Todo: Causally Consistent Transactions with Non-blocking Reads and Deterministic Ordering

I. Definitions and system Model

A. Causal Consistency

A system is causally consistent if its servers return values that are consistent with the order defined by the *causality* relationship. Causality is defined as a happens-before relationship between two events [1], [2]. For two operations $a,\ b$, we say that b causally depends on a, 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 version written by a; $iii)$ there is some other operation c such that $a \rightsquigarrow c$ and $c \rightsquigarrow b$. Intuitively, CC ensures that if a client has seen the effects of operation b and $a \rightsquigarrow b$, then the client also sees the effects of operation a.

We use lower case letters, e.g., x, to refer to a key and the corresponding capital letter, e.g., X to refer to a version of the key. We say that X depends on Y if the write of X causally depends on the write of Y.

B. Transactional Causal Consistency

Semantics. TCC extends CC by means of interactive read-write transactions in which clients can read and write multiple items. TCC enforces two properties.

1. Read from a causal snapshot. A causally consistent snapshot is a set of item versions such that all causal dependencies of those versions are also included in the snapshot. All transactions read from a causally consistent snapshot. For any two items, x and y, if $X \rightsquigarrow Y$ and both X and Y belong to the same causally consistent snapshot, then there is no other X', such that X' is created after X and $X \rightsquigarrow X' \rightsquigarrow Y$.

Transactional reads from a causal snapshot prevent undesirable anomalies which can arise by simply issuing multiple consecutive single read operations [3].

The majority of existing CC systems implement transactional reads by means of one-shot read-only transactions [3]–[6].

2. Atomic updates. Either all the items written by a transaction are visible to other transactions, or none is. If a transaction writes X and Y, then any snapshot visible to other transactions either includes both X and Y, or none of them.

C. System model

Todo is deployed across multiple geographic regions. A region consists of servers connected via a low-latency network. These servers typically reside within a single data-center or multiple data-centers that are in close proximity with each other. Each item is assigned to exactly one geographic region. This is called a home region in both SLOG and Detock. The identifier of the home region for a data item is stored in the header of that data item.

A region may store local data for which it is designated as the home region, and remote data which are materialized by replaying logs asynchronously replicated from other regions. Data is also partitioned locally within a region independently of home status: each partition might contain a mix of local and remote data.

We assume a multiversion data store. An update operation creates a new version of an item. In addition to the actual value of the key, each version also stores some metadata, in order to track causality. The system periodically garbage-collects old versions of items.

Similar to Detock, Todo relies on deterministic transaction execution to substantially reduce cross-region coordination. Fig. 1 shows the architecture of this style of deterministic system in a deployment over two regions A and B. Clients send transactions to their closest region. The first server that receives a transaction becomes its coordinator, which first resolves non-deterministic commands in the transaction (e.g. random() and time()), then attempts to extract its read/write set. When this is not possible via static analysis, the OLLP protocol is used, which obtains an initial estimate of the transaction's read/write set via a reconnaissance query. Each region maintains a distributed index called a Home Directory that contains the cached value of the current home for each known data item. The Forwarder of the coordinator uses this index to augment the read/write set with the home information of every data item. It then annotates the transaction with this augmented read/write set and forwards the transaction to its home region(s).

Once these transactions reach their home regions, they are put into batches and inserted into a Paxos-maintained local log by the Sequencers. This log is synchronously replicated within a region to tolerate failure of individual servers, and optionally to nearby regions to increase availability during (rare) failures of an entire region. A region can deterministically replay a local log from any other region R to obtain the state of R's local data. Therefore, persisting the local logs is sufficient for durability, and replication is performed by shipping these local logs. While it is not required for a region to hold any remote data, having a possibly stale copy of the remote data allows local snapshot reads of data from other regions and makes executing multi-home transactions (including the home-movement transactions in Section 4) faster. To this end, regions

Algorithm 1 Starting a new transaction in client c

```
1: function StartTxn
2:     for ($Key \in$ ReadSets $\cup$ WriteSets) do
3:         $regions \leftarrow$ RegionDirectory($Key$)
4:     end for
5:     $txn.region$ = unique regions in $regions$
6:     send $txn$ to all $txn.region$
7: end function
```

Algorithm 2 Scheduling transactions for execution in server s_m

```
1: upon receive $txn$ from client do
2:     $ht_m \leftarrow max\{Clock, ct\}$
3:     for ($Key \in$ ReadSets $\cup$ WriteSets) do
4:         $regions \leftarrow$ RegionDirectory($Key$)
5:     end for
6:     $id_T \leftarrow generateUniqueId()$
7:     $TX[id_T] \leftarrow ust_n^m$
8:     send $\langle$StartTxResp $\boldsymbol{id_T}, \boldsymbol{TX[id_T]}\rangle$
```

in Todo and Detock asynchronously exchange their local logs to each other, so each region eventually receives and replays the complete local log from every other region, as can be seen in the Log Managers of both regions.

II. Protocols of Todo

We now describe the meta-data stored and the protocols implemented by the clients and servers in Todo.

A. Meta-data

Items.

Clients.

Server. A server which is the local DC of the server. As a standard practice for systems that perform a 2PC protocol, server keeps two queues with prepared and committed transactions. The former, noted $Prepared$, stores transactions for which server has proposed a commit timestamp and for which server is waiting the commit message. The latter, noted $Committed$ stores transactions that have been assigned a commit timestamp and whose modifications are going to be applied to server.

B. Operations

Algorithm 1 reports the client protocol. Algorithm 2 report the protocols executed by a server to run a transaction.

Start. Client c starts a transaction T by picking at random a coordinator partition (denoted p_n^m) in the local DC and sending it a start request with ust_c. p_n^m uses ust_c to update ust_n^m, so that p_n^m can assign to the new transaction a snapshot that is at least as fresh as the one accessed by c in previous transactions. p_n^m uses its updated value of ust_n^m as snapshot for T. p_n^m also generates a unique identifier for T, denoted id_T, and inserts T in a private data structure. p_n^m replies to c with id_T and the snapshot timestamp ust_n^m.

Upon receiving the reply, c updates ust_c and evicts from the cache any version with timestamp lower than or equal to ust_c. c can prune such versions because the UST protocol ensures that they are included in the snapshot installed by any partition in the system. This means that if, after pruning, there is a version X in the private cache of c, then $X.ct > ust$ and hence the freshest version of x visible to c is X.

Client c locally buffers the writes in its write-set WS_c. If a key being written is already present in WS_c, then it is updated; otherwise, it is inserted in WS_c.

C. Correctness

We now provide an informal proof sketch that Todo provides causal consistency by showing that $i)$ reads observe a causally consistent snapshot and $ii)$ writes are atomic.

Lemma 1. *The snapshot time sn_T of a transaction T is always lower than the commit time of T, $sn_T < T.ct$.*

Proof: Let t be a transaction with snapshot time sn_T and commit time ct. The snapshot time is determined during the start of the transaction. The commit time is calculated in the commit phase of the 2PC protocol, as maximum value of the proposed prepare times of all partitions participants in the transaction. In order to reflect causality when proposing a prepare timestamp, each partition proposes higher timestamp than the snapshot timestamp. Thus, the commit time of a transaction, ct, is always greater than the snapshot time, sn_T. ∎

Proposition 1. *If an update u_2 causally depends on an update u_1, $u_1 \rightsquigarrow u_2$, then $u_1.ut < u_2.ut$.*

Proof: Let c be the client that wrote u_2. There are three cases upon which u_2 can depend on u_1, described in Section I-A: *1)* c committed u_1 in a previous transaction; *2)* c has read u_1, written in a previous transaction and *3)* c has read u_3, and there exists a chain of direct dependencies that lead from u_1 to u_3, i.e. $u_1 \rightsquigarrow ... \rightsquigarrow u_3$ and $u_3 \rightsquigarrow u_2$.

Case 1.

∎

Proposition 2. *Writes are atomic.*

Proof: Although updates are made visible independently on each region involved in the commit phase, either all updates are made visible or none of them are, i.e. the atomicity is not violated. All updates from a transaction belong to the same snapshot because they all receive the same commit timestamps. The updates are being installed in the order of their commit timestamps. The visibility of the item versions is determined by the transaction snapsho, which is based on the value of universal stable time. ∎

References

[1] 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.

[2] L. Lamport, “Time, Clocks, and the Ordering of Events in a Distributed System,” *Commun. ACM*, vol. 21, no. 7, pp. 558–565, Jul. 1978.

[3] 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,” 2011.

[4] J. Du, C. Iorgulescu, A. Roy, and W. Zwaenepoel, “GentleRain: Cheap and Scalable Causal Consistency with Physical Clocks,” in *Proc. of SoCC*, 2014.

[5] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen, "Stronger Semantics for Low-latency Geo-replicated Storage," in *Proc. of NSDI*, 2013.

[6] H. Lu, C. Hodsdon, K. Ngo, S. Mu, and W. Lloyd, "The SNOW Theorem and Latency-optimal Read-only Transactions," in *OSDI*, 2016.