Write Fast, Read in the Past: Causal Consistency for Client-side Applications

Marek Zawirski, Nuno Preguiça, Sérgio Duarte, Annette Bieniusa, Valter Balegas, Marc Shapiro

To cite this version:

Marek Zawirski, Nuno Preguiça, Sérgio Duarte, Annette Bieniusa, Valter Balegas, et al.. Write Fast, Read in the Past: Causal Consistency for Client-side Applications. Int. Conf. on Middleware (MIDDLEWARE), Dec 2015, Vancouver, BC, Canada. pp.75-87, 10.1145/2814576.2814733 . hal-01248194

HAL Id: hal-01248194
https://hal.inria.fr/hal-01248194

Submitted on 24 Dec 2015

Write Fast, Read in the Past: Causal Consistency for Client-Side Applications

Marek Zawirski
Sorbonne Universités, Inria, UPMC-LIP6, Paris, France*

Nuno Preguiça
NOVA LINCS, DI, FCT, Universidade NOVA de Lisboa, Portugal

Sérgio Duarte
NOVA LINCS, DI, FCT, Universidade NOVA de Lisboa, Portugal

Annette Bieniusa
U. of Kaiserslautern, Germany

Valter Balegas
NOVA LINCS, DI, FCT, Universidade NOVA de Lisboa, Portugal

Marc Shapiro
Sorbonne Universités, Inria, UPMC-LIP6, Paris, France

ABSTRACT

Client-side apps (e.g., mobile or in-browser) need cloud data to be available in a local cache, for both reads and updates. For optimal user experience and developer support, the cache should be consistent and fault-tolerant. In order to scale to high numbers of unreliable and resource-poor clients, and large database, the system needs to use resources sparingly. The SwiftCloud distributed object database is the first to provide fast reads and writes via a causally-consistent client-side local cache backed by the cloud. It is thrifty in resources and scales well, thanks to consistent versioning provided by the cloud, using small and bounded metadata. It remains available during faults, switching to a different data centre when the current one is not responsive, while maintaining its consistency guarantees. This paper presents the SwiftCloud algorithms, design, and experimental evaluation. It shows that client-side apps enjoy the high performance and availability, under the same guarantees as a remote cloud data store, at a small cost.

Categories and Subject Descriptors

C.2.4 [**Computer-Communication Networks**]: Distributed Systems—distributed databases; H.3.4 [**Information Storage and Retrieval**]: Systems and Software—distributed systems

General Terms

Algorithms, Design, Performance, Reliability

Keywords

Geo-replication, Causal Consistency, Eventual Consistency, Fault Tolerance, Client-side Storage

1. INTRODUCTION

Client-side applications, such as in-browser and mobile apps, are not well supported by current technology. Such apps need, not only to share data remotely, but also to cache and update data locally in order to remain responsive at all times. Existing cloud databases provide only remote data sharing. Developers may implement caching and buffering at the application level, but, in addition to adding complexity, this approach cannot provide system-wide consistency, session guarantees, or fault tolerance. Recent frameworks such as Google Drive Realtime API, TouchDevelop or Mobius [10, 12, 13] do cache data at the client side on a small scale, but do not ensure system-wide guarantees either. Even recent algorithms designed for geo-replication across data centres [3, 5, 17, 25, 26] or small numbers of clients [8, 28] are not adequate for the task, as they are not designed to scale to large numbers of client-side replicas.

We argue here for a distributed database design that includes safety and scalability guarantees to client-side application developers. This system should address the (somewhat conflicting) requirements of consistency, availability, and convergence [27], and uphold them at least as well as existing server-side systems.

Since updates should be always available, concurrency is inevitable. However, no update should be lost, and the database replicas should never diverge permanently. Under these requirements, the strongest possible consistency model is *convergent causal consistency* with support for application-specific concurrency resolution [4, 25, 27, 33].

High numbers of client-side replicas challenges classical approaches: *(i)* To track causality precisely, per client replica, creates unacceptably fat metadata; but the more compact server-side metadata management has fault-tolerance issues. *(ii)* Full replication at high numbers of resource-poor devices would be unacceptable [8]; but partial replication of data and metadata could cause anomalous message delivery or unavailability. *(iii)* Unlike many previous approaches [3, 17, 25, 26], fault tolerance and consistency cannot be solved by assuming that the application is located inside the data centre (DC), or has a sticky session to a single DC [6, 35]. We analyse the above challenges in more detail hereafter.

This work addresses these challenges. We present the algorithms, design, and evaluation of SwiftCloud, the first distributed object store designed for a high number of client-

*Now at Google Inc.

Middleware '15 December 07-11, 2015, Vancouver, BC, Canada
© 2015 ACM ISBN 978-1-4503-3618-5/15/12$15.00
DOI: http://dx.doi.org/10.1145/2814576.2814733.

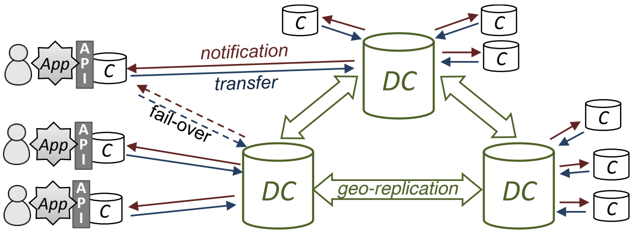


Figure 1: System components (Application processes, Clients, Data Centres), and their interfaces.

side replicas. It efficiently ensures consistent, available, and convergent access to client nodes, tolerating failures. To enable both small metadata and fault tolerance, SwiftCloud uses a flexible client-server topology, and decouples reads from writes. The client *writes fast* into its own cache, and *reads in the past* (also fast) data that is consistent, but occasionally stale.

Our main contribution is a combination of two techniques:

Cloud-backed support for partial replicas (§3).

A DC serves a consistent view of the database to a client, which the client merges with its own updates. In some failure situations, a client may connect to a new DC that happens to be inconsistent with its previous one. Because the client does not have a full replica, it cannot fix the issue on its own. We leverage "reading in the past" to avoid this situation in the common case, and provide control over the inherent trade-off between staleness and unavailability. More precisely, a client observes a *remote* update only if it is stored in some number $K \geq 1$ of DCs [28]; the higher the value of K, the more likely that a K*-stable* version is in both DCs, but the higher the staleness.

Protocols with decoupled, bounded metadata (§4).

SwiftCloud features a decoupled metadata design [23] that separates *tracking causality*, which uses small vectors assigned in the background by DCs, from *unique identification*, based on client-assigned scalar timestamps. Thanks to this design, to the flexible client-server topology, and to "reading in the past," metadata remains small and bounded in size. Furthermore, a DC can prune its log independently of clients, replacing it with a summary of delivered updates.

We implement SwiftCloud and demonstrate experimentally that our design reaches its objective, at a modest staleness cost. We evaluate SwiftCloud in Amazon EC2, against a port of WaltSocial [34] and against YCSB [14]. When data is cached, response time is two orders of magnitude lower than for server-based protocols with similar availability guarantees. With three servers in different DCs, the system can scale to 2,500 of client replicas. Metadata size does not depend on the number of clients, the number of failures, or the size of the database, and increases only slightly with the number of DCs: on average, 15 bytes of metadata per update, compared to kilobytes for previous algorithms with similar safety guarantees. Throughput is comparable to server-side replication, and improved for high locality workloads. When a DC fails, its clients switch to a new DC in under 1000 ms, and remain consistent. Under normal conditions, 2-stability causes fewer than 1% stale reads.

An extended version of this work is available as a technical report [36].

2. PROBLEM OVERVIEW

We consider support for a variety of client-side applications, sharing a database of **objects** that clients can read and update. We aim to scale to thousands of clients, spanning the whole internet, and to a database of arbitrary size.

Fig. 1 illustrates our system model. A cloud infrastructure connects a small set (say, tens) of geo-replicated data centres, and a large set (thousands) of clients. A DC has abundant computational, storage and network resources. Similarly to Sovran et al. [34], we abstract a DC as a powerful sequential process that hosts a **full replica** of the database.[1] DCs communicate in a peer-to-peer way. A DC may fail and recover with its persistent memory intact.

Clients do not communicate directly, but only via DCs. Normally, a client connects to a single DC; in case of failure or roaming, to zero or more. A client may fail and recover (e.g., disconnection while travelling) or fail permanently (e.g., destroyed phone), both without prior warning. We consider only non-byzantine failures.

Client-side apps need to respond quickly and at all times, i.e., they require high **availability** and **responsiveness**. This can be achieved by replicating data locally, and by synchronising updates in the background. However, a client has limited resources; therefore, it hosts a **cache** that contains only the small subset of the database of current interest to the local app. It should not have to receive messages relative to objects that it does not currently replicate [31]. Finally, control messages and piggy-backed metadata should have small and bounded size.

Since a client replica is only *partial*, there cannot be a guarantee of complete availability. The best that can be expected is **conditional availability**, whereby an operation returns without remote communication if the requested object is cached; and after retrieving the data from a remote node (DC) if not. If the data is not there and the network is down, the operation may be unavailable, i.e., it either blocks or returns an error.

2.1 Consistency with convergence

Application programmers and users wish to observe a consistent view of the global database. However, with availability as a requirement, consistency options are limited [4, 19, 20, 27].

Causal consistency.

The strongest available and convergent model is causal consistency [2, 4, 27].

Informally, under causal consistency, every process observes a *monotonically non-decreasing set of updates that includes its own updates, in an order that respects the causality between operations.*[2] Specifically, if an application process reads x, and later reads y, and if the state of x causally-depends on some update u to y, then the state of y that it reads will include update u. When the application requests y, we say there is a **causal gap** if the local replica has not yet received u. A consistent system must detect such a gap,

[1] We refer to prior work for the somewhat orthogonal issues of parallelism and fault-tolerance within a DC [3, 17, 25, 26].

[2] This subsumes the well-known session guarantees [11, 35].

and wait until u is delivered before returning y, or avoid it in the first place. If not, inconsistent reads expose both programmers and users to anomalies caused by gaps [25, 26].

We extend causal consistency with multi-operation **causal transactions**. Such a transaction reads from a **causally-consistent snapshot**, and is **atomic**, i.e., either all its updates are visible, or none is [25, 26]. The transaction's updates are considered to causally depend on all of the transaction's reads.

Convergence.

Applications require **convergence**, which consists of liveness and safety properties: *(i)* **At-least-once delivery**: an update that is delivered (i.e., is visible by the app) at some node, is delivered to all interested nodes (i.e., nodes that replicate or cache the updated object) after a finite number of message exchanges; *(ii)* **Confluence**: on two nodes that have delivered the same set of updates for an object, the object has the same value.

Causal consistency does not guarantee confluence, as two replicas might receive the same updates in different orders. For confluence, we rely on CRDTs, high-level data types with rich confluent semantics [11, 33]. An update on a high-level object is not just an assignment, but is a method associated with the object's type. For instance, a Set object supports add(element) and remove(element); a Counter supports increment() and decrement().

CRDTs include primitive last-writer-wins register (LWW) and multi-value register (MVR) [16, 21], but also higher level types such as Sets, Lists, Maps, Graphs, Counters, etc. [1, 32–34]. Efficient support of high-level objects requires the system to ensure both causal consistency (e.g., in a Set object, to ensure that remove update is delivered after adds it depends on) and **at-most-once delivery**, since many updates are not idempotent (e.g., incrementing a Counter).

Although each of these requirements may seem familiar or simple in isolation, the combination with scalability to high numbers of nodes and database size is a new challenge.

2.2 Metadata design

Metadata serves to identify updates and to ensure correct delivery. Metadata is piggy-backed on update messages, increasing the cost of communication.

One common metadata design assigns each update a timestamp as soon as it is generated on some originating node. The causality data structures tend to grow "fat." For instance, dependency lists [25] grow with the number of updates [17, 26], whereas version vectors [8, 28] grow with the number of clients. (Indeed, our experiments hereafter show that their size becomes unreasonable). We call this the **Client-Assigned, Safe but Fat** approach.

An alternative delegates timestamping to a small number of DC servers [3, 17, 26]. This enables the use of small vectors, at the cost of losing some parallelism. However, this is not fault tolerant if the client does not reside in a DC failure domain. For instance, it may violate at-most-once delivery. Consider a client transmitting update u to be timestamped by DC_1. If it does not receive an acknowledgement, it retries, say with DC_2 (failover). This may result in u receiving two distinct timestamps, and being delivered twice. Duplicate delivery violates safety for many confluent types, or otherwise complicates their implementation [11, 26]. We call this the **Server-Assigned, Lean but Unsafe** approach.

Clearly, neither "fat" nor "unsafe" is satisfactory.

2.3 Causal consistency with partial replication is hard

Since a partial replica receives only a subset of the updates, and hence of metadata, it could miss some causal dependencies [8]. Consider the following example: Alice posts a photo on her wall in a social network application (update a). Bob sees the photo and mentions it in a message to Charles (update b), who in turn mentions it to David (update c). When David looks at Alice's wall, he expects to observe update a and view the photo. However, if David's machine does not cache Charles' inbox, it cannot observe the causal chain $\mathsf{a} \rightarrow \mathsf{b} \rightarrow \mathsf{c}$ and might incorrectly deliver c without a. Metadata design should protect from such causal gaps, caused by transitive dependency over absent objects.

Failures complicate the picture even more. Suppose David sees Alice's photo, and posts a comment to Alice's wall (update d). Now a failure occurs, and David's machine fails over to a new DC. Unfortunately, the new DC has not yet received Bob's update b, on which comment d causally depends. Therefore, it cannot deliver the comment, i.e., fulfill convergence, without violating causal consistency. David cannot read new objects from the DC for the same reason.[3]

Finally, a DC logs an individual update for only a limited amount of time, but clients may be unavailable for unlimited periods. Suppose that David's comment d is accepted by the DC, but David's machine disconnects before receiving the acknowledgement. Much later, after d has been executed and purged away, David's machine comes back, only to retry d. This could violate at-most-once delivery; some previous systems avoid this with fat version vectors [8, 28] or depend on client availability [23].

3. THE SWIFTCLOUD APPROACH

We now describe an abstract design that addresses the above challenges, first in the failure-free case, and next, how we support DC failure. Our design applies the principles of consistency algorithms for full replication systems to build a cloud-based support for partial client replicas.

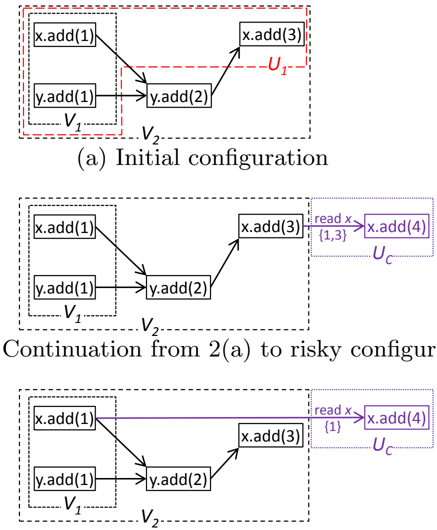
3.1 Causal consistency at full DC replicas

Ensuring causal consistency at fully-replicated DCs is a well-known problem [2, 17, 25, 26]. Our design is log-based, i.e., SwiftCloud stores updates in a log and transmits them incrementally; it includes optimisations, where the full log is occasionally replaced by the state of an object, called **checkpoint** [8, 29]. We discuss checkpoints only where relevant.

A **database version** is any subset of updates, noted U, ordered by causality. A version maps object identifiers to object state, by applying the relevant subsequence of the log; the value of an object is exposed via the read API.

We say that a version U has a **causal gap**, or is **inconsistent** if it is not causally-closed, i.e., if $\exists \mathsf{u}, \mathsf{u}' : \mathsf{u} \rightarrow \mathsf{u}' \wedge \mathsf{u} \notin U \wedge \mathsf{u}' \in U$. As we illustrate shortly, reading from an inconsistent version should be avoided, because, otherwise, subsequent accesses might violate causality. On the other hand, waiting for the gap to be filled would increase latency and decrease availability. To side-step this conundrum, we adopt

[3] Note that David can still perform updates, but they cannot be delivered, thus the system does not converge.



(a) Initial configuration

(b) Continuation from 2(a) to risky configuration

(c) Read-in-the-past: cont. from 2(a) to conservative configuration

Figure 2: Example evolution of configurations for two DCs, and a client. x and y are Sets; box = update; arrow = causal dependence (an optional text indicates the source of dependency); dashed box = named database version/state.

the approach of "reading in the past" [2, 25]. Thus, a DC exposes a gapless but possibly delayed state, noted V.

To illustrate, consider the example of Fig. 2(a). Objects x and y are of type Set. DC_1 is in state U_1 that includes version $V_1 \subseteq U_1$, and DC_2 in a later state V_2. Versions V_1 with value $[\mathsf{x} \mapsto \{1\}, \mathsf{y} \mapsto \{1\}]$ and V_2 with value $[\mathsf{x} \mapsto \{1,3\}, \mathsf{y} \mapsto \{1,2\}]$ are both gapless. However, version U_1, with value $[\mathsf{x} \mapsto \{1,3\}, y \mapsto \{1\}]$ has a gap, missing update y.add(2). When a client requests to read x at DC_1 in state U_1, the DC could return the most recent version, $\mathsf{x} : \{1,3\}$. However, if the application later requests y, to return a safe value of y requires to wait for the missing update from DC_2. By "reading in the past" instead, the same replica exposes the older but gapless version V_1, reading $\mathsf{x} : \{1\}$. Then, the second read will be satisfied immediately with $\mathsf{y} : \{1\}$. Once the missing update is received from DC_2, DC_1 may advance from version V_1 to V_2.

A gapless algorithm maintains a *causally-consistent, monotonically non-decreasing progression* of replica states [2]. Given an update u, let us note u.*deps* its set of causal predecessors, called its **dependency set**. If a full replica, in some consistent state V, receives u, and its dependencies are satisfied, i.e., u.*deps* $\subseteq V$, then it applies u. The new state is $V' = V \oplus \{\mathsf{u}\}$, where we note by $\oplus$ the **log merge operator** that unions the two logs, respecting causality (the operator filters out duplicates, as we discuss in §4.1). State V' is consistent, and monotonicity is respected, since $V \subseteq V'$.

If the dependencies are not met, as detected using metadata, the replica buffers u until the causal gap is filled.

3.2 Causal consistency at partial client replicas

As a client replica contains only part of the database and its metadata, this complicates consistency [8]. To relieve clients of this complexity, we move most of the burden of managing gapless versions to the DC-side full replica.

At any point in time, a client is interested in a subset of the objects in the database, called its **interest set**. Its initial state consists of the projection of some DC's state onto its interest set. This is a causally-consistent state, as shown in the previous section. Client state can change, either because of an update generated by the client itself, called an **internal update**, or because of one received from a DC, called **external**. An internal update obviously maintains causal consistency. If the same DC sends external updates with no gaps, then the client state remains causally consistent.

More formally, let $i \in \mathcal{DC}$ denote DC identifiers and $m \in \mathcal{C}$ client identifiers with $\mathcal{DC} \cap \mathcal{C} = \emptyset$. Where appropiate, we use simply DC and C to refer to some DC and client, respectively.

Consider some recent DC state, which we will call the **base version** of the client, noted V_{DC}. The interest set of client C is noted $O \subseteq x, y, \ldots$. The client state, noted V_C, is restricted to these objects. It consists of two parts. One is the projection of base version V_{DC} onto its interest set, noted $V_{DC}|_O$. The other is the log of internal updates, noted U_C. The client state is their merge $V_C = V_{DC}|_O \oplus U_C|_O$. On cache miss, the client adds the missing object to its interest set, and fetches the object from base version V_{DC}, thereby extending the projection.

Base version V_{DC} is a monotonically non-decreasing causal version (it might be slightly behind the actual current state of the DC due to propagation delays). By induction, internal updates can causally depend only on internal updates, or on updates taken from the base version. Therefore, a hypothetical full version $V_{DC} \oplus U_C$ would be causally consistent. Its projection is equivalent to the client state: $(V_{DC} \oplus U_C)|_O = V_{DC}|_O \oplus U_C|_O = V_C$.

This approach ensures conditional availability. Reads from a version in the cache are always available and guaranteed causally consistent, although possibly slightly stale. If a read misses in the cache, the DC returns a consistent version immediately. Furthermore, the client replica can *write fast*, because it commits updates without waiting, and transfers them to its DC in the background.

Convergence is ensured, because the client's base version and log are synchronised with the DC in the background.

3.3 Failover and causal dependencies

The approach described so far assumes that a client connects to a single DC. However, a client can switch to a new DC at any time, in particular in response to a failure. Although each DC's state is consistent, an update that is delivered to one is not necessarily delivered in the other (because geo-replication is asynchronous, to ensure availability and performance at the DC level), which could potentially create a causal gap in the client.

To illustrate the problem, return to the example of Fig. 2(a). Consider two DCs: DC_1 is in (consistent) state V_1, and DC_2 in (consistent) state V_2; DC_1 does not include two recent updates of V_2. Client C, connected to DC_2, replicates object x only; its state is $V_2|_{\{\mathsf{x}\}}$. Suppose that the client reads the Set $\mathsf{x} : \{1,3\}$, and performs update $\mathsf{u} = \mathsf{add}(4)$, transitioning to the configuration shown in Fig. 2(b).

If this client now fails over to DC_1, and the two DCs cannot communicate, the system is not live:
(1) *Reads are not available*: DC_1 cannot satisfy a request

for y, since the version read by the client is more recent than the DC_1 version, $V_2 \not\subseteq V_1$.

(2) *Updates cannot be delivered (divergence)*: DC_1 cannot deliver u, due to a missing dependency: $\mathsf{u}.deps \not\subseteq V_1$.

Therefore, DC_1 must reject the client (i.e., withhold its requests) to avoid creating the gap in state $V_1 \oplus U_C$.

3.3.1 Conservative read: possibly stale, but safe

SwiftCloud provides a solution to this problem. To avoid such gaps that cannot be satisfied, the insight is to depend only on **K-stable updates** that are likely to be present in the failover DC, similarly to Mahajan et al. [28].

A version V is K-stable if every one of its updates is replicated in at least K DCs, i.e., $|\{i \in \mathcal{DC} \mid V \subseteq V_i\}| \geq K$, where $K \geq 1$ is a threshold configured w.r.t. expected failure model. To this effect, each DC_i maintains a consistent **K-stable version** $V_i^K \subseteq V_i$, which contains the updates for which DC_i has received acknowledgements from at least $K-1$ distinct other DCs.

A client's base version must be K-stable, i.e., $V_C = V_{DC}^K|_O \oplus U_C|_O$, to support failover. In this way, the client depends, either on external updates that are likely to be found in any DC (V_{DC}^K), or internal ones, which the client can always transfer to the new DC (U_C).

To illustrate, let us return to Fig. 2(a), and consider the conservative progression to Fig. 2(c), assuming $K = 2$. The client's read of x returns the 2-stable version $\{1\}$, avoiding the dangerous dependency via an update on y. If DC_2 is unavailable, the client can fail over to DC_1, reading y and propagating its update remain both live.

By the same arguments as in §3.2, a DC version V_{DC}^K is causally consistent and monotonically non-decreasing, and hence the client's version as well. Note that a client observes its internal updates immediately, even if not K-stable.

Parameter K can be adjusted dynamically without impacting correctness. Decreasing it has immediate effect. Increasing K has effect only for future updates, to preserve montonicity.

3.3.2 Discussion

The source of the above problem is an transitive causal dependency on an update to the object that the client replica does not replicate ($\mathsf{y.add}(2)$ in our example). As this is an inherent issue, we conjecture a general impossibility result, stating that genuine partial replication, causal consistency, conditional availability and timely at-least-once delivery (convergence) are incompatible. Accordingly, the requirements must be relaxed.

Note that in many previous systems, this impossibility translates to a trade-off between consistency and availability on the one hand, and performance on the other [15, 25, 34] By "reading in the past," we displace this to a trade-off between freshness and availability, controlled by adjusting K. A higher K increases availability, but updates take longer to be delivered;[4] in the limit, $K = N$ ensures complete availability, but no client can deliver a new update when some DC is unavailable. A lower K improves freshness, but increases the probability that a client will not be able to fail over, and that it will block until its original DC recovers. In the limit, $K = 1$ is identical to the basic protocol from §3.2, and is similar to blocking session-guarantee protocols [35]. $K = 2$ is a good compromise for deployments with three or more DCs, as it covers common scenarios of a DC failure or disconnection [15, 22]. Our experimental evaluation shows that it incurs a negligible staleness.

Network partitions.

Client failover between DCs is safe and generally live, except when the original set of K DCs were partitioned away from both other DCs and the client, shortly after they delivered a version to the client. In this case, the client blocks. To side-step this unavoidable possibility, we provide an unsafe API to read inconsistent data.

When a set of fewer than K DCs is partitioned from other DCs, the clients that connect to them do not deliver their mutual updates until the partition heals. To improve liveness in this scenario, SwiftCloud supports two heuristics: *(i)* a partitioned DC announces its "isolated" status, automatically recommending clients to use another DC, and *(ii)* clients who cannot reach another DC that satisfies their dependencies can use the isolated DCs with K temporarily lowered, risking unavailability if another DC fails.

4. DETAILED DESIGN

We now describe a metadata and concrete protocols implementing the abstract design.

4.1 Metadata

The SwiftCloud approach requires metadata: *(1)* to *uniquely identify an update*; *(2)* to *encode its causal dependencies*; *(3)* to *identify* and *compare versions*; *(4)* and to *identify all the updates of a transaction.* We now describe a metadata design that fulfils the requirements and has a low cost. It combines the strengths of the two approaches outlined in Section 2.3, and is both *lean and safe.*

A timestamp is a pair $(j, k) \in (\mathcal{DC} \cup \mathcal{C}) \times \mathbb{N}$, where j identifies the node that assigned the timestamp (either a DC or a client) and k is a sequence number. Similarly to Ladin et al. [23], the metadata assigned to some update u combines: *(i)* a single **client-assigned timestamp** $\mathsf{u}.t_{\mathcal{C}}$ that uniquely identifies the update, and *(ii)* a set of zero or more **DC-assigned timestamps** $\mathsf{u}.T_{\mathcal{DC}}$. At the initial client, it has only the client timestamp; after delivery to a DC, it receives a DC timestamp; in case of DC failover, it my have several DC timestamps. Nodes can refer to an update via any of its timestamps in order to tolerate failures. The updates in a transaction all have the same timestamp(s), to ensure all-or-nothing delivery [34].

We represent a version or a dependency as a **version vector**. A vector is a partial map from node ID to integer timestamp. For instance, when u has dependency $VV = [DC_1 \mapsto 1, DC_2 \mapsto 2]$, this means that u causally depends on $\{(DC_1, 1), (DC_2, 1), (DC_2, 2)\}$. A vector has at most one client entry, and multiple DC entries; thus, its size is bounded by the number of DCs, limiting network overhead. In contrast to a dependence graph, a vector compactly represents transitive dependencies and can be evaluated locally by any node.

We note by $\mathcal{T}$ the function that maps a vector VV to the timestamps it represents:

$$\mathcal{T}(VV) = \{(j, k) \in \mathsf{dom}(VV) \times \mathbb{N} \mid k \leq VV(j)\}$$

Function $\mathcal{V}$ maps vector VV and state U to a database ver-

[4] The increased number of concurrent updates that this causes is not a big problem, thanks to confluent types.

sion, i.e., the set of updates in U that match the vector:

$$\mathcal{V}(VV, U) = \{\mathsf{u} \in U \mid (\mathsf{u}.T_{\mathcal{DC}} \cup \{\mathsf{u}.t_{\mathcal{C}}\}) \cap \mathcal{T}(VV) \neq \emptyset\}$$

($\mathcal{V}$ is defined for states U that cover all timestamps of VV)

Note that, for $\mathcal{V}$, an update can be identified by any of its timestamps equivalently and that $\mathcal{V}$ is monotonic with growing state U.

The log merge operator $U_1 \oplus U_2$ eliminates duplicates based on client timestamps: two updates $\mathsf{u}_1 \in U_1, \mathsf{u}_2 \in U_2$ are identical if $\mathsf{u}_1.t_{\mathcal{C}} = \mathsf{u}_2.t_{\mathcal{C}}$. The merge operator merges their DC timestamps into $\mathsf{u} \in U_1 \oplus U_2$, such that $\mathsf{u}.T_{\mathcal{DC}} = \mathsf{u}_1.T_{\mathcal{DC}} \cup \mathsf{u}_2.T_{\mathcal{DC}}$.

4.2 Protocols

We now describe the protocols of SwiftCloud by following the lifetime of an update, and with reference to Fig. 1.

State.

The state of a DC U_{DC} consists of a set of object versions maintained in durable storage. Each version individually respects per-object causality and atomicity of transactions. A DC also maintains a vector VV_{DC} that represents a recent database version that preserves atomicity and causal consistency across objects. This state is noted $V_{DC} = \mathcal{V}(VV_{DC}, U_{DC})$.

In addition to its own updates U_C (its commit log), a client stores the state received from its DC, and the corresponding vector. The former is the DC's base version, projected to the client's interest set O, $V_{DC}|_O$ (§3.2). The latter is a copy of the DC's vector VV_{DC} that describes the base version.

Client-side execution.

When an application starts a transaction τ at client C, the client initialises it with an empty buffer of updates $\tau.U \leftarrow \emptyset$ and a **snapshot vector** of the current base version $\tau.depsVV \leftarrow VV_{DC}$; the DC can update the client's base version concurrently with the transaction execution. A read in transaction τ is answered from the version identified by the snapshot vector, merged with its internal updates, $\tau.V = \mathcal{V}(\tau.depsVV, V_{DC}|_O) \oplus U_C|_O \oplus \tau.U$. If the requested object is not in the client's interest set, $x \notin O$, the client extends its interest set, and returns the value once the DC updates the base version projection.

When the application issues internal update u, it is appended to the transaction buffer $\tau.U \leftarrow \tau.U \oplus \{\mathsf{u}\}$, and included in any later `read`. To simplify the notation, in this section a transaction consists of at most one update. This can be easily extended to multiple updates, by assigning the same timestamp to all the updates of the same transaction, ensuring the all-or-nothing property [34].

The transaction commits locally at the client and never fails. If the transaction made an update $\mathsf{u} \in \tau.U$, the client replica commits it locally as follows: *(1)* Assign it client timestamp $\mathsf{u}.t_{\mathcal{C}} = (C, k)$, where k counts the number of update transactions at the client; *(2)* assign it a **dependency vector** initialised with the transaction snapshot vector $\mathsf{u}.depsVV = \tau.depsVV$; *(3)* append it to the commit log of local updates on stable storage $U_C \leftarrow U_C \oplus \{\mathsf{u}\}$. This terminates the transaction; the client can start a new one, which will observe the above update.

Transfer protocol: Client to DC.

The transfer protocol transmits committed updates from a client to its current DC, in the background. It repeatedly picks the first unacknowledged committed update u from the log. If any of u's internal dependencies has recently been assigned a DC timestamp, it merges this timestamp into the dependency vector. Then, the client sends a copy of u to its current DC. The client expects an acknowledgement from the DC, containing the timestamp(s) T that the DC assigned to update u. The client records the timestamps in the original update record $\mathsf{u}.T_{\mathcal{DC}} \leftarrow T$. In the failure-free case, T is a singleton.

A transfer request may fail for three reasons:

(a) Timeout: the DC is suspected unavailable; the client connects to another DC (failover) and repeats the protocol.

(b) The DC reports a *missing internal dependency*, i.e., it has not received some previous update of the client, as a result of a prior failover. The client recovers by marking as unacknowledged all internal updates starting from the oldest missing dependency, and restarting the transfer protocol from that point.

(c) The DC reports a *missing external dependency*; this is also an effect of failover. In this case, the client tries yet another DC. The approach from §3.3.1 avoids this failing repeatedly.

Upon receiving update u, the DC verifies if its dependencies are satisfied, i.e., if $\mathcal{T}(\mathsf{u}.depsVV) \subseteq \mathcal{T}(VV_{DC})$. (If this check fails, the DC accordingly reports error (b) or (c) to the client.) If the DC has not received this update previously (i.e., if its client timestamp is new), the DC performs the following steps: *(1)* Assign it a DC timestamp $\mathsf{u}.T_{\mathcal{DC}} \leftarrow \{(DC, VV_{DC}(DC) + 1))\}$, *(2)* store it durably $U_{DC} \leftarrow U_{DC} \oplus \{\mathsf{u}\}$, *(3)* incorporate its DC timestamp into VV_{DC}, in order to make the update visible in the DC version V_{DC}. These steps may be interleaved and/or batched with transfer requests from different clients. Only Step 3 needs to be linearisable.

If the update has been received before, the DC looks up its previously-assigned DC timestamps. In both cases, the DC acknowledges the transfer to the client with the DC timestamp(s).

Geo-replication protocol: DC to DC.

When a DC accepts a fresh transaction in the transfer protocol, its updates become available to geo-replication, and the accepting DC sends them to all other DCs using uniform reliable broadcast. A DC that receives a broadcast message containing update u does the following: (1) If the dependencies of u are not met, i.e., if $\mathcal{T}(\mathsf{u}.depsVV) \not\subseteq \mathcal{T}(VV_{DC})$, buffer it until they are; (2) incorporate u into durable state $U_{DC} \leftarrow U_{DC} \oplus \{\mathsf{u}\}$ (if u is not fresh, the duplicate-resilient log merge safely unions all timestamps); and (3) incorporate its DC timestamp(s) into the DC version vector VV_{DC}. This last step makes it available to the notification protocol. The K-stable version V_{DC}^{K} is maintained similarly.

Notification protocol: DC to Client.

A DC maintains a best-effort notification session, over a FIFO channel, to each of its connected clients. The soft state of a session includes a copy of the client's interest set O and the last known base version vector used by the client, VV'_{DC}. The DC accepts a new session only if its own state is consistent with the base version provided by the client, i.e., if $\mathcal{T}(VV'_{DC}) \subseteq \mathcal{T}(VV_{DC})$. Otherwise, the client is redirected

	YCSB [14]	**SocialApp** [34]
Type of objects	LWW Map	Set, Counter, Register
Object payload	10 × 100 bytes	variable
Read txns	read fields (A: 50% / B: 95%)	read wall (80%) see friends (8%)
Update txns	update field (A: 50% / B:5%)	message (5%) post status (5%) add friend (2%)
Objects / txn	1 (non-txnal)	2–5
Database size	50,000 objects	400,000 objects
Object popularity	uniform / Zipfian	uniform
Session locality	40% (low) / 80% (high)	

Table 1: Characteristics of applications/workloads.

to another DC, since the DC would cause a causal gap with the client's state (see §3.3.1).

The DC sends over each channel a causal stream of update notifications.[5] Notifications are batched according to either time or to rate [8]. A notification packet consists of a new base version vector VV_{DC}, and a log of all the updates U_δ to the objects of the interest set, between the client's previous base vector VV'_{DC} and the new one. Formally, $U_\delta = \{\mathsf{u} \in U_{DC}|_O \mid \mathsf{u}.\mathcal{T}_{\mathcal{DC}} \cap (\mathcal{T}(VV_{DC}) \setminus \mathcal{T}(VV'_{DC})) \neq \emptyset\}$. The client applies the newly-received updates to its local state described by the old base version, $V_{DC}|_O \leftarrow V_{DC}|_O \oplus U_\delta$, and assumes the new vector VV_{DC}. If any of received updates is a duplicate, the log merge operator discards it safely.

When the client detects a broken channel, it reinitiates the session, possibly on a new DC.

The interest set can change dynamically. When an object is evicted from the cache, the notifications are lazily unsubscribed to save resources. When it is extended with object x, the DC responds with the version of x that includes all updates to x up to the base version vector. To avoid races, a notification includes a hash of the interest set, which the client checks.

4.3 Object checkpoints and log pruning

Update logs consume substantial storage and, to smaller extent, network. To avoid unbounded growth, a **pruning protocol** periodically replaces the prefix of a log by a *checkpoint*. In the common case, a checkpoint is more compact than the corresponding log of updates; for instance, a log containing one thousand increments to a Counter object, each with its timestamps, can be replaced by a checkpoint containing just the number 1000 paired with a version vector.

4.3.1 Log pruning in the DC

The log at a DC provides *(a)* protection from duplicate update delivery, as abstracted by the $\oplus$ operator, and *(b)* the capability to compute different versions, for application processes reading at different causal times. A log entry for update u may be replaced with a checkpoint once its duplicates have been filtered out, and once u has been delivered to all interested application processes.

Precise evaluation of this condition would require access to the client replica states. In practice, we need to *prune aggressively*, but without violating correctness. In order to reduce the risk of pruning a version not yet delivered to an interested application (which could force it to restart an ongoing transaction), we prune only a **delayed version** VV^{Δ}_{DC}, where Δ is a real-time delay [25, 26].

To avoid duplicates despite log pruning, we extend DC metadata as follows. DC_i maintains an **at-most-once guard** $G_i : \mathcal{C} \to \mathbb{N}$, which records the sequence number of each client's last pruned update. The guard is local to a DC. Whenever the DC receives a transfer request or a geo-replication message for update u with client timestamp (C, k) that is not in its log, it checks if $G_i(C) \geq k$. If so the update is a duplicate of a pruned update, which the DC ignores, nonetheless incorporating all of u's DC timestamps in its version vector; and, if it was a transfer request, the DC replies with a vector VV_i, which is an overapproximation of the (discarded) set of u's DC timestamps.

The notification protocol also uses checkpoints. On a client cache miss, instead of a complete log, the DC sends an equivalent checkpoint of the object, together with the client's guard entry, so that the client can merge it with its log safely.

4.3.2 Pruning the client's log

Managing the log at a client is simpler. A client logs its own updates U_C, which may include updates to objects that are currently outside of its interest set. This enables the client to read its own updates, and to propagate them lazily to a DC when connected and convenient. An update u can be discarded as soon as it appears in the K-stable base version V^K_{DC}, i.e., when the client becomes dependent on the presence of u at a DC.

5. EVALUATION

We implement SwiftCloud and evaluate it experimentally, in comparison to alternatives. We show that SwiftCloud provides: *(i)* fast response, under 1 ms for both reads and writes to cached objects (§5.3); *(ii)* while supporting thousands of clients, throughput that scales with the number of DCs and small metadata, sized linear in the number of DCs (§5.4); *(iii)* fault-tolerance w.r.t. client churn (§5.5) and DC outages (§5.6); and *(iv)* low staleness, under 1% of stale reads under common conditions (§5.7).

5.1 Implementation and applications

SwiftCloud and its applications are implemented in Java.[6] SwiftCloud uses an extendable library of log-based CRDT types [33], in-memory storage, Kryo for data marshalling, and a custom RPC implementation. A client cache has a fixed size and uses an LRU eviction policy.

Our client API resembles both production object stores, such as Riak 2.0 or Redis [1, 30], and prototype causal transactional stores, such as COPS or Eiger [25, 26]:

begin_transaction()
commit_transaction()
	read(object) : value
	update(object, method(args . . .))

The actual API also includes caching options omitted here.

Along the lines of previous studies of causally-consistent systems [3, 5, 26, 34], we use two different benchmarks, YCSB and SocialApp, summarised in Tab. 1.

YCSB [14] serves as a kind of micro-benchmark, with simple requirements, measuring baseline costs and specific system properties in isolation. It has a simple key-field-value

[5] Alternatively, the client can ask for invalidations instead, trading responsiveness for lower bandwidth utilization and higher DC throughput.

[6] https://github.com/SyncFree/SwiftCloud

object model, which we implement as a LWW Map type, using a default payload of ten fields of 100 bytes each. YCSB issues single-object reads and writes. We use two of the standard YCSB workloads: update-heavy Workload A, and read-dominated Workload B. The object access pattern can be set to either uniform or Zipfian. YCSB does not rely on transactional semantics or high-level data types.

SocialApp is a social network application modelled after WaltSocial [34]. It employs high-level data types such as Sets, for friends and posts, LWW Register for profile information, Counter for counting profile visits, and object references. SocialApp accesses multiple objects in a causal transaction to ensure that operations such as reading a wall page and profile information behave consistently. The SocialApp workload is read-dominated, but visiting a wall actually increments the wall visit counter. The user popularity distribution is uniform.

In order to model the locality behaviour of a client, both YCSB and SocialApp are augmented with a facility to control access locality, mimicking social network access patterns [9]. Within a client session, a workload generator draws uniformly from a pool of session-specific objects with either 40% (*low locality*) or 80% (*high locality*) probability. Objects not drawn from this local pool are drawn from the global distribution (uniform or Zipfian) described above. The local pool can fit in the client's cache.

5.2 Experimental setup

We run three DCs in geographically distributed Amazon EC2 availability zones (Europe, Virginia, and Oregon), and a pool of distributed clients. Round-Trip Times (RTTs) between nodes are as follows:

	Oregon DC	Virginia DC	Europe DC
nearby clients	60–80 ms	60–80 ms	60–80 ms
Europe DC	177 ms	80 ms	
Virginia DC	60 ms		

Each DC runs on a single m3.m EC2 instance, cheap virtual hardware, equivalent to a single core 64-bit 2.0 GHz Intel Xeon processor (2 ECUs) with 3.75 GB of RAM, and OpenJDK7 on Linux 3.2. Objects are pruned at random intervals between 60–120 s, to avoid bursts of pruning activity. We deploy 500–2,500 clients on a separate pool of 90 m3.m EC2 instances. Clients load DCs uniformly and use the closest DC by default, with a client-DC RTT ranging in 60–80 ms.

For comparison, we provide three protocol modes: *(i) SwiftCloud mode* (default) with client cache replicas of 256 objects, refreshed with notifications at a rate ≤1 s by default; *(ii) Safe But Fat metadata mode* with cache, but with client-assigned metadata only (modelled after PRACTI, or Depot without cryptography [8, 28]), *(iii) server-side replication mode* without client caches; in this mode, an update incurs two RTTs to a DC, modelling the cost of a synchronous write to a quorum of servers to ensure fault-tolerance comparable to SwiftCloud.

5.3 Response time and throughput

We run several experiments to compare SwiftCloud's client-side caching, with reference to the locality potential and server-side geo-replication without caching.

Fig. 3 shows response times for YCSB, comparing server-only (left side) with client replication (right side), under low (top) and high locality (bottom), when the system is not overloaded. Recall that in server-only replication, a read incurs a RTT to the DC, whereas an update incurs 2 RTTs.

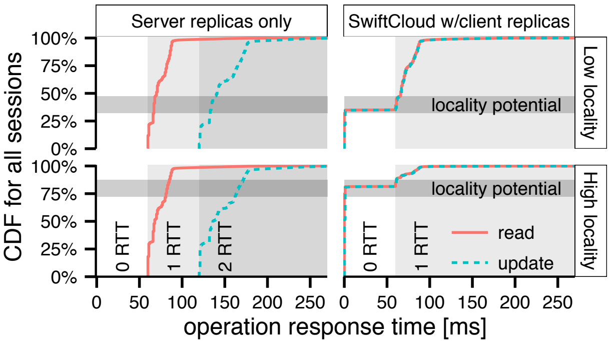


Figure 3: Response time for YCSB operations (workload A, Zipfian object popularity) under different system and workload locality configurations, aggregated for all clients.

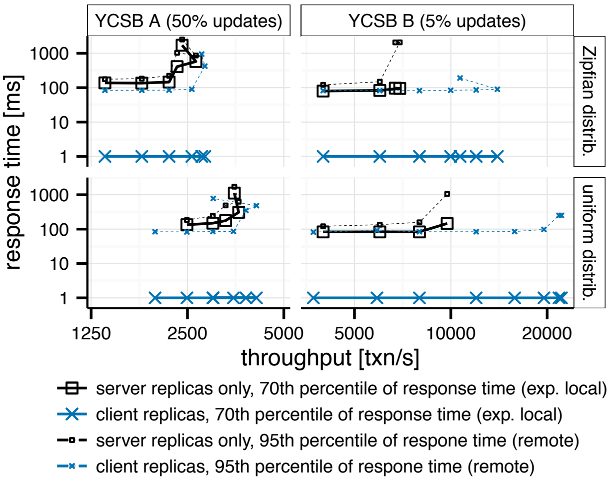


Figure 4: Throughput vs. response time for different system configurations running variants of YCSB.

We expect SwiftCloud to provide much faster response, at least for cached data. Indeed, the figure shows that a significant fraction of operations respond immediately in SwiftCloud mode, and this fraction tracks the locality of the workload (marked "locality potential" on the figure), within a ±7.5 percentage-point margin attributable to caching policy artefacts. The remaining operations require one round-trip to the DC, indicated as 1 RTT. As our measurements for SocialApp show the same message, we do not report them here. These results demonstrate that the consistency guarantees and the rich programming interface of SwiftCloud do *not* affect responsiveness of read and update of cached data.

In terms of throughput, client-side replication is a mixed blessing: it lets client replicas absorb read requests that would otherwise reach the DC, but also puts extra load of maintaining client replicas on DCs. In another experiment (not plotted), we saturate the system to determine its maximum throughput. SwiftCloud's client-side replication consistently improves throughput for high-locality workloads, by 7% up to 128%. It is especially beneficial to read-heavy workloads. In contrast, low-locality workloads show no clear trend; depending on the workload, throughput either increases by up to 38%, or decrease by up to 11%.

Our next experiment studies how response times vary with

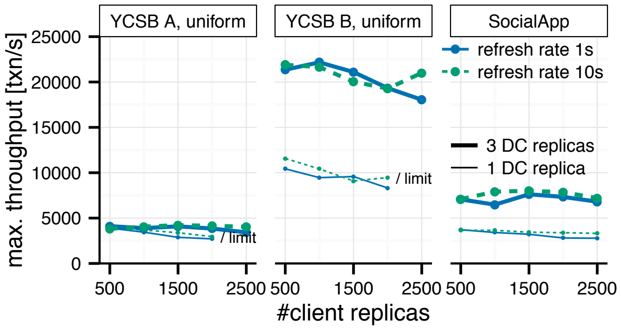


Figure 5: Maximum system throughput for a variable number of client and DC replicas.

server load and with the staleness settings. The results show that, as expected, cached objects respond immediately and are always available, but the responsiveness of cache misses depends on server load. For this study, Fig. 4 plots throughput vs. response time, for YCSB A (left side) and B (right side), both for the Zipfian (top) and uniform (bottom) distributions. Each point represents the aggregated throughput and latency for a given transaction incoming rate, which we increase until reaching the saturation point. The curves report two percentiles of response time: the lower (70th percentile) line represents the response time for requests that hit in the cache (the session locality level is 80%), whereas the higher (95th percentile) line represents misses, i.e., requests served by a DC.

As expected, the lower (cached) percentile consistently outperforms the server-side baseline, for all workloads and transaction rates. A separate analysis, not reported in detail here, reveals that a saturated DC slows down its rate of notifications, increasing staleness, but this does not impact response time, as desired. In contrast, the higher percentile follows the trend of server-side replication response time, increasing remote access time.

Varying the target notification rate (not plotted) between 500 ms and 1000 ms, reveals the same trend: response time is not affected by the increased staleness. At a lower refresh rate, notification batches are less frequent but larger. This increases throughput for the update-heavy YCSB A (up to tens of percent points), but has no effect on the throughput of read-heavy YCSB B. We expect the impact of refresh rate to be amplified for workloads with smaller rate of notification updates.

5.4 Scalability

Next, we measure how well SwiftCloud scales with increasing numbers of DC and of client replicas. Of course, performance is expected to increase with more DCs, but most importantly, the size of metadata should be small, should not depend on the number of clients, and should increase linearly with the number of DCs. Our results confirm these expectations.

In this experiment, we run SwiftCloud with a variable number of client (500–2500) and server (1–3) replicas. We report only on the uniform object distribution, because, under zipfian distribution, it is challenging to maintain a constant total workload while varying the number of clients. To control staleness, we run SwiftCloud with two different notification rates (every 1 s and every 10 s).

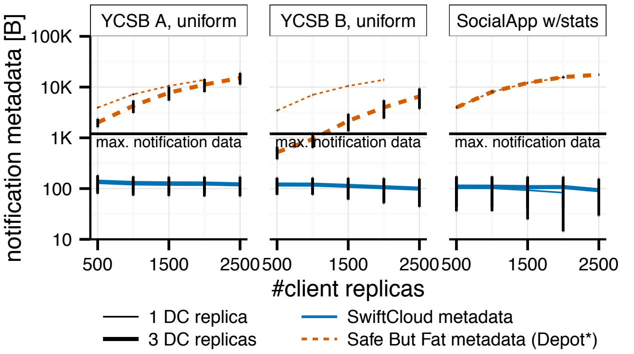


Figure 6: Size of metadata in notification message for a variable number of replicas, mean and standard error. Normalised to a notification of 10 updates.

Fig. 5 shows the maximum system throughput on the Y axis, increasing the number of replicas along the X axis. The thin lines are for a single DC, the bold ones for three DCs. Solid lines represent the fast notification rate, dashed lines the slow one. The figure shows, left to right, YCSB Workload A, YCSB Workload B, and SocialApp.

The capacity of a single DC in our hardware configuration peaks at 2,000 active client replicas for YCSB, and 2,500 for SocialApp. Beyond that, the DC drops sessions.

As to be expected, additional DC replicas increase the system capacity for operations that can be performed at only one replica such as read operations or sending notification messages. Whereas a single SwiftCloud DC supports at most 2,000 clients. With three DCs SwiftCloud supports at least 2,500 clients for all workloads. Unfortunately, as we ran out of resources for client machines at this point, we cannot report an upper bound.

For some fixed number of DCs, adding client replicas increases the aggregated system throughput, until a point where the cost of maintaining client replicas up to date saturates the DCs, and further clients do not absorb enough reads to overcome that cost. Note that the lower refresh rate can control the load at a DC, and reduces it by 5 to 15%.

In the same experiment, Fig. 6 presents the distribution of metadata size in notification messages. (Notifications are the most common and the most costly messages sent over the network.) We plot the size of metadata (in bytes) on the Y axis, varying the number of clients along the X axis. Left to right, the same workloads as in the previous figure. Thin lines are for one DC, thick lines for three DCs. A solid line represents SwiftCloud “Lean and Safe” metadata, and dotted lines the classical “Safe But Fat” approach. Note that our Safe-but-Fat implementation includes the optimisation of sending modified entries of vector rather than the full vector, as in Depot or PRACTI [8, 28]. Vertical bars represent standard error across clients. As notifications are batched, we normalise metadata size to a message carrying exactly 10 updates, corresponding to under approx. 1 KB of data.

This plot confirms that the SwiftCloud metadata is small and constant, at 100–150 bytes/notification (10–15 bytes per update); data plus metadata together fit inside a single standard network packet. It is *independent* both from the number of client replicas and from the workload. Increasing the number of DC replicas from one to three causes a negligible increase in metadata size, of under 10 bytes.

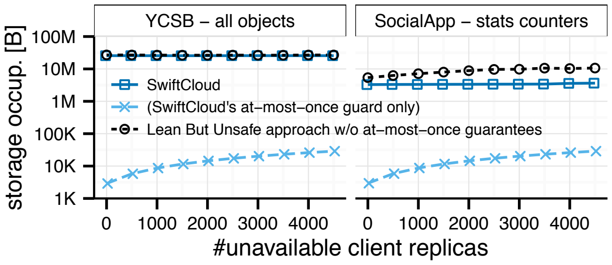


Figure 7: Storage occupation at one DC in reaction to client churn, for SwiftCloud and Lean-but-Unsafe alternative.

In contrast, the classical Safe-but-Fat metadata grows linearly with the number of clients and exhibits higher variability. Its size reaches approx. 1 KB for 1,000 clients in all workloads, and 10 KB for 2,500 clients. Clearly, metadata that is up to 10× larger than the actual data represents a substantial overhead.

5.5 Tolerating client churn

We now turn to fault tolerance. In the next experiment, we evaluate SwiftCloud under client churn, by periodically disconnecting client replicas and replacing them with a new set of clients. At any point in time, there are 500 active clients and a variable number of disconnected clients, up to 5000. Fig. 7 illustrates the storage occupation of a DC for representative workloads, which is also a proxy for the size of object checkpoints transferred. We compare SwiftCloud's log compaction to a protocol without at-most-once delivery guarantees (Lean But Unsafe).

SwiftCloud storage size is approximately constant thanks to the aggressive log compaction. This is safe thanks to the at-most-once guard table per DC. Although the size of the guard (bottom curve) grows with the number of clients, it requires orders of magnitude less storage than the actual database itself.

A protocol without at-most-once delivery guarantees uses Lean-but-Unsafe metadata, without SwiftCloud's at-most-once guard. This requires more complexity in each object's implementation, to protect itself from duplicates. This increases the size of objects, impacting both storage and network costs. As is visible in the figure, the cost depends on the object type: none for YCSB's LWW-Map, which is naturally idempotent, vs. linear in the number of clients for SocialApp's Counter objects.

We conclude that the cost of maintaining SwiftCloud's at-most-once guard is negligible, and easily amortised by its stable behaviour and possible savings.

5.6 Tolerating DC failures

The next experiment studies the behaviour of SwiftCloud when a DC disconnects. The scatterplot in Fig. 8 shows the response time of a SocialApp client application as the client switches between DCs. Starting with a cold cache, response times quickly drops to near zero for transactions hitting in the cache, and to around 110 ms for misses. Some 33 s into the experiment, the current DC disconnects, and the client is diverted to another DC in a different continent. Thanks to K-stability the failover succeeds, and the client continues with the new DC. Response time for cache misses reflects the higher RTT to the new DC. At 64 s, the client switches back the initial DC, and performance smoothly recovers.

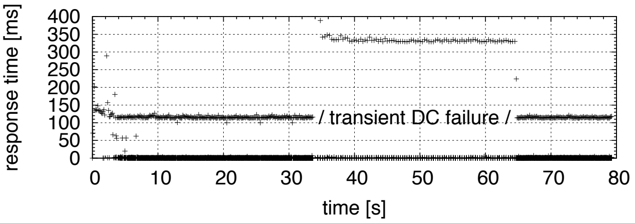


Figure 8: Response time for a client that hands over between DCs during a 30 s failure of a DC.

Recall that a server-side geo-replication system with similar fault-tolerance incurs high response time (cf. §5.3, or Reference 15) and does not ensure at-most-once delivery.

5.7 Staleness cost

The price to pay for our read-in-the-past approach is an increase in staleness. We consider a read *stale* if there exists, at the client's current DC, a non-K-stable version that is more recent than the one returned. A transaction is stale if any of its reads is stale. In the experiments so far, we observed a negligible number of stale reads and transactions, below 1 %. In another experiment (not plotted here), we artificially increase the probability of staleness by various means, e.g., using a smaller database, and setting cache size to zero. Even in this case, stale reads and stale transactions remain under 1% and 2.5% respectively.

6. RELATED WORK

6.1 Related results on consistency and availability

Mahajan et al. [27] and Attiya et al. [4] prove that no stronger consistency model than causal consistency is available and convergent, under *full* replication. We conjecture that these properties are not simultaneously achievable under *partial* replication, and we show how to weaken one of the liveness properties. Bailis et al. [6] also study variants of weak consistency models, and formulate a similar impossibility for a client switching server replicas. However, they do not take into account the capabilities of a client replica, as we do.

Some operations or objects of application may require stronger consistency, which requires synchronous protocols [20]. Prior work demonstrates that combining strong and weak consistency is possible [24, 34]. In particular, Balegas et al. [7] implemented protocols that enforce strong consistency invariants as a middleware layer on top of SwiftCloud.

6.2 Comparison with other systems

Several systems support consistent, available and convergent data access, at different scales.

6.2.1 Replicated databases for client-side apps

PRACTI [8] is a seminal work on causal consistency under partial replication. PRACTI uses Safe-but-Fat client-assigned metadata and a flexible log-exchange protocol that supports arbitrary communication topologies and modes. While PRACTI is very general, it is not viable for large-scale client-side replication deployment: *(i)* Its fat metadata approach (version vectors sized as the number of clients) is prohibitively expensive (see Fig. 6), and *(ii)* any replica can make another unavailable, because of the transitive dependence issue discussed in §3.3.2.

Our high availability techniques are similar to Depot [28], a causally-consistent storage for the client-side, built on top of untrusted cloud replicas. Depot tolerates Byzantine cloud behaviour using cryptographic metadata signatures, in order to detect misbehaviour, and fat metadata, in order to support direct client-to-client communication. Conservatively, Depot either exposes updates signed by K different servers or forces clients to receive all transitive dependencies of their reads. This is at odds with genuine partial replication [31]. Under no failures, a client receives metadata of *every* update; under failures, it may also receive their body. In contrast, SwiftCloud relies on DCs to compute K-stable consistent versions with lean metadata. In the event of an extensive failure involving K DCs, SwiftCloud provides the flexibility to decrease K dynamically or to weaken consistency.

Both PRACTI and Depot systems use Safe-but-Fat metadata. They support only LWW registers, but their rich metadata could conceivably accommodate high-level CRDTs too.

Lazy Replication (LR) protocols [23] support multiple consistency modes for client-side apps executing operations on server replicas. Under causal consistency, LR provides high availability with asynchronous read and write requests to multiple servers. As suggested by Ladin et al. [23], LR could also read stable updates for availability on failover, but that would force its clients to execute updates synchronously. The implementation of LR uses safe and lean metadata similar to SwiftCloud, involving client- and server-assigned timestamps together with vector summaries. A log compaction protocol relies on availability of client replicas and loosely-synchronised clocks for progress.

SwiftCloud structures the database into smaller CRDT objects, which allows it to provide partial client replicas, whereas LR considers only global operations. We show that client replicas can offer higher responsiveness on cached objects, instead of directing all operations to the server side as in LR, and that local updates can be combined with K-stable updates into a consistent view, avoiding slow and unavailable synchronous updates of LR. The log compaction technique of LR is complementary to ours, and optimises for the average case. SwiftCloud's aggressive pruning relies on at-most-once guard table, optimising for failure scenarios.

Recent web and mobile application frameworks, such as TouchDevelop [10], Google Drive Realtime API [12], or Mobius [13] support replication for in-browser or mobile applications. These systems are designed for small objects [12], database that fits on a mobile device [10], or a database of independent objects [13]. It is unknown if/how they support multiple DCs and fault tolerance, whereas SwiftCloud supports consistency, a large database, and fault tolerance. TouchDevelop provides a form of object composition, and offers integration with strong consistency [10]. We are looking into ways of adapting similar mechanisms.

6.2.2 Server-side geo-replicated databases

A number of geo-replicated systems offer available causally consistent data access inside a DC with excellent scale-out by sharding [3, 5, 17, 18, 25, 26].

Server-side geo-replication systems use variety of types of metadata, mostly Lean-but-Unsafe metadata. COPS [25] assigns metadata directly at database clients, and uses explicit dependencies (a graph). Follow-up work shows that this approach is costly, and assigns metadata at object/ shard replicas instead [17, 26], or on a designated node in the DC [3, 34]. The location of assignment directly impacts the size of causality metadata. In most systems, it varies with the number of reads, with the number of dependencies, and with the stability conditions in the system. When fewer nodes assign metadata, it tends to be smaller (as in SwiftCloud), but this may limit throughput. Recent work of Du et al. [18] make use of full stability, a special case of K-stability, to remove the need for dependency metadata in messages, thereby improving throughput.

Server-side designs do not easily extend beyond the scope and the failure domain of a DC, because *(i)* their protocols do not tolerate external client failures and DC outages, either blocking or violating safety (due to inadequate metadata, and the causal dependence issue); *(ii)* as they assume that data is updated by overwriting, implementing high-level confluent data types that work on the client-side is complex and costly (see Fig. 7); *(iii)* their metadata can grow with database size.

SwiftCloud's support for server-side sharding is limited compared to the most scalable geo-replicated designs. Reconciling client-side replication with a more decentralised sharding support, and small metadata size, is future work.

7. CONCLUSION

We presented the design of SwiftCloud, the first object database that offers client-side apps a local access to partial replica with the guarantees of geo-replicated systems.

Our experiments show that the design of SwiftCloud is able to provide immediate and consistent response for reads and updates on local objects, and to maintain the throughput of a server-side geo-replication, or better. SwiftCloud's metadata allows it to scale safely to thousands of clients with 3 DCs, with small size objects, and metadata at the level of 15 bytes per update, independent of the number of connected and disconnected clients. Our fault-tolerant protocols handle failures nearly transparently, at a low staleness cost.

SwiftCloud's design leverages a common principle that helps to achieve several goals: client buffering and controlled staleness can absorb the cost of scalability, availability, and consistency.

Acknowledgments.

Thanks to Carlos Baquero, Peter Bailis, Allen Clement, Alexey Gotsman, Masoud Saeida Ardekani, João Leitão, Vivien Quéma, Luís Rodrigues, and the anonymous reviewers, for their constructive comments. This research is supported in part by ANR (France) project ConcoRDanT (ANR-10-BLAN 0208), by the Google Europe Fellowship 2010 awarded to Marek Zawirski, by a Google Faculty Research Award 2013, by European FP7 project SyncFree (609 551, 2013–2016), and by NOVA LINCS (FCT UID/CEC/04516/2013, Portugal).

References

[1] Introducing Riak 2.0: Data types, strong consistency, full-text search, and much more, Oct. 2013. URL http://basho.com/introducing-riak-2-0/.

[2] M. Ahamad, G. Neiger, J. E. Burns, P. Kohli, and P. W. Hutto. Causal memory: definitions, implementation, and programming. *Distributed Computing*, 9(1):37–49, Mar. 1995.

[3] S. Almeida, J. Leitão, and L. Rodrigues. ChainReaction: a causal+ consistent datastore based on Chain Replication. In *Euro. Conf. on Comp. Sys. (EuroSys)*, Apr. 2013.

[4] H. Attiya, F. Ellen, and A. Morrison. Limitations of highly-available eventually-consistent data stores. In *Symp. on Principles of Dist. Comp. (PODC)*, pages 385–394. ACM, 2015.

[5] P. Bailis, A. Ghodsi, J. M. Hellerstein, and I. Stoica. Bolt-on causal consistency. In *Int. Conf. on the Mgt. of Data (SIGMOD)*, pages 761–772, New York, NY, USA, 2013.

[6] P. Bailis, A. Davidson, A. Fekete, A. Ghodsi, J. M. Hellerstein, and I. Stoica. Highly Available Transactions: Virtues and limitations. In *Int. Conf. on Very Large Data Bases (VLDB)*, Riva del Garda, Trento, Italy, 2014.

[7] V. Balegas, N. Preguiça, R. Rodrigues, S. Duarte, C. Ferreira, M. Najafzadeh, and M. Shapiro. Putting the consistency back into eventual consistency. In *Euro. Conf. on Comp. Sys. (EuroSys)*, Bordeaux, France, Apr. 2015.

[8] N. Belaramani, M. Dahlin, L. Gao, A. Nayate, A. Venkataramani, P. Yalagandula, and J. Zheng. PRACTI replication. In *Networked Sys. Design and Implem. (NSDI)*, pages 59–72, San Jose, CA, USA, May 2006. Usenix.

[9] F. Benevenuto, T. Rodrigues, M. Cha, and V. Almeida. Characterizing user behavior in online social networks. In *Internet Measurement Conference (IMC)*, 2009.

[10] S. Burckhardt. Bringing TouchDevelop to the cloud. Inside Microsoft Research Blog, Oct. 2013. URL http://blogs.technet.com/b/inside_microsoft_research/archive/2013/10/28/bringing-touchdevelop-to-the-cloud.aspx.

[11] S. Burckhardt, A. Gotsman, H. Yang, and M. Zawirski. Replicated data types: Specification, verification, optimality. In *Symp. on Principles of Prog. Lang. (POPL)*, pages 271–284, San Diego, CA, USA, Jan. 2014.

[12] B. Cairns. Build collaborative apps with Google Drive Realtime API. Google Apps Developers Blog, Mar. 2013. URL http://googleappsdeveloper.blogspot.com/2013/03/build-collaborative-apps-with-google.html.

[13] B.-G. Chun, C. Curino, R. Sears, A. Shraer, S. Madden, and R. Ramakrishnan. Mobius: Unified messaging and data serving for mobile apps. In *Int. Conf. on Mobile Sys., Apps. and Services (MobiSys)*, pages 141–154, New York, NY, USA, 2012.

[14] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with YCSB. In *Symp. on Cloud Computing*, pages 143–154, Indianapolis, IN, USA, 2010.

[15] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford. Spanner: Google's globally-distributed database. In *Symp. on Op. Sys. Design and Implementation (OSDI)*, pages 251–264, Hollywood, CA, USA, Oct. 2012. Usenix.

[16] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. In *Symp. on Op. Sys. Principles (SOSP)*, volume 41 of *Operating Systems Review*, pages 205–220, Stevenson, Washington, USA, Oct. 2007. Assoc. for Computing Machinery.

[17] J. Du, S. Elnikety, A. Roy, and W. Zwaenepoel. Orbe: Scalable causal consistency using dependency matrices and physical clocks. In *Symp. on Cloud Computing*, pages 11:1–11:14, Santa Clara, CA, USA, Oct. 2013. Assoc. for Computing Machinery.

[18] J. Du, C. Iorgulescu, A. Roy, and W. Zwaenepoel. GentleRain: Cheap and scalable causal consistency with physical clocks. In *Symp. on Cloud Computing*, 2014.

[19] R. Friedman and K. Birman. Trading consistency for availability in distributed systems. Technical Report TR96-1579, Cornell U., Computer Sc., Ithaca NY, USA, Apr. 1996.

[20] S. Gilbert and N. Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2):51–59, 2002.

[21] P. R. Johnson and R. H. Thomas. The maintenance of duplicate databases. Internet Request for Comments RFC 677, Information Sciences Institute, Jan. 1976.

[22] A. Kansal, B. Urgaonkar, and S. Govindan. Using dark fiber to displace diesel generators. In *Hot Topics in Operating Systems*, Santa Ana Pueblo, NM, USA, 2013.

[23] R. Ladin, B. Liskov, and L. Shrira. Lazy replication: Exploiting the semantics of distributed services. *Operating Systems Review*, 25(1):49–55, Jan. 1991.

[24] C. Li, D. Porto, A. Clement, J. Gehrke, N. Preguiça, and R. Rodrigues. Making geo-replicated systems fast as possible, consistent when necessary. In *Symp. on Op. Sys. Design and Implementation (OSDI)*, pages 265–278, Hollywood, CA, USA, Oct. 2012.

[25] 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 *Symp. on Op. Sys. Principles (SOSP)*, pages 401–416, Cascais, Portugal, Oct. 2011. Assoc. for Computing Machinery.

[26] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen. Stronger semantics for low-latency geo-replicated storage. In *Networked Sys. Design and Implem. (NSDI)*, pages 313–328, Lombard, IL, USA, Apr. 2013.

[27] P. Mahajan, L. Alvisi, and M. Dahlin. Consistency, availability, and convergence. Technical Report UTCS TR-11-22, Dept. of Comp. Sc., The U. of Texas at Austin, Austin, TX, USA, 2011.

[28] P. Mahajan, S. Setty, S. Lee, A. Clement, L. Alvisi, M. Dahlin, and M. Walfish. Depot: Cloud storage with minimal trust. *Trans. on Computer Systems*, 29(4): 12:1–12:38, Dec. 2011.

[29] K. Petersen, M. J. Spreitzer, D. B. Terry, M. M. Theimer, and A. J. Demers. Flexible update propagation for weakly consistent replication. In *Symp. on Op. Sys. Principles (SOSP)*, pages 288–301, Saint Malo, Oct. 1997. ACM SIGOPS.

[30] Redis. An open source key-value store. http://redis.io/, May 2014.

[31] N. Schiper, P. Sutra, and F. Pedone. P-Store: Genuine partial replication in wide area networks. In *Symp. on Reliable Dist. Sys. (SRDS)*, pages 214–224, New Dehli, India, Oct. 2010. IEEE Comp. Society.

[32] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski. A comprehensive study of Convergent and Commutative Replicated Data Types. Rapp. de Recherche 7506, Institut National de la Recherche en Informatique et Automatique (Inria), Rocquencourt, France, Jan. 2011.

[33] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski. Conflict-free replicated data types. In X. Défago, F. Petit, and V. Villain, editors, *Int. Symp. on Stabilization, Safety, and Security of Distributed Systems (SSS)*, volume 6976 of *Lecture Notes in Comp. Sc.*, pages 386–400, Grenoble, France, Oct. 2011. Springer-Verlag.

[34] Y. Sovran, R. Power, M. K. Aguilera, and J. Li. Transactional storage for geo-replicated systems. In *Symp. on Op. Sys. Principles (SOSP)*, pages 385–400, Cascais, Portugal, Oct. 2011. Assoc. for Computing Machinery.

[35] D. B. Terry, A. J. Demers, K. Petersen, M. J. Spreitzer, M. M. Theimer, and B. B. Welch. Session guarantees for weakly consistent replicated data. In *Int. Conf. on Para. and Dist. Info. Sys. (PDIS)*, pages 140–149, Austin, Texas, USA, Sept. 1994.

[36] M. Zawirski, N. Preguiça, S. Duarte, A. Bieniusa, V. Balegas, and M. Shapiro. Write fast, read in the past: Causal consistency for client-side applications. Rapp. de Recherche RR-8729, Institut National de la Recherche en Informatique et Automatique (Inria), Rocquencourt, France, May 2015.