CS 505: Distributed Transactional Protocols in a Replicated Data Store

Prashant and Samodya

1 System Overview

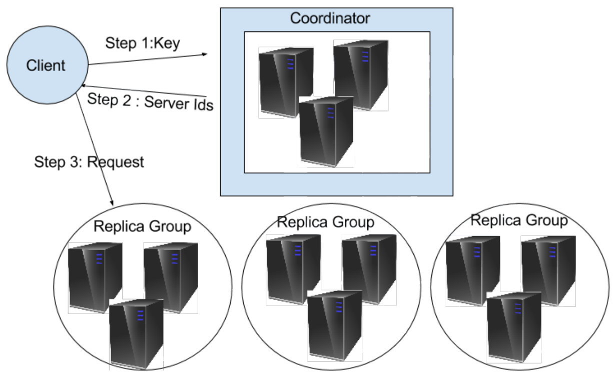

Figure 1: System Architecture

For the purpose of this report, we will assume that a replication group refers to a set of servers that replicate a key-value among themselves. The Coordinator is also a set of servers, but they differ from replication groups in that a configuration is agreed upon by these servers. The configuration refers to a system state that maintain such information as, the replication group responsible for a given key and the servers in that replication group. Keys are partitioned among replication groups equably and the responsibility for this is placed on the Coordinators. So, when a request is made by the client, it first goes to any of the coordinator servers. The coordinator servers in turn respond to the client with information about which replication group (and its corresponding servers) are responsible for that key. For any future requests regarding a set of keys, the client would use this information to contact any server in the replication group responsible for the keys. The operations we support currently are get, put and delete. The code base is primarily in Java with some shell scripts to invoke tests. For communication, we use JAVA RMI, which is the remote procedure call implementation of JAVA.

Motivation for such an architecture came from the design of such sophisticated systems as the Apache HBase , Google Spanner and HyperDex. They all use a coordination, configuration and replication model under the hood. Replication provides reliability to the system in the event of network failures and multiple coordinator servers helps decentralize the system, although not entirely. Originally, we sought out to build a completely decentralized system such as Chord and apply transactional protocols to it, but in the interest of time chose to simplify the project to a partially decentralized system.

Now, onto the main requirements for this project. We aim to build different distributed transactional protocols in a fully asynchronous setting using the above described architecture subject to non-Byzantine faults. More information on these protocols is provided in the subsequent sections, Using this framework we wish to provide functionality for a replication group to join or leave the network freely. Replication groups may join so as to provide scalability to the system , specifically, throughput by load balancing the keys across the different replication groups equably. Replication groups may also leave in the event of the need for maintenance or repair work. We wish to realize a system that can handle transactions in the presence of such re-configurations.

Replication for the coordinator servers that need to agree on a configuration, and the replication of key-value pairs within a replication group are performed by using Classic Paxos protocol. We have, during the process of writing identified that the system uses too many messages to accomplish some tasks. Hence, to reduce number of message delays incurred we have introduced the Fast Paxos protocol.

Since we know that request to join or leave the system by replication groups to the Coordinator servers are very rarely going to collide, we could implement a flavor of Paxos that reduces one message delay in achieving consensus, called Fast Paxos. With this we could send out requests directly to the acceptors, bypassing the need to go through the proposing phase. This however, needs that quorum achieved during the accepting phase is 3t+1 instead of 2t+1, as in Classic Paxos, where t is the maximum number of servers that can fail.

2 Transactions

In this project we strived to implement transactions with ACID (Atomic, Consistent, Isolated, Durable) properties using mutiple transaction protocols. As well as contention management, in a distributed setting performance concerns arise as there is the need for coordination between a group of servers to come to consensus on whether the decision of the transaction is to be committed or aborted. We are implementing three transactional commit protocols: two-phase commit protocol [1] [5], paxos commit protocol [5] and acyclic transaction commit protocl [3].

2.1 Client Implementation

In our simple implementation we provide with a client that can execute an input script provided by a user. Client at the moment support a basic instruction set of START_TRANSACTION, COMMIT_TRANSACTION wrappers, ADDI, PRINT utilities as well as the basic PUT, GET, DELETE instructions. The instruction set also includes support for local registers identified with $NAME. With the consideration of the discussion in "On Optimistic Methods for Concurrency Control" [6] we decided to implement optimistic transactions. Therefore the client was implemented in a manner that a transaction would be executed optimistically (ie. client executes the instructions optimistically as if any other concurrent transaction would not conflict with its transaction). Any aborted transaction will be retried by backing up and re-executing from its original local state. Client periodically queries the coordinator for the state of the system to get to know which replica group owns each key/shard.

Listing 1: Pseudo code of the client (only essential transaction parts included)

```
on GET x:
  replicaGroup = keyToReplicaGroup(x)
  loop replicaGroup.members as member:
    success, reponse = member.GET(x)
    if success:
      ltransaction.store.x = response.x
      return

on PUT x value:
  ltransaction.store.x = value

on START_TRANSACTION:
  new ltransaction
  ltransaction.start = currentLine

on COMMIT_TRANSACTION:
  decision = TryCommitTransaction(ltranscation) # protocol dependent
  if decision = ABORT:
    nextInstruction = ltransactionStore.start
  else:
    localStore.merge(ltransactionStore.store)
```

2.1.1 Analysis

As we can observe client GETs are optimistic in the sense that server will respond with the value, irrespective of concurrent transactions on the same key. PUT instructions are handled with in the local store of the client. Within COMMIT_TRANSACTION only the client will synchronize exactly with the server. An aborted transaction re-executes the same instructions, essentially until transaction is committed.

Within the scope of the client, we can claim liveness guarantee of wait freedom (a transaction will be decided in bounded number of steps), yet in the context of transactions, aborted transactions can not be accounted as real progress. Local progress [2] (every transaction commits in finite steps) cannot be guaranteed, but obstruction free termination can be guaranteed as the client will make progress as long as there is no other conflicting transaction. Safety guarantee expected is serializability of committed transactions, and for that we need to examine the commit protocols implemented.

2.2 Optimistic Two-Phase Commit Protocol

The general idea of two-phase commit [1] [5] is conveyed through following picture from [5]. RMs are the resource managers which in our case are the replica groups. TM is the transaction manager which we chose as the fitting role for the client. Transaction manager (TM) initiates a transaction by sending PREPARE message to each RM and each Resource Manager (RM) will send a reply back with whether the transaction can be committed or aborted from their view. If the TM did not get any response to abort the TM sends a decision to all RMs to commit otherwise send an abort. Two-Phase commit protocol [6]

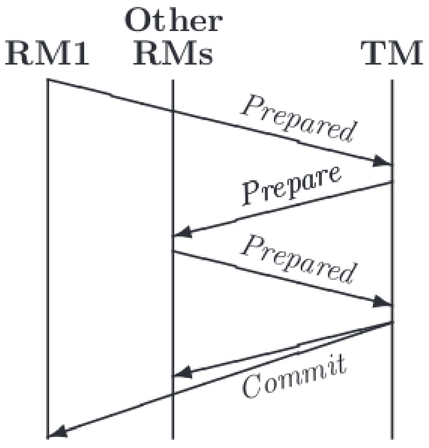

Figure 2: Generic two phase commit from [5]

was adapted to the a practical setting of shard replicated key-value store. Any RM is a member of a replication group where each member of the replication group have come to agreement on the values it has stored, ongoing transactions as well as decided transactions. For this purpose we implemented a Paxos log so as long as there is a majority of correct RMs in a replica group the transaction can be decided correctly. To support concurrent transactions of non overlapping keys, a row (key) locking mechanism is used. A transaction can obtain an exclusive write lock on a key (if it is in the write-set) or shared read-lock on a key (if it is only in the read-set).

Listing 2: Simplified two phase commit pseudo code for Client

```
TM on TryCommitTransaction:
  replicaGroups = keysToReplicaGroup(ltransaction.store.keyset())
  # in paralell
  send transaction context to each replica group
  wait for reponse from all replica groups
  if any replica group responded with abort:
    decision = ABORT
  else:
    decision = COMMIT
  # in parallel
  send transaction context and decision to all replica groups
  return decision
```

Listing 3: Simplified two phase commit pseudo code for DbServer: Try Commit

```
RM (2PC version) on TryCommitTransaction:
  # use paxos to come to consensus among all members in replica group
  # to process this request
  # a Paxos replicated log was implemented and used
  Get sequence number from Paxos instance
  Propose the client transaction context
  # ... once the replica group accepts
  HandleTryCommitTransaction(clientTransactionContext)

on HandleTryCommitTransaction(transactionContext):
  # server is in the process of transferring keys
  if server.state is TRANSFER:
    reply ABORT
  per each key:value in transactionContext.readset():
    if key is write locked by another concurrent transaction:
      reply ABORT
    if key:value does not match local key:value:
      reply ABORT
  per each key:value in transactionContext.writeSet():
    if key is read locked by another concurrent transaction:
      reply ABORT
    if key is write locked by another concurrent transaction:
      reply ABORT
  # else ready to accept
  per each key in transactionContext.readset():
    put read lock on key with transcation id
  per each key in transactionContext.writeset():
    put write lock on key with transaction id
  reply ACCEPT
```

Listing 4: Simplified two phase commit pseudo code for DbServer: Decide Commit

```
RM (2PC version) on DecideCommitTransaction:
  Get sequence number from Paxos instance
  Propose the client transaction context
  # ... once the replica group accepts
  HandleDecideCommitTransaction(clientTransactionContext)

on HandleDecideCommitTransaction(transactionContext, decision):
  if decision is COMMIT:
    merge writeset with localstore
  per each key in transactionContext.readset():
    release read lock on key with transcation id
  per each key in transactionContext.writeset():
    release write lock on key with transaction id
```

2.2.1 Analysis

In two phase commit protocol, the first phase is the pseudo code listing 3, second phase is the pseudo code listing 4. The first phase is the locking phase which is very important in ensuring safety. Concurrent transactions are ordered by the locking mechanism, Shared locks for key reads, and Mutually exclusive locks for key writes are utilized. Serilizability of committed transactions can be guaranteed as if keys are not in contention, the transactions can progress concurrently. If the read set seen by the client transaction at the time of GET is not compatible with the current state of the Resource Manger (DbServer) then transaction is aborted. If keys are in contention, a write on a key requires a mutually exclusive lock on that key, and if that is not obtainable the transaction is aborted. And in the same way a read on a key requires a shared key lock and, and if that is not obtainable the transaction is aborted. So by making sure that first transaction to obtain exclusive locks gain precedence and other concurrent transactions must be retried, we are able to guarantee serializability of committed transactions.

Although it seems the process is RM crash failure tolerant because of the use of Paxos log based replica group, a concern from the above steps, what is the outcome if the client (TM) fails to respond. We can observe that in the case client failure the above mentioned protocol will lead to permanently locked keys on the RMs. This is one of the main drawbacks inherent to the basic two phase commit protocol itself. Therefore next we will look into how the above process can be tuned to handle TM failure with Paxos commit protocol.

2.3 Paxos Commit

Paxos Commit described in Consensus on Transaction Commit by Jim Gray and Leslie Lamport [5], fixes the main drawback of two phase commit protocol. The general idea is to have a set acceptors in addition to the TM, which acts as the mediators between RMs and TM.

To adapt to our project setting we extended the two phase commit implementation to introduce a set of acceptor servers that were selected by the client who still acts as the TM. Each accpetors collects the decisions from all replica groups involved and send their decision to TM. TM gets the majority decision from the acceptors and sends decision message to all replica group. In the case of TM failure, a leader is selected among the acceptors, which sends the final decision to each replica group.

2.4 Acyclic Transaction Commit Protocol

We next look at the novel approached proposed in Warp [4]. The key insight for this protocol, according to the authors, is to arrange the servers involved in the transaction to be in a chain, and to validate and order the transactions using a dynamically determined number of passes through this chain.

To give a brief overview of the protocol:

1. Client optimistically performs the GET and PUT requests [6], then client library constructs chain of nodes that operate on keys in the transaction on transaction commit and send request to first in chain.
2. Transactions with keys in common will pass through common set of nodes. First server in common between two transactions can determine the relative ordering.
3. In the forward pass validation will be performed at each server. Validation involves checking read set of client with data store and each value with that of local state of previously validated transactions. If validation was a success, pass valid forward in chain, if not backward pass abort.
4. The last node in chain will start a backward pass with a commit message. The backward pass ends at the first node in chain which will respond the client with abort or commit.

With chaining the protocol ensures that there is at most one replica group that is processing this transaction, thus provides more parallelism among concurrent transactions (the details on handling concurrent transactions are not mentioned here in this report, refer to [4, p. 4]). Due to time constraints we couldn't finish the coding this protocol and hence, have refrained from prov

3 Testing

First we needed to make sure the correctness of transactions, therefore a simple test case was made with the intention of concurrent transactions that they all work on the same keys to have the highest contention. The process of testing is documented in the README file, but the general idea was multiple bank accounts initialized to 0 that get increased by 10 concurrently. Depending on the number of transactions started the end result must be 10 * (number of transactions started).

Next is performance comparison of key-value store without transactions vs with transactions on normal GET PUT operations. Code associated has the tools to regenerate the results and the steps involved are in the README provided.

4 Final Project Status

We completed writing the code for the above described coordinator-replica key partitioning model, Classic Paxos protocol, Fast Paxos protocol and a simple key value store. We have tested the 2PC protocol for correctness given concurrent transaction requests assuming replication groups don't join or leave the system after starting configuration in a partial synchronous crash-stop model.

References

[1] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1987.

[2] V. Bushkov and R. Guerraoui. *Transactional Memory. Foundations, Algorithms, Tools, and Applications: COST Action Euro-TM IC1001*, chapter Liveness in Transactional Memory, pages 32–49. Springer International Publishing, Cham, 2015.

[3] R. Escriva, B. Wong, and E. G. Sirer. Warp: Lightweight multi-key transactions for key-value stores. *CoRR*, abs/1509.07815, 2015.

[4] R. Escriva, B. Wong, and E. G. Sirer. Warp: Lightweight multi-key transactions for key-value stores. *CoRR*, abs/1509.07815, 2015.

[5] J. Gray and L. Lamport. Consensus on transaction commit. *ACM Transactions on Database Systems (TODS)*, 31(1):133–160, 2006.

[6] H. T. Kung and J. T. Robinson. On optimistic methods for concurrency control. *ACM Trans. Database Syst.*, 6(2):213–226, June 1981.