Closing The Performance Gap between Causal Consistency and Eventual Consistency

Jiaqing Du Călin Iorgulescu Amitabha Roy Willy Zwaenepoel

EPFL

ABSTRACT

It is well known that causal consistency is more expensive to implement than eventual consistency due to its requirement of dependency tracking and checking for causality. To close the performance gap between the two consistency models, we propose a new protocol that implements causal consistency for both partitioned and replicated data stores.

Our protocol trades the visibility latency of updates across different client sessions for higher throughput. An update, either from a local client or a remote replica, is only visible to other clients after it is replicated by all replicas. As a result, a read operation never introduces dependencies to its client session. Only update operations introduce dependencies. By exploiting the transitive property of causality and total order update propagation, an update always has at most one dependency. By reducing the number of tracked dependencies and the number of messages for dependency checking down to one, we believe our protocol can provide causal consistency with similar cost to eventual consistency.

1. INTRODUCTION

Distributed data stores are a critical infrastructure component of many large-scale online services. Choosing a consistency model for those data stores is challenging. The CAP theorem shows that among Consistency, Availability, and (network) Partition-tolerance, a replicated system can only have two properties out of the three. Having all the three is impossible.

Among different consistency models, causal consistency [2] preserves the virtues of eventual consistency [11]: high availability, partition-tolerance, and low update latency. In addition, it guarantees that replicated updates are applied at each replica in an order that respects causality [2, 7]. However, causality does not come for free. With existing solutions, causal consistency is much more expensive to implement than eventual consistency due to tracking and checking dependencies.

The problem addressed in this paper is to provide an efficient and scalable implementation of causal consistency for both partitioned and replicated data stores. We propose a solution to close the performance gap between causal consistency and eventual consistency.

In a distributed data store, causality comes from two sources: 1) previous updates in the same client session and 2) reading updates created by a different client session. Each update causally depends on previous updates in the same session, which we define as *internal dependencies*. It also causally depends on updates of other sessions whose values are read previously in the same session, which we define as *external dependencies*. To provide causal consistency, a replica does not apply an update propagated from another replica until all its causal dependency states are installed locally.

Existing systems [6, 8, 9] that provide causal consistency track both internal and external dependencies by maintaining some dependency metadata at the client side. A client associates each update operation with the dependency metadata. When an update is propagated to another replica, the replica checks whether all the dependencies of this update have been applied at the local copy using its dependency metadata. Hence *dependency checking* during update replication may require messages to other local partitions. Compared with eventual consistency, this is the major performance overhead of causal consistency. Storing and transmitting dependency metadata also consumes CPU cycles, storage, and network bandwidth. To reduce the overhead of tracking and checking unnecessary dependencies to applications, Bailis et al. propose to let applications assist dependency tracking with better semantics knowledge [3, 4]. In this paper, we target causal consistency in the data store level, which is transparent to applications.

To demonstrate the problem, we compare the throughput of causal and eventual consistency by an experiment in a distributed key-value store. The data store provides read and write operations on a single item. It implements causal consistency as in COPS [8] and Eiger [9]. A client only tracks the nearest dependencies because of the transitivity of causality. In the experiment, a client reads a random item at each partition and updates one random item at a randomly selected partition in the local data center. The update is then propagated to replicas at remote data centers for replication. This workload stretches causal consistency since it creates dependencies across all partitions for each update operation. Notice that this workload is not rare in real world applications. For instance, the default page of a user of Twitter or Facebook loads at least dozens

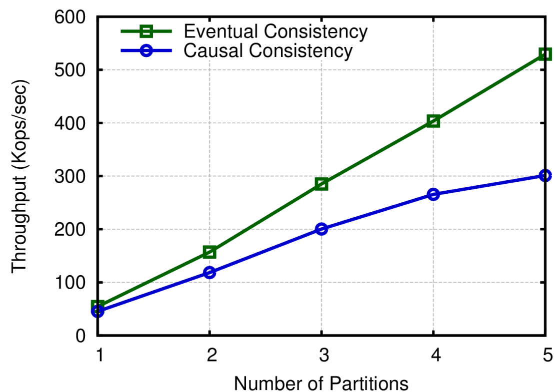

Figure 1: Throughputs of a distributed data store with eventual and causal consistency. Each partition is replicated by three replicas. Clients read an item from each partition once and update an item at one partition.

of or even hundreds of states. Any subsequent updates via the page, such as commenting on other users' posts, causally depends all the loaded states.

Figure 1 shows the throughputs of causal and eventual consistency. As more partitions are added to the system, the performance gap between the two becomes larger. For causal consistency, dependency checking messages to each local partition are the major source of performance degradation.

The fundamental reason of tracking and checking dependencies is that, when a client issues an update to the data store, it does not know for sure whether all its previously accessed states, i.e., its dependencies, are fully replicated. To make sure that this update appears after all its dependencies at all replicas, each remote replica performs dependency checking before it applies the update. To completely remove the overhead of tracking and checking dependencies, one may ask why we cannot wait until a state is fully replicated and then access it. However, if a system behaves like this, it becomes strongly consistent, losing the advantages of partition-tolerance and low update latency provided by causal consistency.

2. OVERVIEW OF SOLUTION

We propose a protocol that provides causal consistency for data stores that are both partitioned and replicated. It improves throughput by slightly increasing the visibility latency of updates across different client sessions. It completely removes the necessity of tracking and checking external dependencies.

The protocol allows a client to read other clients' updates only after they are fully replicated. In other words, updates that are not fully replicated are only visible to their originating clients, not other clients. Doing this does not compromise the virtues of causal consistency, because an update from a local client returns right after it is executed locally.

To distinguish updates from different clients, we associate each update with the id of the client/user that issues the operation. This introduces a small amount of metadata to each update. However, compared with existing approaches [4, 6, 8, 9], this metadata is much smaller and its size is constant. In addition, online social networks naturally require storing and associating user ids with user generated content in the database to distinguish data of different users [1, 5]. For this type and other similar applications, our protocol does not introduce more metadata.

To determine whether an update is fully replicated by all replicas of a partition, replicas of the same partition exchange replication confirmation messages in our protocol. This is not necessarily required by eventual consistency. To minimize the overhead of replication confirmation, we rely on version vectors [2, 10] to summarize the local and remote updates that a replica has applied. Replicas periodically exchange their version vectors and a replica knows which updates are fully replicated by examining the latest received version vectors. A tradeoff between the visibility latency of updates and the cost of replication confirmation exists. Notice that, if the interval is set to one hundred milliseconds, which is totally fine for online social networks, the overhead of replication confirmation would become negligible, compared with the update replication traffic.

Since a client can still access its own updates before they are fully replicated, our protocol still tracks and checks internal dependencies. Similar to existing approaches, it exploits the transitivity of causality to only track the nearest internal dependencies, i.e., the nearest previous update. The protocol also propagates the local updates of a partition to its replicas in their execution order. As a result, an update only possibly depends on one state created by the same client session. Only if this dependency is from a different partition, the dependency checking process at a remote replica requires a message to another local partition.

In summary, our protocol only tracks internal dependencies. For any type of workloads, an update has at most one dependency, which is constant and small. Our protocols trade the visibility latency of updates across different client sessions for higher throughput. As almost all existing causally consistent systems target online social networks, we believe delaying the visibility of other users' updates by less than a second is completely acceptable.

3. MODEL AND DEFINITION

In this section we describe our system model and define causality.

3.1 Architecture

We assume a distributed key-value store that manages a large set of data items. The key-value store provides two basic operations to the clients:

- PUT(key, val): A PUT operation assigns value *val* to an item identified by *key*. If item *key* does not exist, the system creates a new item with initial value *val*. If *key* exists, a new version storing *val* is created and overwrites the existing one.
- val ← GET(key): The GET operation returns the value of the item identified by *key*.

The data store is partitioned into N partitions, and each partition is replicated by M replicas. A data item is assigned to a partition based on the hash value of its key. In a typical configuration, the data store is replicated at M different data centers for high availability and low operation latency. The data store is fully replicated. All N partitions are present at each data center.

A client is collocated with the data store servers in a particular data center and only accesses those servers in the same data center. The application tier relies on the clients to access the underlying data store. A client does not issue the next operation until it receives the reply to the current one. Each operation happens in the context of a client session.

3.2 Causality

Causality is a *happens-before* relationship between two events [2, 7]. We denote causal order by $\rightsquigarrow$. For two operations a and b, if $a \rightsquigarrow b$, we say b depends on a or a is a dependency of b. $a \rightsquigarrow b$ if and only if one of the following three rules holds:

- Thread-of-execution. a and b are in a single thread of execution. a happens before b.
- Reads-from. a is a write operation and b is a read operation. b reads the state created by a.
- Transitivity. There is some other operation c that $a \rightsquigarrow c$ and $c \rightsquigarrow b$.

We define the *nearest dependencies* of a state as all the states that it directly depends on, without relying on the transitivity of causality.

4. PROTOCOL

In this section, we present our protocol that efficiently implements causal consistency for both partitioned and replicated data stores.

4.1 Definitions

The protocol tracks only internal dependencies of each client session by introducing some dependency metadata at both the client and server side. It also associates each data item a small and constant amount of dependency metadata. Table 1 provides a summary of the symbols used in the protocol.

Symbols	Definitions
N	number of partitions
M	number of replicas per partition
uid	client user id
DT_{uid}	dependency time of client uid
DP_{uid}	dependency partition of client uid
p_n^m	the mth replica of the nth partition
VV_n^m	version vector of p_n^m
VVS_n^m	version vector set of p_n^m
RVV_n^m	replication version vector of p_n^m
US_{uid}	update space of client uid
RS_n^m	replication space of p_n^m
GS_n^m	global space of p_n^m
d	item tuple $\langle k, v, ut, sr, dt, dp, idt, idr \rangle$
k	key
v	value
ut	update time
sr	source replica id
dt	dependency time
dp	dependency partition id
idt	item dependency time
idr	item dependency replica id

Table 1: Definition of symbols.

Client States. Without losing generality, we assume a client has one session to the data store. A client has a unique user identifier, uid. It maintains for its session a dependency time, DT_{uid}, and a dependency partition id, DP_{uid}. These two states record the (logical) update time and the local partition when and where the last update of the client issues.

Sever States. Each partition maintains a *version vector* (VV) [2, 10]. The version vector of partition p_n^m is VV_n^m, which consists of M non-negative integer elements. $VV_n^m[m]$ counts the number of updates p_n^m has executed locally. $VV_n^m[i]$ $(i \neq m)$ indicates that p_n^m has applied the first $VV_n^m[i]$ updates propagated from p_n^i, a replica of the same partition located at another data center. Each partition also maintains a *replication version vector* (RVV). RVV_n^m at p_n^m has a similar structure to VV_n^m. It indicates that the first $RVV_n^m[i]$ updates originated from p_n^i $(0 \leqslant i \leqslant M-1)$ have been fully replicated by all replicas of p_n^i.

A partition p_n^m divides its storage space into three parts to record updates of different stages. It temporarily stores an update from a local client with uid at the client's private *update space*, US_{uid}, which is only visible to its own client. The partition also temporarily keeps an update propagated from a remote replica for replication in the partition's *replication space*, RS_n^m, which is not visible at all to clients. Only updates in the partition's *global space*, GS_n^m, are visible to all clients. When

Algorithm 1 Client operations at client with *uid*

1: **Get**(key k, user id uid)
2: send $\langle$GetReq $k\rangle$ to server p_n^m
3: receive $\langle$GetReply $v\rangle$
4: return v
5: **Put**(key k, value v, user id uid)
6: send $\langle$PutReq $k, v, uid, DT_{uid}, DP_{uid}\rangle$ to server p_n^m
7: receive $\langle$PutReply $ut\rangle$
8: update dependency: $DT_{uid} \leftarrow ut$, $DP_{uid} \leftarrow n$

Algorithm 2 Server operations at partition p_n^m

1: **upon** receive $\langle$GetReq $k, uid\rangle$
2: **if** $\exists d \in US_{uid}$ s.t. $d.k = k$ **then**
3: obtain d from US_{uid}
4: **else** obtain d from GS_n^m
5: send $\langle$GetReply $d.v\rangle$ back to client
6: **upon** receive $\langle$PutReq $k, v, uid, DT_{uid}, DP_{uid}\rangle$
7: increment version vector: $VV_n^m[m] \leftarrow VV_n^m[m] + 1$
8: create new item d
9: set key: $d.k \leftarrow k$
10: set value: $d.v \leftarrow v$
11: set user id: $d.uid \leftarrow uid$
12: set update time: $d.ut \leftarrow VV_n^m[m]$
13: set source replica: $d.sr \leftarrow m$
14: set dependency time: $d.dt \leftarrow DT_{uid}$
15: set dependency partition: $d.dp \leftarrow DP_{uid}$
16: **if** $\exists$ latest $d' \in \{US_{uid} \cup GS_n^m\}$ s.t. $d'.k = d.k$ **then**
17: set item dependency time: $d.idt \leftarrow d'.ut$
18: set item dependency replica: $d.idr \leftarrow d'.sr$
19: add d to US_{uid}
20: send $\langle$PutReply $d.ut\rangle$ back to client
21: **for** each server p_n^i, $i \in \{0..M-1\}$ **do**
22: send $\langle$Replicate $d\rangle$ to p_n^i in order of $d.ut$
23: **upon** receive $\langle$Replicate $d\rangle$
24: **if** $d.dp \neq n$ **then**
25: wait until $VV_{d.dp}^m[d.sr] \geq d.dt$ via a dependency check message to $p_{d.dp}^m$
26: add d to RS_n^m
27: update version vector: $VV_n^m[d.sr] \leftarrow d.ut$
28: **upon** every Δ time
29: **for** each server p_n^i, $i \in \{0..M-1\}$ **do**
30: send $\langle$RepConfirm $VV_n^m\rangle$ to p_n^i
31: **upon** receive $\langle$RepConfirm $VV_n^k\rangle$
32: $VVS_n^m[k] \leftarrow VV_n^k$
33: **for** $i \in \{0..M-1\}$ **do**
34: $RVV_n^m[i] \leftarrow min(\{VVS_n^m[j][i] \mid j \in \{0..M-1\}\})$
35: **for** $d \in \{US_{uid} \mid$ all valid uid$\}$ **do**
36: **if** $d.ut \leqslant RVV_n^m[m]$ **then**
37: ApplyUpdate(d)
38: remove d from US_{uid}
39: **for** $d \in RS_n^m$ **do**
40: **if** $d.ut \leqslant RVV_n^m[d.sr]$ **then**
41: ApplyUpdate(d)
42: remove d from RS_n^m
43: **function** ApplyUpdate(d)
44: **if** $\exists\ d'$ s.t. $d'.k = d.k$ **then**
45: **if** $d.sr = d'.sr$ **then**
46: add d to GS_n^m
47: **else if** $d.idr = d'.sr \wedge d.idt = d'.ut$ **then**
48: add d to GS_n^m
49: **else if** $d.ut \oplus d.sr \geq d'.ut + d'.sr$ **then**
50: add d to GS_n^m
51: **else** discard d
52: **else** add d to GS_n^m

the partition knows that updates in the update space and replication space are fully replicated, it moves them to the global space to make them visible to all clients.

A partition updates an item by either executing an update request from its clients or by applying a propagated update from one of its replicas at other data centers. We call the partition that updates an item to the current value by executing a client request the *source partition* of the item.

Item Metadata. We represent an item d as a tuple $\langle k, v, ut, sr, dt, dp, idt, idr\rangle$. k is a unique key that identifies the item. v is the value of the item. ut is the *update time*, the logical creation time of the item at its source partition. sr is the *source replica*, the replica id of the item's source partition. dt is the *dependency time*, the update time of the item's nearest internal dependency. dp is the *dependency partition*, the source partition id of the item's nearest internal dependency. idt is the *item dependency time*, the update time of the previous version of the item, if exists. idr is the *item dependency replica*, the creation replica id of the previous version of the item, if exists.

We show all the possible metadata in the tuple for completeness. Some are not always necessary in implementation, such as sr, dt, and dp. idt and idr are not indispensable for providing causal consistency, as we show next in explanations of the protocol. They are used to detect and resolve conflicts caused by concurrent updates on the same item, which is important for certain applications.

4.2 Protocol

We now present how our protocol executes GET and PUT operations from clients and replicates PUT operations while preserving causality. Algorithm 1 and 2 show the pseudocode of the protocol running at the client and server side, respectively.

GET. A client sends a read request with an item key to the partition that stores the corresponding item. Upon receiving the request, the partition first checks whether a recent update to the required item by the client resides in its private update space, i.e., its own update that is not fully replicated yet. If not, the partition reads the item from the global space. It then re-

turns the item value back to the client. Since a partition never returns the updates of other clients unless they are fully replicated, read operations do not introduce any external dependencies. A partition does return a previous update of the same client if it is not fully replicated yet, which preserves *read-your-own-writes* session consistency.

PUT. A client sends an update request, including an item key, update value, and the client's dependency time and dependency partition id, to the partition that manages the item. Upon receiving the request, the partition increments its local logical clock in its version vector and creates a new version of the item by assigning it a tuple that consists of the key, value, update time, update replica id, dependency time, dependency partition id. If the partition already has an item with the same key, it sets the item dependency time and item dependency replica id of the new item to the existing one's dependency time and dependency replica id, respectively, for conflict detection during replication. The partition then stores the newly created item in the client's private update space. It sends a reply with the update time and its partition id back to the client. Upon receiving the reply, the client updates its dependency time and dependency partition id to track the nearest internal dependency.

Update replication. While a partition sends an update reply to the client, it also replicates the update by sending it to all replicas of the same partition at different data centers. A partition always propagates out its updates in the order of their update time. When a partition receives a propagated update, it checks its internal dependency if the update depends on a previous update to a different partition. The dependency checking requires one round of messages to the corresponding local partition. Its dependency is satisfied automatically if it depends on an update to the same partition, because of the total order update propagation. After the dependency checking, the partition stores the update in its replication space and increments the corresponding element in its version vector. The replicated update is not visible to the partition's local clients until it is fully replicated by all replicas of the partition.

Replication confirmation. To make local and replicated updates visible to all clients, a partition needs to find out when they are fully replicated. Replicas of the same partition periodically exchange their version vectors to tell each other the updates they have replicated. A partition builds its RVV by choosing the minimum one of each element from all the received version vectors. The RVV safely tells which updates from each replica of a partition are fully replicated. After the RVV is updated, the partition goes through updates in its update space and replication space. It moves all fully replicated updates to its global space so that they are visible to all local clients.

Conflict detection. Our protocol detects and resolves conflicting updates to the same item using standard techniques [8], as function ApplyUpdate in Algorithm 2 shows.

5. FUTURE WORK

We plan to implement the proposed protocol and compare its performance with eventual consistency and existing implementations of causal consistency. It is also interesting to investigate how our protocol should handle various failures.

References

[1] Big data in real-time at twitter. http://www.slideshare.net/nkallen/q-con-3770885, 2010.

[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, 1995.

[3] P. Bailis, A. Fekete, A. Ghodsi, J. M. Hellerstein, and I. Stoica. The potential dangers of causal consistency and an explicit solution. In *Proceedings of the Third ACM Symposium on Cloud Computing*, page 22. ACM, 2012.

[4] P. Bailis, A. Ghodsi, J. M. Hellerstein, and I. Stoica. Bolt-on causal consistency. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pages 761–772. ACM, 2013.

[5] N. Bronson, Z. Amsden, G. Cabrera, et al. Tao: Facebook's distributed data store for the social graph. In *USEXNIX ATC*, 2013.

[6] J. Du, S. Elnikety, A. Roy, and W. Zwaenepoel. Orbe: scalable causal consistency using dependency matrices and physical clocks. In *Proceedings of the 4th annual Symposium on Cloud Computing*, page 11. ACM, 2013.

[7] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.

[8] 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 *Proceedings of the 23rd ACM Symposium on Operating Systems Principles*, pages 401–416. ACM, 2011.

[9] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen. Stronger semantics for low-latency geo-replicated storage. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, pages 313–328. USENIX Association, 2013.

[10] D. S. Parker Jr, G. J. Popek, G. Rudisin, A. Stoughton, B. J. Walker, E. Walton, J. M. Chow, D. Edwards, S. Kiser, and C. Kline. Detection of mutual inconsistency in distributed systems. *IEEE Transactions on Software Engineering*, (3):240–247, 1983.

[11] W. Vogels. Eventually consistent. *Communications of the ACM*, 52(1):40–44, 2009.