algorithms is infinite, but a KVS, provided by most cloud storage services, has consensus number one. Therefore these two providers have fundamentally different power for synchronizing operations of multiple clients in wait-free algorithms (formally captured in Proposition 1). This result explains why replication algorithms using KVS providers, such as DepSky [7] and Intercloud Storage [3], use more complex methods to synchronize multiple clients than traditional data replication schemes.

Our result also gives an incentive for considering extensions of the KVS interface, such as the active KVS model introduced recently [22].

Acknowledgments

We thank Sabrina Pérez and Marko Vukolić for interesting discussions and helpful comments.

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

References

[1] F. B. Schneider and L. Zhou, "Implementing trustworthy services using replicated state machines," *IEEE Security & Privacy Magazine*, pp. 34–43, Sep. 2005.

[2] M. Garcia, A. N. Bessani, I. Gashi, N. F. Neves, and R. R. Obelheiro, "OS diversity for intrusion tolerance: Myth or reality?" in *Proc. DSN*, 2011, pp. 383–394.

[3] C. Băsescu, C. Cachin, I. Eyal, R. Haas, A. Sorniotti, M. Vukolić, and I. Zachevsky, "Robust data sharing with key-value stores," in *Proc. DSN*, Jun. 2012.

[4] K. D. Bowers, A. Juels, and A. Oprea, "HAIL: A high-availability and integrity layer for cloud storage," in *Proc. CCS*, 2009, pp. 187–198.

[5] H. Abu-Libdeh, L. Princehouse, and H. Weatherspoon, "RACS: A case for cloud storage diversity," in *Proc. SOCC*, 2010.

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

[7] A. Bessani, M. Correia, B. Quaresma, F. André, and P. Sousa, "DepSky: Dependable and secure storage in a cloud-of-clouds," in *Proc. EuroSys*, 2011, pp. 31–46.

[8] G. Chockler and D. Malkhi, "Active disk Paxos with infinitely many processes," *Distributed Computing*, vol. 18, no. 1, pp. 73–84, 2005.

[9] B. Charron-Bost, F. Pedone, and A. Schiper, Eds., *Replication: Theory and Practice*, ser. Lecture Notes in Computer Science. Springer, 2010, vol. 5959.

[10] H. Attiya, A. Bar-Noy, and D. Dolev, "Sharing memory robustly in message-passing systems," in *Proc. PODC*, 1990, pp. 363–375.

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

[12] M. Herlihy, "Wait-free synchronization," *ACM Transactions on Programming Languages and Systems*, vol. 11, no. 1, pp. 124–149, Jan. 1991.

[13] M. Herlihy and N. Shavit, *The Art of Multiprocessor Programming*. Morgan Kaufmann, 2008.

[14] M. P. Herlihy and J. M. Wing, "Linearizability: A correctness condition for concurrent objects," *ACM Transactions on Programming Languages and Systems*, vol. 12, no. 3, pp. 463–492, Jul. 1990.

[15] L. Lamport, "A new solution of Dijkstra's concurrent programming problem," *Communications of the ACM*, vol. 17, no. 8, pp. 453–455, 1974.

[16] ——, "Time, clocks, and the ordering of events in a distributed system," *Communications of the ACM*, vol. 21, no. 7, pp. 558–565, Jul. 1978.

[17] "Amazon Simple Storage Service," http://aws.amazon.com/s3/.

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

[19] "Rackspace hosting," http://www.rackspacecloud.com/cloud_hosting_products/files/.

[20] "jclouds — multi-cloud library," http://www.jclouds.org/.

[21] Y. Afek, H. Attiya, D. Dolev, E. Gafni, M. Merritt, and N. Shavit, "Atomic snapshots of shared memory," *Journal of the ACM*, vol. 40, no. 4, pp. 873–890, 1993.

[22] R. Geambasu, A. A. Levy, T. Kohno, A. Krishnamurthy, and H. M. Levy, "Comet: An active distributed key-value store," in *Proc. OSDI*, 2010.