2014 IEEE 33rd International Symposium on Reliable Distributed Systems Workshops

SwiftCloud: Fault-Tolerant Geo-Replication Integrated all the Way to the Client Machine

Nuno Preguiça[†]

joint work with:

Marek Zawirski[*], Annette Bieniusa[‡], Sérgio Duarte[†], Valter Balegas[†], Carlos Baquero[§], Marc Shapiro[*]
[*]*Inria & UPMC-LIP6*
[†]*NOVA-LINCS/CITI/U. Nova de Lisboa*
[‡]*U. Kaiserslautern*
[§]*INESC Tec & U. Minho*

***Abstract*—Client-side logic and storage are increasingly used in web and mobile applications to improve response time and availability. Current approaches tend to be ad-hoc and poorly integrated with the server-side logic. We present a principled approach to integrate client- and server-side storage. We support both mergeable and strongly consistent transactions that target either client or server replicas and provide access to causally-consistent snapshots efficiently. In the presence of infrastructure faults, a client-assisted failover solution allows client execution to resume immediately and seamlessly access consistent snapshots without waiting. We implement this approach in SwiftCloud, the first transactional system to bring geo-replication all the way to the client machine.**

Example applications show that our programming model is useful across a range of application areas. Our experimental evaluation shows that SwiftCloud provides better fault tolerance and at the same time can improve both latency and throughput by up to an order of magnitude, compared to classical geo-replication techniques.

I. Introduction

Cloud computing infrastructures support a wide range of services, from social networks and games to collaborative spaces and online shops. Cloud platforms improve availability and latency by geo-replicating data in several data centers (DCs) across the world [1], [2], [3], [4], [5], [6]. Nevertheless, the closest DC is often still too far away for an optimal user experience. For instance, round-trip times to the closest Facebook DC range from several tens to several hundreds of milliseconds, and several round trips per operation are often necessary [7]. Furthermore, mobile clients may be completely disconnected from any DC for an unpredictable period of minutes, hours or days.

Caching data at client machines can improve latency and availability for many applications, and even allow for a temporary disconnection. While increasingly used, this approach often leads to ad-hoc implementations that integrate poorly with server-side storage and tend to degrade data consistency guarantees. To address this issue, we present SwiftCloud, the first system to bring geo-replication all the way to the client machine and to propose a principled approach to access data replicas at client machines and cloud servers.

Although extending geo-replication to the client machine seems natural, it raises two big challenges. The first one is to provide programming guarantees for applications running on client machines, at a reasonable cost at scale and under churn. Recent DC-centric storage systems [5], [6], [4] provide transactions, and combine support for causal consistency with mergeable objects [8]. Extending these guarantees to the clients is problematic for a number of reasons: standard approaches to support causality in client nodes require vector clocks entries proportional to the number of replicas; seamless access to client and server replicas require careful maintenance of object versions; fast execution in the client requires asynchronous commit. We developed protocols that efficiently address these issues despite failures, by combining a set of novel techniques.

Client-side execution is not always beneficial. For instance, computations that access a lot of data, such as search or recommendations, or running strongly consistent transactions, is best done in the DC. SwiftCloud supports server-side execution, without breaking the guarantees of client-side in-cache execution.

The second challenge is to maintain these guarantees when the client-DC connection breaks. Upon reconnection, possibly to a different DC, the outcome of the client's in-flight transactions is unknown, and state of the DC might miss the causal dependencies of the client. Previous cloud storage systems either retract consistency guarantees in similar cases [5], [6], [9], or avoid the issue by waiting for writes to finish at a quorum of servers [4], which incurs high latency and may affect availability.

SwiftCloud provides a novel client-assisted failover protocol that preserves causality cheaply. The insight is that, in addition to its own updates, a client observes a causally-consistent view of stable (i.e., stored at multiple servers) updates from other users. This approach ensures that a client always observes his previous updates and that it can safely

978-1-4799-7361-3/14 $31.00 © 2014 IEEE
DOI 10.1109/SRDSW.2014.33

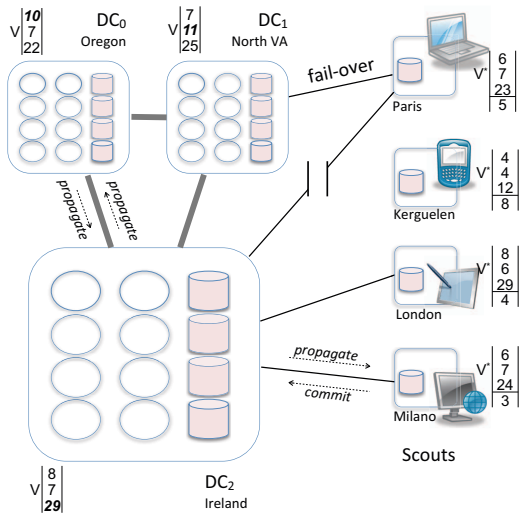

Figure 1. SwiftCloud system structure.

reconnect to other DC, as it can replay its own updates and other observed updates being stable, are already in other DCs.

Experimental evaluation shows that under sufficient access locality, SwiftCloud enjoys order-of-magnitude improvements in both response time and throughput over the classical approach. This is because, not only reads (if they hit in the cache), but also updates commit at the client side without delay; servers only need to store and forward updates asynchronously. Although our fault tolerance approach delays propagation, the proportion of stale reads remains under 1%.

In the remaining of this paper, we briefly overview the key solutions developed in the context of SwiftCloud [10].

II. System overview

SwiftCloud is a data storage systems for cloud platforms that spans both client nodes and data center servers (DCs), as illustrated in Figure 1. The core of the system consists of a set of *data centers (DCs)* that replicate every object. At the periphery, applications running in *client nodes* access the system through a local module called *scout*. A scout caches a subset of the objects. If the appropriate objects are in cache, responsiveness is improved and a client node supports disconnected operation.

SwiftCloud provides a straightforward transactional key-object API. An application executes transactions by interactively executing sequences of reads and updates, concluded by either a commit or rollback.

Our transactional model, Transactional Causal+ Consistency, offers the following guarantees: every transaction reads a causally consistent snapshot; updates of a transaction are atomic (all-or-nothing) and isolated (no concurrent transaction observes an intermediate state); and concurrently committed updates do not conflict.

This transactional model allows different clients to observe the same set of concurrent updates applied in different orders, which poses a risk of yielding different operation outcomes on different replicas or at different times. We address this problem by disallowing non-commutative (order-dependent) concurrent updates. Practically, we enforce this property with two different types of transactions: *Mergeable* and *Classical, non-mergeable* transaction, akin to the model of Walter [4] or Red-Blue [9]:

Mergeable transactions commute with each other and with non-mergeable transactions, which allows to execute them immediately in the cache, commit asynchronously in the background, and remain available in failure scenarios. Mergeable transaction are either read-only transaction or update transactions that modify Conflict-free Replicated Data Types (CRDT)[8], [11]. CRDTs encapsulate the logic to merge concurrent updates deterministically, independently of the order of execution of updates.

Classical transactions provide the traditional strongly-consistent transaction model, in which non-commuting concurrent updates conflict (as determined by an oracle on pairs of updates) and cannot both commit. These transactions execute completely in the data centers.

III. Algorithms for Mergeable transactions

We now present the key ideas of the algorithms for executing *mergeable transactions* in a failure-free case. In the next section we address the problems posed by failures.

An application issues a *mergeable transaction* iteratively through the scout. Reads are served from the local scout; on a cache miss, the scout fetches the data from the DC it is connected to. Updates execute in a local copy. When a mergeable transaction terminates, it is locally committed and updates are applied to the scout cache. Updates are also propagated to a data center (DC) for being globally committed. The DC eventually propagates the effects of transactions to other DCs and other scouts scouts as needed.

Atomicity and Isolation: For supporting atomicity and isolation, a transaction reads from a database snapshot. Each transaction is assigned a DC timestamp by the DC that received it from the client. Each DC maintains a vector clock with the summary of all transactions that have been executed in that DC, which is updated whenever a transaction completes its execution in that DC. This vector has as n entries, with n the number of DCs. Each scout maintains a vector clock with the version of the objects in the local cache.

When a transaction starts in the client, the current version of the cache is selected as the transaction snapshot. If the transaction accesses an object that is not present in the cache, the appropriate version is fetched from the DC - to this end, DCs maintain recent versions of each object.

Read your writes: When a transaction commits in the client, the local cache is updated. The following transactions access a snapshot that includes these locally committed transactions. To this end, each transaction executed in the client is assigned a scout timestamp. The vector that summarizes the transactions reflected in the local cache has $n+1$ entries, with the additional entry being used to summarize locally submitted transactions. This approach guarantees that a client always reads a state that reflects his previous transactions.

Causality: The system ensures the invariant that every node (DC or scout) maintains a causally-consistent set of object versions. To this end, a transaction only executes in a DC after its dependencies are satisfied - the dependencies of a transaction, summarized in the transaction snapshot, are propagated both from the client to the initial DC and from one DC to other DCs.

When a scout caches some object, the DC it is connected to becomes responsible of notifying it with updates to those cached objects. SwiftCloud includes a notification subsystem that guarantees that updates from a committed transaction are propagated atomically and respecting causality. As a result, the cache in the scout is also causally consistent.

IV. Fault-tolerant session and durability

We discuss now how SwiftCloud handles network, DC and client faults, focusing on client-side mergeable transactions. When a scout loses communication with its current DC, due to network or DC failure, the scout may need to switch over to a different DC. The latter's state is likely to be different, and it might have not processed some transactions observed or indirectly observed (via transitive causality) by the scout. In this case, ensuring that the clients' execution satisfies the consistency model and the system remains live is more complex. As we will see, this also creates problems with durability and exactly-once execution.

A. Causal dependency issue

When a scout switches to a different DC, the state of the new DC may be unsafe, because some of the scout's causal dependencies are missing. Some geo-replication systems avoid creating dangling causal dependencies by making synchronous writes to multiple data centers, at the cost of high update latency [1]. Others remain asynchronous or rely on a single DC, but after failover clients are either blocked or they violate causal consistency [5], [6], [9]. The former systems trade consistency for latency, the latter trade latency for consistency or availability.

An alternative approach would be to store the dependencies on the scout. However, since causal dependencies are transitive, this might include a large part of the causal history and a substantial part of the database.

Our approach is to make scouts co-responsible for the recovery of missing session causal dependencies at the new DC. Since, as explained earlier, a scout cannot keep track of all transitive dependencies, we restrict the set of dependencies. We define a transaction to be K*-durable* [12] at a DC, if it is known to be durable in at least K DCs, where K is a configurable threshold. Our protocols let a scout observe only the union of: *(i)* its own updates, in order to ensure the "read-your-writes" session guarantee [13], and *(ii)* the K-durable updates made by other scouts, to ensure other session guarantees, hence causal consistency. In other words, the client depends only on updates that the scout itself can send to the new DC, or on ones that are likely to be found in a new DC. When failing over to a new DC, the scout helps out by checking whether the new DC has received its recent updates, and if not, by repeating the commit protocol with the new DC.

SwiftCloud prefers to serve a slightly old but K-durable version, instead of a more recent but more risky version. Instead of the consistency and availability vs. latency tradeoff of previous systems, SwiftCloud trades availability for staleness.

B. Durability and exactly-once execution issue

A scout sends each transaction to its DC to be globally-committed. The DC assigns a DC timestamp to the transaction, and eventually transmits it to every replica. If the scout does not receive an acknowledgment, it must retry the global-commit, either with the same or with a different DC. However, the outcome of the initial global-commit remains unknown. If it happens that the global commit succeeded with the first DC, and the second DC assigns a second DC timestamp, the danger is that the transaction's effects could be applied twice under the two identities.

For some data types, this is not a problem, because their updates are idempotent, for instance `put(key,value)` in a last-writer-wins map. For other mergeable data types, however, this is not true: think of executing `increment(10)` on a counter. Systems restricted to idempotent updates can be much simpler [6], but in order to support general mergeable objects with rich merge semantics, SwiftCloud must ensure exactly-once execution.

Our approach separates the concerns of tracking causality and of uniqueness, following by the insight of [14]. Recall that a transaction has both a scout timestamp and a DC timestamp. The scout timestamp identifies a transaction uniquely, whereas the DC timestamp is used when a summary of a set of transactions is needed. Whenever a scout globally-commits a transaction at a DC, and the DC does not have a record of this transaction already, the DC assigns it a new DC timestamp. This approach makes the system available, but may assign several DC timestamp aliases for the same transaction. All alias DC timestamps are equivalent in the sense that, if updates of T' depend on T, then T' comes after T in the causality order, no matter what DC timestamp T' uses to refer to T.

When a DC processes a commit record for an already-known transaction with a different DC timestamp, it adds the alias DC timestamp to its commit record on durable storage.

To provide a reliable test whether a transaction is already known, each DC maintains durably a map of the last scout timestamp received from each scout. Thanks to causal consistency, this value is monotonically non-decreasing. Thus, a DC knows that a transaction being received for global-commit from a scout has already been processed if the recorded value for that scout is greater or equal to the scout timestamp of the received transaction.

V. Final remarks

We overview the design of SwiftCloud, the first system that brings geo-replication to the client machine, providing a principled approach for using client and data center replicas. SwiftCloud allows applications to run transactions in the client machine, for common operations that access a limited set of objects, or in the DC, for transactions that require strong consistency or accessing a large number of objects. Our evaluation of the system [10] shows that the latency and throughput benefit can be huge when compared with traditional cloud deployments for scenarios that exhibit good locality, a property verified in real workloads [15].

SwiftCloud also proposes a novel client-assisted failover mechanism that trades latency by a small increase in staleness. Our evaluation shows that our approach helps reducing latency while increasing stale reads by less than 1%.

Acknowledgment

This research was supported in part by EU FP7 project SyncFreee (grant agreement no 609551), ANR project ConcoRDanT (ANR-10-BLAN 0208), by the Google Europe Fellowship in Distributed Computing awarded to Marek Zawirski, and by Portuguese FCT/MCT projects PEst-OE/EEI/UI0527/2014 and PTDC/EEI-SCR/1837/2012 and Phd scholarship awarded to Valter Balegas (SFRH/BD/87540/2012).

References

[1] 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 *OSDI*. Hollywood, CA, USA: Usenix, Oct. 2012, pp. 251–264.

[2] A. Thomson, T. Diamond, S.-C. Weng, K. Ren, P. Shao, and D. J. Abadi, "Calvin: fast distributed transactions for partitioned database systems," in *SIGMOD*, Scottsdale, AZ, USA, May 2012, pp. 1–12.

[3] J. Baker, C. Bond, J. C. Corbett, J. Furman, A. Khorlin, J. Larson, J.-M. Léon, Y. Li, A. Lloyd, and V. Yushprakh, "Megastore: Providing scalable, highly available storage for interactive services," in *CIDR*, Asilomar, CA, USA, Jan. 2011, pp. 229–240.

[4] Y. Sovran, R. Power, M. K. Aguilera, and J. Li, "Transactional storage for geo-replicated systems," in *SOSP*. Cascais, Portugal: Assoc. for Comp. Mach., Oct. 2011, pp. 385–400.

[5] 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 *SOSP*. Cascais, Portugal: Assoc. for Comp. Mach., Oct. 2011, pp. 401–416.

[6] ——, "Stronger semantics for low-latency geo-replicated storage," in *NSDI*, Lombard, IL, USA, Apr. 2013, pp. 313–328.

[7] M. P. Wittie, V. Pejovic, L. Deek, K. C. Almeroth, and B. Y. Zhao, "Exploiting locality of interest in online social networks." Philadelphia, PA, USA: Assoc. for Comp. Mach., Dec. 2010, pp. 25:1–25:12.

[8] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski, "Conflict-free replicated data types," in *SSS*, ser. LNCS, X. Défago, F. Petit, and V. Villain, Eds., vol. 6976. Grenoble, France: Springer Verlag, Oct. 2011, pp. 386–400.

[9] C. Li, D. Porto, A. Clement, J. Gehrke, N. Preguiça, and R. Rodrigues, "Making geo-replicated systems fast as possible, consistent when necessary," in *OSDI*, Hollywood, CA, USA, Oct. 2012, pp. 265–278.

[10] M. Zawirski, A. Bieniusa, V. Balegas, S. Duarte, C. Baquero, M. Shapiro, and N. Preguiça, "SwiftCloud: Fault-tolerant geo-replication integrated all the way to the client machine," INRIA, Rapp. Rech. RR-8347, Aug. 2013.

[11] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski, "Convergent and commutative replicated data types," *Bulletin of the European Association for Theoretical Computer Science (EATCS)*, no. 104, pp. 67–88, Jun. 2011.

[12] P. Mahajan, S. Setty, S. Lee, A. Clement, L. Alvisi, M. Dahlin, and M. Walfish, "Depot: Cloud storage with minimal trust," *TOCS*, vol. 29, no. 4, pp. 12:1–12:38, Dec. 2011. [Online]. Available: http://doi.acm.org/10.1145/2063509.2063512

[13] 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 *PDIS*, Austin, Texas, USA, Sep. 1994, pp. 140–149.

[14] P. S. Almeida, C. Baquero, R. Gonçalves, N. M. Preguiça, and V. Fonte, "Scalable and accurate causality tracking for eventually consistent stores," in *Proc. 14th Int. Conf. Distributed Applications and Interoperable Systems (LNCS 8460)*, 2014, pp. 67–81.

[15] F. Benevenuto, T. Rodrigues, M. Cha, and V. Almeida, "Characterizing user behavior in online social networks," in *Proceedings of the 9th ACM SIGCOMM conference on Internet measurement conference*, ser. IMC '09, 2009, pp. 49–62.