Transactions in a Distributed Key-Value Store

6.824 Final Project

Deepak Narayanan, James Thomas, Arjun Srinivasan

November 19, 2014

Introduction

Over the last several years, NoSQL databases such as Redis, Cassandra and MongoDB are becoming increasingly popular because of their performance and scalability benefits over traditional relational databases. Having transactions run on top of a key-value store allows us to use key-value stores in far more effective ways than previously possible. For example, transactions enable atomic test-and-set operations and other basic concurrency primitives.

The main objective of our project was to build a key-value stores capable of supporting cross-shard transactions. We built the functionality on top of the multi-shard key value store from Lab 4.

Our solution to building `Transaction` functionality into key-value stores aims at optimizing performance – we avoided using techniques such as two-phase locking and two-phase commit – so that the system can stay responsive under considerable load.

System Design

Our design is inspired by Robert Escriva's Warp system [1]. As in lab 4, we have a fixed number of shards that are distributed over a variable number of replica groups.

Client API

We support all of the operations from lab 4b and add support for atomic multi-shard transactions. We tried to design the Client API of our system in such a way so as to make it as easy and intuitive to use as possible.

- `StartTransaction()`: This returns a brand new `Transaction` object to which operations such as `Put`, `Get` and `PutHash` can be added. The `Transaction` can then be committed using the transaction's `Commit()` method described below.

- `Transaction.Get(Key)`: This method immediately returns the value `Value` corresponding to the provided key from the key-value server. It also adds a `CheckValue(Key, Value)` operation to the list of the transaction's operations so that when the `Transaction` is committed, the servers can verify that the value for `Key` is still `Value` – this ensures that if `Put` and `PutHash` in a `Transaction` are dependent on the results of `Get` in the same transaction, then the `Transaction` will successfully commit only if the results of those Gets are still valid.

- `Transaction.Put(Key, Value)`: This adds a `Put(Key, Value)` operation to the list of the transaction's operations. The `(Key, Value)` pair is inserted in the key-value server only when the `Transaction` is committed – this ensures that if the `Transaction` is aborted (e.g. due to a failed `CheckValue`), no changes are made to any server state. However, in order to ensure that Gets in a `Transaction` can see its own Puts we store the results of a transaction's put operations in the client's per-`Transaction` local cache – in the `Transaction.Get()` method, we first check to see if a (key, value) pair is in the local cache – only when it is not do we actually make a Get request to the key-value server.

- `Transaction.PutHash(Key, Value)`: A Transaction's `PutHash` operation is very similar to the `PutHash` operation from the labs. First the value corresponding to the provided key is retrieved (let's call it `PreviousValue` for convenience) and then `Hash(Value, PreviousValue)` is computed and stored in the key-value store. As before, the `PutHash` function returns `PreviousValue` back to the user. Under the hood, the `Transaction` object's `PutHash` method is implemented as a `Transaction.Get(Key)` – to obtain `PreviousValue` – and then a `Transaction.Put(Hash(Value, PreviousValue))`. Thus, if at the commit point PreviousValue is not longer the value for Key, the PutHash will fail and the entire `Transaction` will be aborted.

- `Transaction.Commit()`: This method returns true if the `Transaction` successfully commits and false if it aborts. A `Transaction` is aborted if one of its CheckValues fails or if the first server it is sent to does not own the first shard in the transaction. If a `Transaction` successfully commits all of its operation are guaranteed to have been executed, and if it aborts all of its operations are guaranteed not to have been executed.

Consistency Guarantees to Client

We make two important consistency guarantees to clients; we will explain in the next section how these guarantees are achieved.

- Linearizability (**external consistency**) of transactions – transactions appear to occur atomically and in an externally consistent order

- Further clarifying A, we ensure **observed atomicity** – if a Get observes the effects of a transaction, a subsequent Get cannot observe state that has yet to be

updated by the transaction

Server-side Design

Initial Background

A `Transaction` is sent from the client to the server in the following struct (fields not important for this discussion have been elided):

```
type Transaction struct {
  RequestId int64
  ClientId int64
  ShardOps [shardmaster.NShards]ShardOps
  Groups [shardmaster.NShards]int64
  // the group that handled each shard in the forward pass
  CurrentShard int
  Dependencies map[RequestKey]DepInfo
}
```

The entry at index i in `ShardOps` contains the operations in the `Transaction` (in order of appearance in the transaction) that affect shard i.

The client's `Commit()` method sends the `Transaction` to the replica group responsible for the first (lowest-index) shard modified / accessed by operations in the transaction. This replica group initiates what we call the "forward pass" portion of the transaction, in which the `Transaction` passes through the shards it touches in increasing order of shard index and picks up dependencies on conflicting in-flight transactions.

Before continuing, it is important to describe the key data structure on the server side: `inFlightTransactions [shardmaster.NShards] map[string] map[RequestKey] InFlightTransactionInfo`, which keeps track of dependency information on a per-key basis. `InFlightTransactionInfo` is a list of maps, such that each map in the list corresponds to a shard. Each map in this list maps a key served by that shard to another map called `keyTransactionsInfo` that contains important per-key dependency information. For each key, `keyTransactionsInfo` maps the `RequestKeys` of transactions that touch the key to information about the `Transaction` (including a list of other transactions the `Transaction` depends on).

Forward Pass

When a server receives a forward pass RPC for a particular shard as a part of some `Transaction` T0, it first commits the operation in its replica group's Paxos log, applying all unapplied operations in prior log slots. This ensures that all replicas in the same replica group execute all operations in the same order.

Assuming that T0 has not already been aborted at a previous shard, it then continues through the forward pass by executing the following steps:

1. It first checks whether it owns the shard; if it does not, it immediately returns, indicating the error to the caller. The caller then tries to update its config from the shardmaster, and then try tries to resend the `ForwardPass` RPC.

2. It then iterates through all of T0's CheckValues for the shard and verifies them (makes sure that the key-value mapping is unchanged in the replica's current version of the key-value store and also that no in-flight transactions are set to modify the value at the key); if any of the CheckValues fail, T0 must be aborted and we skip to step 4

3. For each `Transaction` operation (a Put for key K and value V) for this shard, iterate through the `inFlightTransactions` map for the shard and add any uncommitted (i.e. not yet committed or aborted in the backward pass) transactions that affect K into T0's Dependencies set (see the definition for the `Transaction` struct above), as long as these transactions are not themselves already dependent on T0 (to prevent dependency cycles). An entry for a `Transaction` in `inFlightTransactions` contains its full set of transitive dependencies (i.e. none of a `Transaction` Tx's dependencies are themselves dependent on transactions not in Tx's dependency list), so we can immediately evaluate whether adding a dependency causes a cycle.

   Each dependency is associated with an instance of the `DepInfo` struct, which simply holds the operations that the dependency performs at each shard (a copy of the `ShardOps` array). We also make T0 dependent on all of the transitive dependencies of these first-level dependencies (since it is possible for T0 and a dependency T2 of a first-level dependency T1 to affect a key K' that is not touched by the T1, and we want to ensure complete linearizability, i.e. T2 happens before T1 happens T0; adding second-level dependencies also prevents dependency cycles), which, as mentioned before, are all present in the first-level dependencies' dependency lists. Thus, in assembling T0's dependency list, we maintain the invariant that all transitive dependencies are contained in transactions' dependency lists.

4. We add T0 to `inFlightTransactions` for all of the keys it affects (including keys involved only in CheckValues) so that future transactions can be dependent on it.

5. We cede the server's mutex (it has been held throughout so far), so that more operations can be committed to the Paxos log and applied, and spawn a new routine to try to send the `Transaction` (as well as information about whether it has aborted) on to the replica group that owns the next shard (in order of shard index) that it affects. We loop all of the groups listed in our current configuration until a server tells us that it owns the shard. This loop will never terminate if a new group has joined and now owns the desired shard (we cannot advance our configuration to see this new group, as explained below), so we do not currently support concurrent joins and transactions.

If the `Transaction` was already aborted at a previous shard, we can skip directly to step 5.

The high-level idea of the forward pass is to validate the CheckValues at all shards before committing any operations, and simultaneously establish a strict dependency order among conflicting in-flight transactions to ensure linearizability. We ensure that we do not establish cyclic dependencies by storing a transaction's full transitive dependency list, and also by passing through the shards in a consistent order (from lowest to highest index). Passing through the shards in a consistent order means that if we are about to make a `Transaction` T1 dependent on a `Transaction` T2 at server SR/shard SH, then if `Transaction` T2 is already dependent (directly or transitively) on T1, T1 will be present in T2's dependency list at SR because T2 must have picked up the dependency on T1 at a shard prior to SH in the ordering, since T1 can only have previously reached shards prior to SH. So we will not establish any cyclic dependencies. (One more case about cyclic dependencies remains to be proven at the end of the description of the backward pass.)

Backward Pass

Once we have made the forward pass through all of the shards affected by a transaction, we start the backward pass, where we actually execute the operations of transactions that were not aborted during the forward pass. The backward pass passes through the shards in reverse order of shard index.

When a server receives a backward pass RPC for a particular shard that a `Transaction` T0 modifies, it first commits the operation in its replica group's Paxos log, applying all unapplied operations in prior log slots. (As with the `ForwardPass`,this helps ensure that all operations are performed in the same order on every replica in the replica group) Assuming that the `Transaction` was not aborted, it then applies the backward pass by executing the following steps:

1. A `Transaction` cannot be committed at a shard until all of its dependencies are committed first. All transactions that modify any of the same keys that the current `Transaction` does at this shard (the keys that each dependency modifies can be determined from the associated DepInfo) and in the current transaction's dependency list must have their status in `inFlightTransactions` be changed to committed or aborted (which happens at the end of the dependencies' backward passes at this shard) before the current `Transaction` can make progress. While this spinning occurs, the server's lock is released to allow further operations to be added to the Paxos log and applied, which is necessary if progress is to be made on committing or aborting the dependencies. If we don't cede the lock, our code will deadlock and won't be able to make any forward progress.

2. Once all the dependencies are committed or aborted, we apply all of the Puts in T0 (with the lock held, of course).

3. We mark T0 as committed in all places in `inFlightTransactions` where it appears. We cannot simply remove T0 from `inFlightTransactions` because transactions that depend on T0 need to know that T0 has been committed when they

reach this shard in their backward passes. Marking T0 as committed also ensures that no new transactions that pass through this shard in their forward passes add T0 as a dependency.

4. We spawn a new thread to pass the `Transaction` on to the next shard affected by the transaction. In the forward pass we store the group ID of the replica group that handles each shard (see the Groups array in the `Transaction` struct in part A of this section), and we know that the replica group will still own the shard since replica groups cannot transfer away shards for which they have in-flight transactions (see transfers discussion below).

If T0 was aborted, we can skip directly to step 3 and mark everything as aborted rather than committed.

A thread on the server the client originally sent the `Transaction` to, spins and waits for the backward pass to complete, at which point it notifies the client about whether `Transaction` committed successfully or not.

There is one final cyclic dependency case that we must prove is impossible. In particular, suppose we make `Transaction` T2 dependent on T1 at a shard Sx. Then T1 could go on and become dependent at a later shard Sy on a `Transaction` T3 that depends (directly or transitively) on T2. One might think that a deadlock would occur in this case – T2 would spin at Sx in the backward pass waiting for T1 to be committed or aborted at Sx, but T1 would spin at Sy waiting for T3 to commit, which wouldn't happen because its dependency T2 would spin infinitely at Sx. This explanation already hints at its own flaw – since transactions commit in reverse shard order, T3's completion at Sy can only depend on the completion of transactions at Sy or later shards (the dependencies of T3 simply need to reach Sy for T3 to be committed at Sy). Since Sy is later in the shard order than Sx, the completion of T3 at Sy cannot possibly be dependent on the completion of T2 at Sx.

Comments about Paxos Log, Fault Tolerance and At-most once semantics

Since the forward and backward pass operations are put in the Paxos log, every replica attempts to execute them, and only one of them will come first and actually do useful work. It is important that we do this so that we can tolerate many machine failures. But in the case that most machines are working, many extraneous attempts to execute each forward and backward pass operation will be made, and there will be many extraneous RPC calls to replica groups controlling later shards in transactions. Thus, we make sure to properly detect duplicate forward and backward pass operations and return success to the caller in those cases (we only store results in the duplicates map on success, i.e. when the shard is owned).

Note that if a `ForwardPass` RPC succeeds (that is, `ForwardPass.Success = true`), then we guarantee that all downstream ForwardPasses / BackwardPasses will execute at some point since we assume that at all points in time, a majority of replicas in each replication group remain alive. A similar guarantee is made for `BackwardPass` RPCs as well, this ensures that if we see a `ForwardPass` or `BackwardPass` that we've

already processed before, we know we can return Success without having to worry about re-executing downstream Forward and Backward Passes.

Since we always place an operation in the first available slot in the Paxos log, we ensure that our Paxos log has no holes, which ensures external consistency.

Shard Transfers

If a transfer of a shard has already started (we propose an `InitiateTransfer` operation in the Paxos log to indicate that a transfer is beginning), we do not allow any operations (within transactions or otherwise) to affect the shard so that the recipient of the shard never sees a stale copy – that is, as soon as a `Transfer` is initiated, we stop serving requests to that shard until the `Transfer` has been completed.

In addition, we don't allow transfers to be initiated until there are no more in-flight (uncommitted) transactions for a shard. This ensures that if a replica group handles a shard on the forward pass, it will still own that shard during the backward pass, a fact that the backward pass logic depends on. This means shard transfers are generally slow to happen and replica groups cannot move through configs quickly, but we think this is a reasonable assumption to make – in real systems, config changes are fairly infrequent.

Handling of Non-`Transaction` Puts and Gets

We essentially turn non-`Transaction` puts and gets into one-operation transactions that wait out any in-flight transactions that affect their keys, since these transactions may expect that the key state will be unchanged when they return to this shard for the backward pass (i.e. they have CheckValues for this key) – if Gets and Puts did not respect the per-key dependency information stored in inFlightTransactions, then we would reach a state where a `Transaction` already validated in the `ForwardPass` would become invalid (because CheckValues no longer return true), but since we do no validation whatsoever in the BackwardPass, we would be oblivious to this invalidation and transactions would up committing with wrong results.

Non-`Transaction` Puts and Gets are also entered into `inFlightTransactions` so other transactions or Puts/Gets become dependent on them. In particular, it is important for gets to be dependent on in-flight transactions so that if one get shows that a `Transaction` has happened, then a subsequent get cannot show state from prior to the execution of the `Transaction` (this is our second guarantee to clients).

Comparison to Other Approaches

One common way to add `Transaction` support to a system is to use two-phase commit and two-phase locking. We claim that our approach is more efficient than two-phase commit because there are fewer messages sent between the client and server machines. In particular, in two-phase commit, if a replica group is sent a `Transaction` T1 that modifies a particular key and has already committed to another `Transaction` T2 that modifies that key, it will send a failure message back to the client and the client will have

to retry T1 later. In our design, T1 will simply sit at the replica group waiting for T2 to complete execution – the client-server network latency is not incurred, and the client does not have to devise policies for when to retry. In short, with our design, aborts happen only when absolutely necessary – when a CheckValue fails (or when the server the `Transaction` is initially sent to does not control the first shard in the transaction, an uncommon case). Two-phase commit may be necessary for more complicated relational data models, but in the key-value store case it appears that more efficient protocols are feasible.

Testing Approach

We wrote a number of tests to convince ourselves that our implementation indeed works. We describe these tests in greater detail below.

- **Basic test**: This test aims at making sure that all the basic `Transaction` functionality works as expected. We sequentially execute all the basic operations – Gets, Puts, PutHashes and Moves – among other things, this ensures that we can test if all state that needs to be passed on between Shards during transfers is being passed on correctly.

- **Limp test**: Multiple servers in each replica group are killed at random, to make sure that progress with operations can be still made even without every server in a replica group being alive.

- **TestAndIncrement**: In this test, multiple threads try to read a value corresponding to a particular key and then increment that value by 1. By initializing the value corresponding to that key to be initially 0, we can check if our system correctly implements TestAndSets by ensuring that the final value corresponding to the key equals the number of threads.

- **Multi-Shard invariants**: In this test we try to test the atomicity of `Transaction` operations. By checking if dependencies across multiple shards are maintained we can show that `Transaction` operations cannot interleave with each other (for example `Transaction` 1 executes before `Transaction` 2 on shard 1 but `Transaction` 2 executes before `Transaction` 1 on shard 2).

  We first set up an invariant across multiple keys split across multiple shards – for example the sum of values corresponding to the keys is a certain value. Multiple threads then make modifications to the values stored in the key-value store such that each modification still respects the global invariant that the sum of all values is a constant. At the end of the multiple concurrent updates, we check to see if the global invariant across the multiple keys still holds.

- **Concurrent**: This test is very similar to the staff-provided Concurrent test – it attempts to interleave transactions containing Puts, Gets and PutHashes with Moves – all of these operations are attempted simultaneously in multiple threads.

In addition, to make sure that normal Puts, Gets and PutHashes cannot interleave with operations part of a Transaction, we had to tweak our implementation of the standard `Get`, `Put` and `PutHash` operations (this is described in greater detail in the previous section). To make sure that we didn't suffer a regression in functionality, along with the new `Transaction` tests we introduced, we also ran the Lab4b tests.

Applications

Cross-shard and cross-table atomic transactions have a number of useful applications. One example is with derived tables – tables whose data / contents are "derived" from other base tables. Updates to records stored in the base tables would need to be propagated to the derived tables atomically as well – in such a scenario, cross-table atomic transactions really help in preventing multiple tables with the same data going out-of-sync from each other. Some examples of derived tables include reverse indices (here the "base" table would be the primary index, and any change to the primary index would need to be reflected in the reverse index) and the Bids and Items example presented in the Lynx paper (where Items contains a consolidated list of all objects and Bids contains the bidding history for each object)

Future Work

Over time, the `inFlightTransactions` data structure stored at each server in our distributed key-value system can become arbitrarily large as more transactions are committed. We can remove all information about a `Transaction` in `inFlightTransactions` once all transactions that have dependencies on that `Transaction` complete their BackwardPass. We refer to the process of removing information about already committed transactions garbage collection. Due to a lack of time, we haven't yet implemented garbage collection in our system, however this remains one of our major priorities for future work.

For reasons highlighted above, our system also doesn't allow Joins to be executed concurrently with transactions (we currently only support Moves and Leaves) – tweaking our protocol to support this feature is another area for future work. Finally, we are very interested in comparing our approach in terms of performance with other approaches such as two-phase commit – time didn't allow us to actually implement a solution that used two-phase commit. Such an exercise could lead to extremely interesting results since all such results would be quantitative instead of purely qualitative – we would be able to run the different solutions on multiple benchmarks to see how the two approaches match up with each other.

References

[1] Warp: Multi-Key Transactions for Key-Value Stores. Robert Escriva, et. al.. Technical Report, Nov 2013.