Optimistic Causal Consistency for Geo-Replicated Key-Value Stores

Kristina Spirovska, Diego Didona, Willy Zwaenepoel
EPFL
Email: first.last@epfl.ch

Abstract—**In this paper we present a new approach to implementing causal consistency in geo-replicated data stores, which we call Optimistic Causal Consistency (OCC). The optimism in our approach lies in that updates from a remote data center are immediately made visible in the local data center, without checking if their causal dependencies have been received. Servers perform the dependency check needed to enforce causal consistency only upon serving a client operation, rather than on the receipt of a replicated data item as in existing systems.**

OCC explores a novel trade-off in the landscape of causal consistency protocols. The potentially blocking behavior of OCC makes it vulnerable to network partitions. Because network partitions are rare in practice, however, OCC chooses to trade availability to maximize data freshness and reduce the communication overhead. We further propose a recovery mechanism that allows an OCC system to fall back on a pessimistic protocol to continue operating even during network partitions.

POCC is an implementation of OCC based on physical clocks. We show that OCC improves data freshness, while offering comparable or better performance than its pessimistic counterpart.

I. Introduction

Causal consistency is an attractive model for large-scale geo-replicated data platforms, because it hits a sweet spot in the trade-off between ease of programming and performance [1].

Existing causally consistent systems implement causal consistency by different techniques, but they all share the following key mechanism. When serving a read operation, a server returns the most recent version that is stable in its data center (DC) [1], [2], [3], [4], [5]. We define a version of an item as *stable* in a DC, if all its dependencies have been replicated in that DC. More recent versions may exist, but these cannot be returned to the client, because they are not stable, and may lead to causality violations later.

This approach allows these systems to tolerate network partitions and DC failures. It has, however, two drawbacks. First, it delays the visibility of new versions of data items, increasing the staleness of the data returned to clients. Second, it requires the implementation of dependency checking [1], [2] or stabilization protocols [3], [4] that result in computational and communication overhead.

Optimistic Causal Consistency. In this paper we argue that existing protocols are too *pessimistic* for modern data center deployments and we propose Optimistic Causal Consistency (OCC).

According to OCC a server always returns the most recent available version of an item. Unresolved dependencies are detected by the server upon serving an operation, without the need for synchronization with other servers. This is accomplished by means of cheap dependency meta-data supplied by the client. When an unresolved dependency is detected, a server blocks the read while it receives the dependency.

The effectiveness of our optimistic approach stems from two main insights. First, recent works have revealed that update replication in data stores exhibits a *naturally consistent order*. In other words, a data item is typically replicated (and accessed) after its dependencies have already been propagated [6], [7]. Hence, the most recent version of a data item can be returned without violating consistency. OCC leverages this insight by having servers waiting to receive missing dependencies when serving a client's request, rather than relying on expensive dependency checking and stabilization protocols. Servers rarely incur such waiting overhead, because of the naturally consistent order of updates.

Second, network partitions are relatively rare events, and complete DC failures even more rare [6], [8]. OCC leverages this insight by avoiding (most of) the overhead associated with network partition tolerance during normal operation, and by incurring this overhead only when a network partition is actually occurring. To this end, OCC entails the infrequent –hence cheap– execution of a recovery protocol to allow a system to fall back on a pessimistic protocol in the presence of a network partition. This design contrasts with existing ones, which incur dependency checking overhead and expose stale data items to clients even under normal operational conditions.

II. Optimistic Causal Consistency

A. Causal Consistency

Causal consistency requires that servers of a system return values that are consistent with the order defined by the *causality* relationship. Causality is a happens-before relationship between two events [9]. For two operations a, b, we say that a causally depends on b, and write $a \rightsquigarrow b$, if and only if at least one of the following conditions holds: *i*) a and b are operations in a single thread of execution, and a happens before b; *ii*) a is a write operation, b is a read operation, and b reads the value written by a; *iii*) There is some other operation c such that $a \rightsquigarrow c$ and $c \rightsquigarrow b$.

B. System Model

We assume a distributed, large scale, multiversion key-value store. The keyspace is split into N disjoint partitions according to a hash function. Each partition is replicated at M different sites, each corresponding to a different DC. Hence, a full copy of the data is stored at each DC. We define m as the local replica to which a client connects. We further assume nodes in the system can communicate through point to point lossless FIFO channels.

We use lower case letters, e.g., x, to refer to a key and the corresponding capital letter, e.g., X, for a version of the key.

The system provides the following operations:

- PUT(key, val): A PUT operation assigns value *val* to an item identified by *key*.
- val ← GET(key): A GET operation returns the value of the item identified by *key*. The return value must not break causal consistency. Assume that $X \rightsquigarrow Y$. Suppose that a client c issues a GET(y) operation, receiving Y as result. Then, any subsequent GET(x) operation issued by c must return either X or a version X' such that $X' \not\rightsquigarrow X$.
- $\langle$vals$\rangle$ ← RO-TX$\langle$keys$\rangle$: This operation provides a causally consistent read-only transaction [1], [2]. If a read-only transaction returns X and Y, and $X \rightsquigarrow Y$, then there does not exist another version of x, X', such that $X \rightsquigarrow X' \rightsquigarrow Y$.

We assume the last-writer-wins rule to arbitrate conflicting updates [2]. OCC can, nevertheless, be implemented with other techniques to achieve convergence [1], [4].

C. The Design of OCC

The overall goal of OCC is to maximize the freshness of data returned to clients without the need to rely on dependency checking messages or stabilization protocols to enforce causal consistency. To achieve this goal, OCC implements a client-assisted lazy dependency resolution protocol.

Data freshness maximization. Maximizing the freshness of data returned to the client takes a different meaning depending on whether the client performs a GET operation or a RO-TX. On a GET(x) from client c, OCC always returns to c the most recent available version of an item that respects causal consistency, even if this version is not stable yet. If c does not have any dependencies on versions of x later than the most recent version currently available on the server, then this version is returned immediately. Otherwise, the GET operation is blocked until a newer version arrives that satisfies c's dependency on x. The difference with existing, pessimistic designs lies in that OCC allows p to return a version of x even if such version is not stable yet.

In case c reads x from p in the context of a RO-TX operation, p cannot safely return the freshest available version of x. Assume that X is the freshest version of x and that there exist Z, Z' such that $Z \rightsquigarrow Z' \rightsquigarrow X$. If c reads Z within a transaction, returning X in the same transaction would violate the semantics of the RO-TX operation. To explain how OCC maximizes the freshness of data returned within the scope of a transaction, we introduce the concept of a snapshot visible to a transaction. We define it as the set of data items that can be returned to c as a result of a transaction without violating the semantics of the RO-TX operation. The optimism of OCC lies in how the boundaries of a transaction's snapshot are determined. In OCC, the boundaries of a transactional snapshot are defined on the basis of the items received by nodes in the local data center at the time the transaction was issued. In existing systems, instead, such boundaries are determined by the set of items that are stable at the time the transaction is issued [3], [4].

Client-assisted lazy dependency resolution. Because OCC exposes unstable items, a client c could establish a dependency towards an item Z that has not been received yet by its corresponding local partition p_Z. If then c wants to read such item, OCC must prevent p_Z from returning a version of z that is not causally consistent with c's history. To this end, OCC implements a client-assisted lazy dependency resolution protocol.

Clients store information about the causal dependencies established when performing operations. For example, if c reads Y and there exists X such that $X \rightsquigarrow Y$, c needs to record it has established a dependency towards X and Y. The current client-side dependency meta-data is supplied by c when it performs an operation.

For a read operation (GET or RO-TX), such information is needed to allow a server p to determine whether its own state is consistent with c's history. Referring to the previous example, if c wants to read X after it has read Y, the dependency meta-data provided by c, together with some state information that p locally stores, allows p to check whether it has already received X or not. If p has already received X, then the freshest local version of x that p stores is compatible with c's history, and it is returned to c. If p has not received X yet, p must receive it before serving c's request. In this case, p simply stalls c's request until it receives X.

This lazy dependency resolution scheme is cheap to implement, as it does not require any synchronization among servers.

For a PUT operation, the client provides dependency information to set the dependencies of the newly created item.

D. Data Freshness vs Availability

OCC aims to maximize the freshness of data exposed to clients. Its potentially blocking behavior, however, makes it vulnerable to network partitions. Suppose, for example, there are two items X, Y such that $X \rightsquigarrow Y$. Assume Y gets replicated to DC', but X is prevented from doing so by a network partition. If a client c in DC' establishes a dependency on Y and then tries to read X from node p, p must block the read operation until the network partition heals to receive X and, thus, to preserve causal consistency. If the partition does not heal, however, the client's read request is blocked indefinitely. In other words, OCC is not *always available*, because it cannot guarantee that any client's operation is always completed in a finite amount of time.

Highly available OCC. To circumvent this limitation, we propose to augment OCC with a recovery procedure aimed at regaining availability during network partitions. We explain our recovery mechanism starting from the blocking condition example. If p blocks while serving a request from c, as soon as p realizes there is a network partition occurring, it closes the session with c. A network partition can be identified by p if it blocks for more than a configurable amount of time. At this point, c re-initializes its session. This new session is managed according to a pessimistic protocol and, therefore, it is ensured not to block even during the ongoing network partition. The cost for this session re-initialization is that the client might not be able to see the same version of some data items read or written in the optimistic session.

Equipped with this recovery mechanism, OCC can be implemented in a *highly available* fashion, representing a novel and unexplored trade-off between data freshness and availability.

OCC trades the ability to *seamlessly* tolerate network partitions –only minimally impacting the availability property– with a higher freshness of the data returned to clients.

We believe that, for many applications, e.g., social networks, this is a reasonable, if not favorable, trade-off. As already stated, in fact, careful engineering and redundant links make modern geo-distributed DCs reliable enough to regard network partitions as infrequent events [8]. Further, we argue that the cost of re-initializing a client session is affordable both from a time and data visibility perspective.

In case of network partitions, our recovery mechanism does not expose the application to any behavior or anomaly that is not encompassed by a pessimistic implementation of causal consistency. In fact, an application that relies on the causal semantics must be coded to tolerate the unexpected re-initialization of a session. This stems by the stickiness property of causal consistency, which means clients always contact the same server for each of their requests.

When a network partition heals, a client can be promoted again to the optimistic version. If a partition never heals, e.g., in the case of full DC failure, the client keeps operating according to the pessimistic design. We further discuss the unlikely case of full DC failure in the extended technical report [10]. In the report we also describe how to arrange for the safe co-existence of clients operating optimistically and pessimistically.

III. POCC: A Scalable Implementation of OCC

We implement OCC in a new system called POCC. The key challenge in POCC is to succinctly encode the dependencies of clients. OCC can in fact be implemented with any dependency tracking mechanism that has been proposed in the literature, e.g., dependency lists/matrices or physical scalar/vector clocks.

POCC efficiently tracks dependency by means of physical clocks and dependency vectors. In POCC each update u is assigned a physical clock timestamp that represents the time at which the corresponding item has been created and a dependency vector with one entry per DC.

POCC ensures that if $X \rightsquigarrow Y$ then the update time of Y is greater then the update time of X. This invariant is used to track dependencies on the client side and enforce causal consistency on the server side as we explain in the following.

The main difference between the optimistic and the pessimistic approaches lies in how POCC enforces causal consistency despite exposing possibly unstable items to client. For space constraint we only show how POCC achieves this goal by describing how it serves a GET operation. Further details can be found in [10].

During a session, a client c maintains a read dependency vector RDV_c, with one entry per DC. $RDV_c[i]$ is the update time of the item d with the highest timestamp such that $i)$ d has been originated at the $i-$th DC and $ii)$ c has read an item that depends on d.

A server p maintains a version vector VV, consisting of M physical timestamps corresponding to the latest updates received by p from each DC.

As stated in Section II-C, in the optimistic protocols, the GET operation might block. When a client c sends a GET request, besides the key k that needs to be read, it also sends RDV_c. Then the server checks whether its version vector is entry-wise greater than or equal to RDV_c. This check is not done for the m-th entry because dependencies towards local items are trivially always satisfied. If the check succeeds, it means that p has received all the items of its partition on which c depends. In this case, p returns the freshest version of k. If at least one remote entry of VV is smaller than RDV_c, it means that c might depend on a version of k that has not been replicated at p. Then, p has to wait for its receipt, or else it might return a version of the k that is not causally consistent with c's history. Then, p blocks until $RDV_c[i] \leq VV[i], i \neq m$.

IV. Evaluation

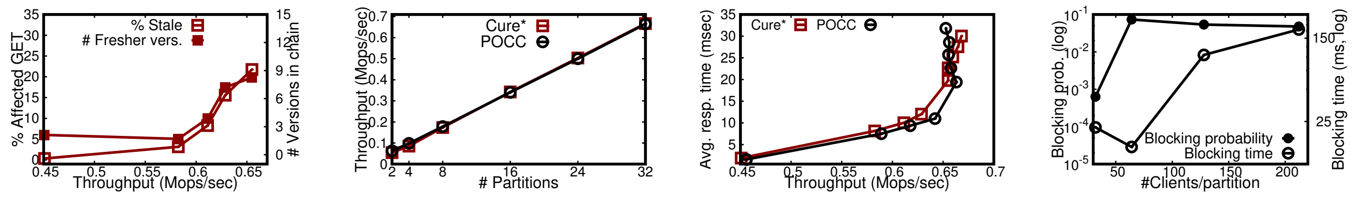
We show that POCC maximizes data freshness while delivering comparable or better performance than its pessimistic counterpart.

To evaluate the effectiveness of OCC, we compare the performance achieved by POCC with the one achieved by Cure*, a reimplementation of the Cure protocol [4] that follows our system model.

We focus on workloads composed of GET and PUT operations. A more extensive evaluation can be found in [10].

A. Experimental Testbed

We use an Amazon AWS deployment consisting of 3 DCs (located in Oregon, Virginia and Ireland) and 32 partitions per DC. We use *c*4.*large* instances, corresponding to 2 virtual CPUs and 3.75 GB of RAM. Each data partition is composed of one million 8 byte key-value pairs, and clients choose which key to access within each partition according to a zipf distribution with parameter 0.99. We introduce a think time of 25 milliseconds between client operations to simulate a more realistic workload. We run NTP to keep physical clocks synchronized. We run a workload with a 32:1 GET:PUT ratio over 2 to 32 partitions per DC.



(a) Data staleness on Cure*.

(b) Throughput while varying the number of partitions from 2 to 32.

(c) Avg. resp. time.

(d) Blocking behavior in POCC with different # clients per partition.

Fig. 1. Experimental results for POCC and Cure* for workloads with a 32:1 GET:PUT ratio

B. Experimental Results

The benefits of the optimistic over the pessimistic design in terms of data freshness are shown on Figure 1a. The plot shows the percentage of stale data items returned in Cure* and the number of fresher versions available in the version chain of a stale returned data item. The plot shows that the probability to return stale data increases with the load (and it grows as high as 20%). This is not only because of the higher remote update rate, but also because the higher contention on physical resources slows down the execution of the stabilization protocol needed to identify stable versions. In contrast, these values are always zero in POCC [1].

Figure 1b shows that the two systems achieve basically the same throughput, showing that the optimistic approach can be implemented with no throughput loss with respect to the corresponding pessimistic implementation.

Figure 1c shows that POCC achieves a slightly lower average operation response time than Cure* before reaching the saturation point at about 0.65 Mops/sec. POCC rarely blocks upon serving an operation and it is more resource efficient than Cure*. Under very high load, POCC performs slightly worse, because the better resource efficiency is outweighed by the increasing cost of blocking, which we discuss next.

Figure 1d reports the probability that an operation blocks and the average blocking time for a blocked operation. The plot shows that the blocking probability is negligible under moderate to high load and that becomes noticeable only as the system approaches the saturation point. Similarly, the blocking time is in the order of a few microseconds as long as the system does not become overloaded.

V. Related Work

Our proposal is primarily related to the vast literature on causally consistent systems [1], [2], [3], [4], [5]. The common trait of these systems is that they are pessimistic. That implies they proactively check the dependency of remote updates before making them visible, by means of dependency checking messages or stabilization protocol. OCC, instead, implements a lazy dependency checking scheme, only when needed upon a client's operation. As discussed, this design choice increases data freshness and improves resource efficiency at the cost of a slight penalty availability-wise.

[1]In bigger deployments the execution of the stabilization protocol would progress at a lower pace, with a commensurate increase in the perceived staleness. In addition, the presented results correspond to running the stabilization protocol every 5 milliseconds. Higher values would allow the system to reach only a slightly higher throughput [3], but would come at the cost of an increased data staleness. By contrast, POCC is immune to this trade-off.

VI. Conclusion

We have presented OCC, an optimistic approach to achieve causal consistency in geo-replicated key-value stores. OCC regards network partitions and data center failures as rare events in modern deployments. Therefore, OCC partially trades some of the availability properties achievable by causal consistency for a higher freshness in the data returned to clients and a higher resource efficiency. We have implemented OCC in a system named POCC. We have shown that POCC can achieve performance that is comparable or better than its "pessimistic" counterpart while maximizing data freshness.

Acknowledgment

This research has been supported by The Swiss National Science Foundation through Grant No. 166306. The authors are also grateful to Amazon for providing a research grant through AWS Cloud Credits.

References

[1] 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 *Proc. of SOSP*, 2011.

[2] ——, "Stronger semantics for low-latency geo-replicated storage," in *Proc. of NSDI*, 2013.

[3] J. Du, C. Iorgulescu, A. Roy, and W. Zwaenepoel, "Gentlerain: Cheap and scalable causal consistency with physical clocks," in *Proc. of SoCC*, 2014.

[4] D. D. Akkoorath, A. Tomsic, M. Bravo, Z. Li, T. Crain, A. Bieniusa, N. Preguiça, and M. Shapiro, "Cure: Strong semantics meets high availability and low latency," in *Proc. of ICDCS*, 2016.

[5] J. Du, S. Elnikety, A. Roy, and W. Zwaenepoel, "Orbe: Scalable causal consistency using dependency matrices and physical clocks," in *Proc. of SoCC*, 2013.

[6] P. Bailis, S. Venkataraman, M. J. Franklin, J. M. Hellerstein, and I. Stoica, "Probabilistically bounded staleness for practical partial quorums," *Proc. VLDB Endow.*, vol. 5, no. 8, pp. 776–787, Apr. 2012.

[7] H. Lu, K. Veeraraghavan, P. Ajoux, J. Hunt, Y. J. Song, W. Tobagus, S. Kumar, and W. Lloyd, "Existential consistency: Measuring and understanding consistency at facebook," in *Proc. of SOSP*, 2015.

[8] E. Brewer, "Cap twelve years later: How the "rules" have changed," *Computer*, vol. 45, no. 2, pp. 23–29, 2012.

[9] M. Ahamad, G. Neiger, J. E. Burns, P. Kohli, and P. W. Hutto, "Causal memory: Definitions, implementation, and programming," *Distributed Computing*, vol. 9, no. 1, pp. 37–49, 1995.

[10] K. Spirovska, D. Didona, and W. Zwaenepoel, "Optimistic causal consistency for geo-replicated key-value stores," EPFL-REPORT-225991, Tech. Rep., 2017.